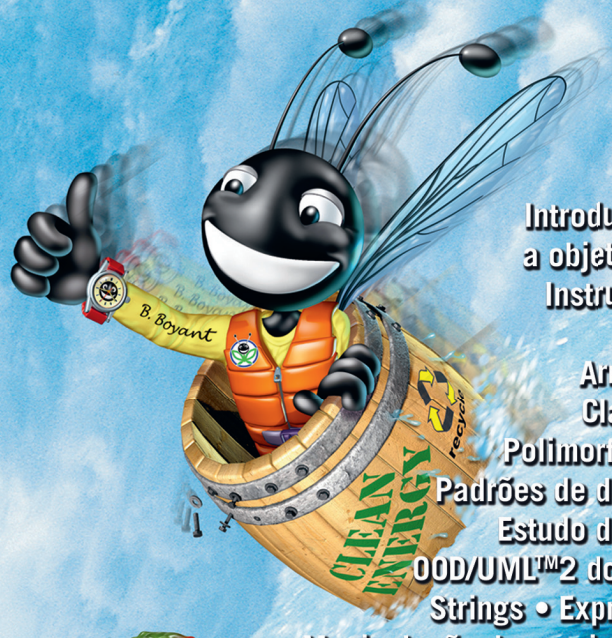
Java™ COMO PROGRAMAR

DEITEL®

8ª edição



E/S
Tipos
Operadores
Introdução à orientação a objetos desde o início
Instruções de controle
Métodos
Arrays • ArraysLists
Classes • Heranças
Polimorfismo • Interfaces
Padrões de design • Exceções
Estudo de caso do projeto OOD/UML™2 do caixa eletrônico
Strings • Expressões regulares
Manipulação de arquivos • Depuradores
GUI Swing • Nimbus • Aplicativos gráficos
Applets • Multimídia • Java Web Start
Redes • Threads • SwingWorker
Recursão • Pesquisas • Ordenação
Estruturas de dados • Genéricos • Coleções
JDBC™ • Java DB • MySQL®
Aplicativos Web • Ajax • Serviços Web

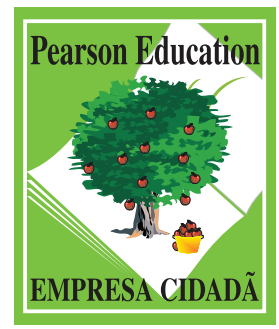
PAUL DEITEL
HARVEY DEITEL

ALWAYS LEARNING
PEARSON

Java™

COMO PROGRAMAR

OITAVA EDIÇÃO



Java™ COMO PROGRAMAR

OITAVA EDIÇÃO

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Tradução

Edson Furmankiewicz
Docware Traduções Técnicas

Revisão técnica

Fábio Luis Picelli Lucchini
Bacharel em Ciência da Computação pelo Centro Universitário Padre Anchieta
Mestre em Engenharia da Computação pela Unicamp (FEEC)
Professor Universitário do Centro Universitário Padre Anchieta

PEARSON

abdr ASSOCIAÇÃO BRASILEIRA DE DIREITOS REPROGRÁFICOS
Respeite o direito autoral

São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Peru Porto Rico Venezuela

© 2010 by Pearson Education do Brasil

© 2009 by Prentice Hall, uma empresa do grupo Pearson Education, Inc.

Tradução autorizada a partir da edição original em inglês, *Java™ How to Program -- Eighth Edition*, de Harvey M. Deitel e Paul J. Deitel, publicada pela Pearson Education Inc., sob o selo Prentice Hall PTR.

Todos os direitos reservados. Nenhuma parte desta publicação pode ser reproduzida ou transmitida de qualquer modo ou por qualquer meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Muitas das designações utilizadas por fabricantes e vendedores para distinguir seus produtos são protegidas como marcas comerciais. Onde essas aparecem no livro, e a Manning Publications estava ciente de uma proteção de marca comercial, as designações foram impressas com a primeira letra ou todas as letras maiúsculas.

Os nomes de empresas e produtos mencionados neste livro são marcas comerciais ou registradas de seus respectivos proprietários.

Diretor editorial: Roger Trimer

Gerente editorial: Sabrina Cairo

Supervisor de produção editorial: Marcelo Françoso

Editoras: Gabriela Trevisan e Marina S. Lupinetti

Revisão: Letícia Scarp, Adriane Schirmer e Norma Gusukuma

Capa: Casa de ideias sobre o projeto original de Abbey S. Deitel, Harvey M. Deitel, Francisco Santalucia e Kristine Carney

Editoração eletrônica: Docware Traduções Técnicas

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Deitel, P. J.

Java : como programar / Paul Deitel e Harvey Deitel ; tradução Edson Furmankiewicz ; revisão técnica Fábio Luis Picelli Lucchini. -- 8. ed. -- São Paulo : Pearson Prentice Hall, 2010.

Título original: Java : how to program.
ISBN 978-85-7605-563-1.

I. Java (Linguagem de programação para computador) I. Deitel, Harvey. II. Lucchini, Fábio Luis Picelli. III. Título.

09-09979 CDD-005.133

Índice para catálogo sistemático:

I. Java : Linguagem de programação :
Computadores : Processamento de dados
005.133

2ª reimpressão – novembro 2012

Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil Ltda.,
uma empresa do grupo Pearson Education

Rua Nelson Francisco, 26
CEP 02712-100 – São Paulo – SP – Brasil
Fone: 11 2178-8686 – Fax: 11 2178-8688
e-mail: vendas@pearson.com

Em memória de Kristen Nygaard, coinventor do Simula — a primeira linguagem de programação orientada a objetos do mundo.

Paul e Harvey Deitel

Sumário

Prefácio xvii

Antes de você começar xxvii

1 Introdução aos computadores, à Internet e à World Wide Web 1

1.1 Introdução ... 2
1.2 Computadores: Hardware e software ... 3
1.3 Organização do computador ... 3
1.4 Primeiros sistemas operacionais ... 4
1.5 Computação pessoal distribuída e computação cliente/servidor ... 4
1.6 A Internet e a World Wide Web ... 5
1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível ... 5
1.8 História do C e do C++ ... 6
1.9 História do Java ... 6
1.10 Bibliotecas de classe do Java ... 7
1.11 Fortran, Cobol, Pascal e Ada ... 7
1.12 Basic, Visual Basic, Visual C++, C# e .NET ... 8
1.13 Ambiente típico de desenvolvimento Java ... 8
1.14 Notas sobre o Java e este livro ... 11
1.15 Testando um aplicativo Java ... 11
1.16 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML ... 15
1.17 Web 2.0 ... 17
1.18 Tecnologias de software ... 18
1.19 Conclusão ... 19
1.20 Recursos da Web ... 19

2 Introdução aos aplicativos Java 28

2.1 Introdução ... 29
2.2 Nosso primeiro programa Java: imprimindo uma linha de texto ... 29
2.3 Modificando nosso primeiro programa Java ... 34
2.4 Exibindo texto com `printf` ... 35
2.5 Outro aplicativo: somando inteiros ... 36
2.6 Conceitos de memória ... 40
2.7 Aritmética ... 41
2.8 Tomada de decisão: operadores de igualdade e operadores relacionais ... 43
2.9 Conclusão ... 47

3 Introdução a classes e objetos 56

3.1 Introdução ... 57
3.2 Classes, objetos, métodos e variáveis de instância ... 57
3.3 Declarando uma classe com um método e instanciando um objeto de uma classe ... 58
3.4 Declarando um método com um parâmetro ... 61
3.5 Variáveis de instância, métodos *set* e *get* ... 63
3.6 Tipos primitivos versus tipos por referência ... 67
3.7 Inicializando objetos com construtores ... 68
3.8 Números de ponto flutuante e tipo `double` ... 70
3.9 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando caixas de diálogo ... 73
3.10 Conclusão ... 75

4 Instruções de controle: Parte I 81

4.1 Introdução ... 82
4.2 Algoritmos ... 82
4.3 Pseudocódigo ... 82
4.4 Estruturas de controle ... 83
4.5 A instrução de seleção única `if` ... 84
4.6 A instrução de seleção dupla `if...else` ... 85
4.7 A instrução de repetição `while` ... 88
4.8 Formulando algoritmos: repetição controlada por contador ... 89
4.9 Formulando algoritmos: repetição controlada por sentinela ... 93
4.10 Formulando algoritmos: instruções de controle aninhadas ... 99
4.11 Operadores de atribuição composta ... 102
4.12 Operadores de incremento e decremento ... 102
4.13 Tipos primitivos ... 105
4.14 (Opcional) Estudo de caso de GUI e imagens gráficas: criando desenhos simples ... 105
4.15 Conclusão ... 108

5 Instruções de controle: Parte 2 119

5.1 Introdução ... 120
5.2 Princípios básicos de repetição controlada por contador ... 120
5.3 Instrução de repetição `for` ... 121
5.4 Exemplos com a estrutura `for` ... 124
5.5 Instrução de repetição `do...while` ... 127
5.6 A estrutura de seleção múltipla `switch` ... 129
5.7 Instruções `break` e `continue` ... 134
5.8 Operadores lógicos ... 136
5.9 Resumo de programação estruturada ... 140
5.10 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando retângulos e ovais ... 144
5.11 Conclusão ... 147

6 Métodos: uma visão mais aprofundada 154

6.1 Introdução ... 155
6.2 Módulos de programa em Java ... 155
6.3 Métodos `static`, campos `static` e classe `Math` ... 156
6.4 Declarando métodos com múltiplos parâmetros ... 158
6.5 Notas sobre a declaração e utilização de métodos ... 161
6.6 Pilha de chamadas de método e registros de ativação ... 161
6.7 Promoção e coerção de argumentos ... 162

6.8 Pacotes da Java API ... 163
6.9 Estudo de caso: geração de números aleatórios ... 164
6.10 Estudo de caso: um jogo de azar; introdução a enumerações ... 168
6.11 Escopo das declarações ... 172
6.12 Sobrecarga de método ... 174
6.13 (Opcional) Estudo de caso de GUI e imagens gráficas: Cores e formas preenchidas ... 176
6.14 Conclusão ... 179

7 Arrays e ArrayLists 189

7.1 Introdução ... 190
7.2 Arrays ... 190
7.3 Declarando e criando arrays ... 191
7.4 Exemplos que utilizam arrays ... 192
7.5 Estudo de caso: simulação de embaralhamento e distribuição de cartas ... 199
7.6 A estrutura `for` aprimorada ... 202
7.7 Passando arrays para métodos ... 203
7.8 Estudo de caso: classe `GradeBook` utilizando um array para armazenar notas ... 205
7.9 Arrays multidimensionais ... 209
7.10 Estudo de caso: classe `GradeBook` utilizando um array bidimensional ... 212
7.11 Listas de argumentos de comprimento variável ... 217
7.12 Utilizando argumentos de linha de comando ... 218
7.13 Classe `Arrays` ... 219
7.14 Introdução a coleções e classe `ArrayList` ... 221
7.15 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando arcos ... 223
7.16 Conclusão ... 225

8 Classes e objetos: uma visão mais aprofundada 241

8.1 Introdução ... 242
8.2 Estudo de caso da classe `Time` ... 242
8.3 Controlando o acesso a membros ... 245
8.4 Referenciando membros do objeto atual com a referência `this` ... 246
8.5 Estudo de caso da classe `Time`: construtores sobrecarregados ... 248
8.6 Construtores padrão e sem argumentos ... 252
8.7 Notas sobre os métodos *set* e *get* ... 252
8.8 Composição ... 253
8.9 Enumerações ... 256
8.10 Coleta de lixo e o método `finalize` ... 258
8.11 Membros da classe `static` ... 258
8.12 Importação `static` ... 261
8.13 Variáveis de instância `final` ... 262
8.14 Estudo de caso da classe `Time`: criando pacotes ... 264
8.15 Acesso de pacote ... 267
8.16 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando objetos com imagens gráficas ... 269
8.17 Conclusão ... 271

9 Programação orientada aobjetos: herança 278

9.1 Introdução ... 279
9.2 Superclasses e subclasses ... 279
9.3 Membros `protected` ... 281
9.4 Relacionamento entre superclasses e subclasses ... 281
9.5 Construtores em subclasses ... 297

9.6 Engenharia de software com herança ... 297
9.7 Classe `Object` ... 298
9.8 (Opcional) Estudo de caso de GUI e imagens gráficas: exibindo texto e imagens com rótulos ... 299
9.9 Conclusão ... 301

10 Programação orientada a objetos: polimorfismo 304

10.1 Introdução ... 305
10.2 Exemplos de polimorfismo ... 306
10.3 Demonstrando um comportamento polimórfico ... 307
10.4 Classes e métodos abstratos ... 309
10.5 Estudo de caso: Sistema de folha de pagamentos utilizando polimorfismo ... 311
10.6 Métodos e classes `final` ... 321
10.7 Estudo de caso: criando e utilizando interfaces ... 321
10.8 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando com polimorfismo ... 330
10.9 Conclusão ... 331

11 Tratamento de exceções 335

11.1 Introdução ... 336
11.2 Visão geral do tratamento de erros ... 336
11.3 Exemplo de divisão por zero sem tratamento de exceções ... 337
11.4 Exemplo de tratamento de `ArithmeticExceptions` e `InputMismatchExceptions` ... 339
11.5 Quando usar tratamento de exceções ... 342
11.6 Hierarquia de exceções no Java ... 343
11.7 Bloco `finally` ... 345
11.8 Desempilhamento de pilha ... 348
11.9 `printStackTrace`, `getStackTrace` e `getMessage` ... 349
11.10 Exceções encadeadas ... 351
11.11 Declarando novos tipos de exceção ... 353
11.12 Pré-condições e pós-condições ... 354
11.13 Assertivas ... 354
11.14 Conclusão ... 355

12 Estudo de caso de ATM, Parte 1: projeto orientado a objetos com UML 360

12.1 Introdução a estudos de caso: ... 361
12.2 Examinando o documento de requisitos ... 361
12.3 Identificando classes em um documento de requisitos ... 366
12.4 Identificando atributos de classe ... 371
12.5 Identificando estados e atividades dos objetos ... 374
12.6 Identificando operações de classe ... 377
12.7 Indicando colaboração entre objetos ... 382
12.8 Conclusão ... 387

13 Estudo de caso de ATM, Parte 2: implementando um projeto orientado a objetos 390

13.1 Introdução ... 391
13.2 Começando a programar as classes do sistema ATM ... 391
13.3 Incorporando herança e polimorfismo ao sistema ATM ... 395
13.4 Implementação do estudo de caso ATM ... 399
13.5 Conclusão ... 416

14 Componentes GUI: Parte 1 418

14.1 Introdução ... 419
14.2 A nova interface Nimbus do Java ... 420
14.3 Entrada/saída baseada em GUI simples com `JOptionPane` ... 421
14.4 Visão geral de componentes Swing ... 423
14.5 Exibição de texto e imagens em uma janela ... 425
14.6 Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas ... 428
14.7 Tipos comuns de eventos GUI e interfaces ouvintes ... 433
14.8 Como o tratamento de evento funciona ... 434
14.9 `JButton` ... 435
14.10 Botões que mantêm o estado ... 438
14.11 `JComboBox` e uso de uma classe interna anônima para tratamento de eventos ... 443
14.12 `JList` ... 445
14.13 Listas de seleção múltipla ... 447
14.14 Tratamento de evento de mouse ... 449
14.15 Classes adaptadoras ... 452
14.16 Subclasse `Jpanel` para desenhar com o mouse ... 455
14.17 Tratamento de eventos chave ... 458
14.18 Introdução a gerenciadores de layout ... 460
14.19 Utilizando painéis para gerenciar layouts mais complexos ... 467
14.20 `JTextArea` ... 468
14.21 Conclusão ... 471

15 Imagens gráficas e Java 2D™ 483

15.1 Introdução ... 484
15.2 Contextos gráficos e objetos gráficos ... 484
15.3 Controle de cor ... 486
15.4 Manipulando fontes ... 491
15.5 Desenhando linhas, retângulos e ovais ... 495
15.6 Desenhando arcos ... 498
15.7 Desenhando polígonos e polilinhas ... 500
15.8 Java 2D API ... 503
15.9 Conclusão ... 508

16 Strings, caracteres e expressões regulares 515

16.1 Introdução ... 516
16.2 Fundamentos de caracteres e strings ... 516
16.3 Classe `String` ... 517
16.4 Classe `StringBuilder` ... 526
16.5 Classe `Character` ... 531
16.6 Tokenização de `Strings` ... 535
16.7 Expressões regulares, classe `Pattern` e classe `Matcher` ... 536
16.8 Conclusão ... 542

17 Arquivos, fluxos e serialização de objetos 551

17.1 Introdução ... 552
17.2 Hierarquia de dados ... 552
17.3 Arquivos e fluxos ... 554
17.4 Classe `File` ... 555
17.5 Arquivos de texto de acesso sequencial ... 558

17.6 Serialização de objeto ... 569
17.7 Classes java.io adicionais ... 576
17.8 Abrindo arquivos com JFileChooser ... 578
17.9 Conclusão ... 581

18 Recursão 589
18.1 Introdução ... 590
18.2 Conceitos de recursão ... 591
18.3 Exemplo que utiliza recursão: fatoriais ... 591
18.4 Exemplo que utiliza recursão: série de Fibonacci ... 594
18.5 Recursão e a pilha de chamadas de método ... 596
18.6 Recursão vs. Iteração ... 597
18.7 Torres de Hanói ... 598
18.8 Fractais ... 600
18.9 Retorno recursivo ... 607
18.10 Conclusão ... 608

19 Pesquisa, classificação e Big O 614
19.1 Introdução ... 615
19.2 Algoritmos de pesquisa ... 615
19.3 Algoritmos de classificação ... 622
19.4 Conclusão ... 632

20 Coleções genéricas 636
20.1 Introdução ... 637
20.2 Visão geral das coleções ... 637
20.3 Classes empacotadoras de tipo para tipos primitivos ... 638
20.4 Autoboxing e auto-unboxing ... 638
20.5 Interface Collection e classe Collections ... 639
20.6 Listas ... 639
20.7 Métodos de coleções ... 645
20.8 Classe Stack do pacote java.util ... 654
20.9 Classe PriorityQueue e interface Queue ... 655
20.10 Conjuntos ... 656
20.11 Mapas ... 658
20.12 Classe Properties ... 661
20.13 Coleções sincronizadas ... 663
20.14 Coleções não modificáveis ... 664
20.15 Implementações abstratas ... 664
20.16 Conclusão ... 665

21 Classes e métodos genéricos 670
21.1 Introdução ... 671
21.2 Motivação para métodos genéricos ... 671
21.3 Métodos genéricos: implementação e tradução em tempo de compilação ... 673
21.4 Questões adicionais da tradução em tempo de compilação: métodos que utilizam um parâmetro de tipo como tipo de retorno ... 675
21.5 Sobrecarregando métodos genéricos ... 677
21.6 Classes genéricas ... 678
21.7 Tipos brutos ... 683

21.8 Curingas em métodos que aceitam parâmetros de tipo ... 687
21.9 Genéricos e herança: notas ... 690
21.10 Conclusão ... 690

22 Estruturas de dados genéricas personalizadas 694

22.1 Introdução ... 695
22.2 Classes autorreferenciais ... 695
22.3 Alocação dinâmica de memória ... 696
22.4 Listas vinculadas ... 696
22.5 Pilhas ... 704
22.6 Filas ... 707
22.7 Árvores ... 709
22.8 Conclusão ... 714

23 Applets e Java Web Start 722

23.1 Introdução ... 723
23.2 Applets de exemplo fornecidas com o JDK ... 723
23.3 Applet Java simples: desenhando uma string ... 727
23.4 Métodos de ciclo de vida de applet ... 730
23.5 Inicializando uma variável de instância com o método `init` ... 731
23.6 Modelo de segurança da caixa de areia ... 732
23.7 Java Web Start e o Java Network Launch Protocol (JNLP) ... 733
23.8 Conclusão ... 737

24 Multimídia: applets e aplicativos 742

24.1 Introdução ... 743
24.2 Carregando, exibindo e dimensionando imagens ... 743
24.3 Animação de uma série de imagens ... 748
24.4 Mapas de imagem ... 753
24.5 Carregando e reproduzindo clipes de áudio ... 756
24.6 Reproduzindo vídeo e outros tipos de mídia com o Java Media Framework ... 758
24.7 Conclusão ... 761
24.8 Recursos da Web ... 761

25 Componentes GUI: Parte 2 768

25.1 Introdução ... 769
25.2 `JSlider` ... 769
25.3 Windows: notas adicionais ... 772
25.4 Utilizando menus com frames ... 773
25.5 `JPopupMenu` ... 779
25.6 Aparência e comportamento plugável ... 781
25.7 `JDesktopPane` e `JInternalFrame` ... 784
25.8 `JTabbedPane` ... 787
25.9 Gerenciadores de layout: `BoxLayout` e `GridBagLayout` ... 789
25.10 Conclusão ... 798

26 Multithreading 803

26.1 Introdução ... 804
26.2 Estados de thread: ciclo de vida de uma thread ... 805
26.3 Prioridades de thread e agendamento de thread ... 807

26.4 Criando e executando threads ... 808
26.5 Sincronização de thread ... 812
26.6 Relacionamento entre produtor e consumidor sem sincronização ... 818
26.7 Relacionamento de produtor/consumidor: `ArrayBlockingQueue` ... 823
26.8 Relacionamento entre produtor e consumidor com sincronização ... 825
26.9 Relacionamento de produtor/consumidor: buffers limitados ... 829
26.10 Relacionamento de produtor/consumidor: as interfaces `Lock` e `Condition` ... 835
26.11 Multithreading com GUI ... 840
26.12 Interfaces `Callable` e `Future` ... 851
26.13 Conclusão ... 851

27 Redes 858

27.1 Introdução ... 859
27.2 Manipulando URLs ... 860
27.3 Lendo um arquivo em um servidor da Web ... 863
27.4 Estabelecendo um servidor simples utilizando sockets de fluxo ... 866
27.5 Estabelecendo um cliente simples utilizando sockets de fluxo ... 867
27.6 Interação cliente/servidor com conexões de socket de fluxo ... 868
27.7 Interação cliente/servidor sem conexão com datagramas ... 877
27.8 Jogo da velha cliente/servidor que utiliza um servidor com multithread ... 882
27.9 [Bônus Web] Estudo de caso: servidor e cliente `DeitelMessenger` ... 893
27.10 Conclusão ... 893

28 Acesso a bancos de dados com o JDBC 898

28.1 Introdução ... 899
28.2 Bancos de dados relacionais ... 900
28.3 Visão geral de um banco de dados relacional: o banco de dados `books` ... 900
28.4 SQL ... 903
28.5 Instruções para instalar o MySQL e o MySQL Conector/J ... 910
28.6 Instruções para configuração de uma conta de usuário MySQL ... 910
28.7 Criando banco de dados `books` no MySQL ... 911
28.8 Manipulando bancos de dados com o JDBC ... 911
28.9 Interface `RowSet` ... 924
28.10 Java DB/Apache Derby ... 926
28.11 `PreparedStatements` ... 927
28.12 Procedures armazenadas ... 938
28.13 Processamento de transações ... 939
28.14 Conclusão ... 939
28.15 Recursos da Web ... 939

29 Aplicativos Web JavaServer™ Faces 946

29.1 Introdução ... 947
29.2 Transações HTTP simples ... 947
29.3 Arquitetura de aplicativo multithread ... 950
29.4 Tecnologias Web Java ... 950
29.5 Criando e executando um aplicativo simples no NetBeans ... 952
29.6 Componentes JSF ... 961
29.7 Monitoramento de sessão ... 970
29.8 Conclusão ... 986

30 Aplicativos Web JavaServer™ Faces compatíveis com Ajax 994

30.1 Introdução ... 995
30.2 Acessando bancos de dados em aplicativos Web ... 995
30.3 Componentes JSF compatíveis com o Ajax ... 1005
30.4 Criando um **Text Field** de autocompletamento e utilizando formulários virtuais ... 1006
30.5 Conclusão ... 1013

31 Serviços Web 1017

31.1 Introdução ... 1018
31.2 Fundamentos do serviço Web ... 1019
31.3 Simple Object Access Protocol (SOAP) ... 1019
31.4 Representational State Transfer (REST) ... 1020
31.5 JavaScript Object Notation (JSON) ... 1020
31.6 Publicando e consumindo serviços Web baseados em SOAP ... 1020
31.7 Publicando e consumindo serviços Web XML baseados em REST ... 1028
31.8 Publicando e consumindo serviços Web JSON baseados em REST ... 1032
31.9 Rastreamento de sessão em um serviço Web baseado em SOAP ... 1035
31.10 Consumindo um serviço Web baseado em SOAP orientado a banco de dados ... 1045
31.11 Gerador de equação: retornando tipos definidos pelo usuário ... 1051
31.12 Conclusão ... 1059

A Tabela de precedência de operadores 1066

B Conjunto de caracteres ASCII 1068

C Palavras-chave e palavras reservadas 1069

D Tipos primitivos 1070

E Usando a documentação da Java API 1071

E.1 Introdução ... 1071
E.2 Navegando pela Java API ... 1071

F Utilizando o depurador 1077

F.1 Introdução ... 1078
F.2 Pontos de interrupção e os comandos `run`, `stop`, `cont` e `print` ... 1078
F.3 Os comandos `print` e `set` ... 1081
F.4 Controlando a execução utilizando os comandos `step`, `step up` e `next` ... 1082
F.5 O comando `watch` ... 1085
F.6 O comando `clear` ... 1086
F.7 Conclusão ... 1088

G Saída formatada 1090

G.1 Introdução ... 1091
G.2 Fluxos ... 1091
G.3 Formatando a saída com `printf` ... 1091
G.4 Imprimindo inteiros ... 1092

G.5 Imprimindo números de ponto flutuante ... 1092
G.6 Imprimindo strings e caracteres ... 1094
G.7 Imprimindo datas e horas ... 1094
G.8 Outros caracteres de conversão ... 1096
G.9 Imprimindo com larguras e precisões de campos ... 1098
G.10 Utilizando flags na string de formato `printf` ... 1099
G.11 Imprimindo com índices de argumento ... 1102
G.12 Imprimindo literais e sequências de escape ... 1103
G.13 Formatando saída com a classe `Formatter` ... 1103
G.14 Conclusão ... 1104

Apêndices na Web **1109**

H Sistemas de numeração

I GroupLayout

J Java Desktop Integration Components (JDIC)

K Mashups

L Unicode®

M Criando documentaçãocom o javadoc

N Manipulação de bits

O Instruções rotuladas `break` e `continue`

P UML 2: Tipos de diagramas adicionais

Q Padrões de design

Índice remissivo **1110**

Prefácio

Nunca mais viver em fragmentos, apenas se conectar.
— Edgar Morgan Foster

Bem-vindo ao Java e ao *Java — Como programar*, oitava edição! Este livro apresenta tecnologias de computação de ponta para estudantes, instrutores, desenvolvedores de software e profissionais de TI.

Utilizamos a característica "abordagem live-code" de nossos livros, apresentando a maioria dos conceitos no contexto de programas Java totalmente funcionais, em vez do uso de trechos de código. Cada exemplo de código é imediatamente seguido por uma ou mais execuções de exemplo. Todo o código-fonte está disponível em `www.deitel.com/books/jhtp8/` e `www.prenhall.com/deitel_br`.

Na Deitel & Associates, escrevemos livros universitários e livros profissionais sobre linguagens de programação para a Pearson/Prentice Hall, ministramos cursos de treinamento corporativo no mundo todo e desenvolvemos negócios na Internet baseados na Web 2.0. Atualizamos a edição anterior deste livro com base nas modificações recentes na linguagem Java e na evolução das maneiras preferidas de ensino e aprendizagem de programação. Todos os capítulos foram significativamente aprimorados.

Recursos novos e atualizados

Aqui estão as atualizações que fizemos neste livro:

- O livro tem um novo design interno que organiza, esclarece e destaca graficamente as informações e aprimora sua pedagogia.
- Atualizamos o livro inteiro de acordo com a Java Standard Edition 6 Update 11 e auditamos e comparamos cuidadosamente o manuscrito com a *Java Language Specification*.
- Adicionamos o conjunto de exercícios "Fazendo a diferença": os estudantes *querem* fazer a diferença. Nós os encorajamos a associar os computadores e a Internet à solução de problemas que realmente importam para pessoas, comunidades, países e para o mundo. Esperamos que os novos exercícios estimulem os estudantes a pensar por conta própria à medida que eles exploram questões sociais complexas. Esses exercícios não visam a fazer uma declaração política. Eles foram concebidos para aumentar a conscientização em relação a questões importantes que o mundo enfrenta. Os estudantes devem abordar essas questões no contexto dos seus próprios valores, política e crenças. Muitos dos novos exercícios exigem que os estudantes façam pesquisas na Web — e transformem os resultados em um processo de solução de problemas próprio. Eis uma lista dos 33 novos exercícios de "Fazendo a diferença":
  - ✓ Test drive: Calculadora de emissão de carbono
  - ✓ Test drive: Calculadora de índice de massa corpórea
  - ✓ Atributos dos veículos híbridos
  - ✓ Neutralidade de sexos
  - ✓ Calculadora de índice de massa corpórea
  - ✓ Crescimento demográfico mundial
  - ✓ Perguntas sobre fatos relacionados ao aquecimento global
  - ✓ Alternativas de planejamento tributário; o "imposto justo"
  - ✓ Instrução auxiliada por computador
  - ✓ Instrução auxiliada por computador: Redução da fadiga do aluno
  - ✓ Instrução auxiliada por computador: Monitoramento do desempenho estudantil
  - ✓ Instrução auxiliada por computador: Níveis de dificuldade
  - ✓ Instrução auxiliada por computador: Variação dos tipos de problemas

✓ Enquetes
✓ Controle de tráfego aéreo
✓ Interface de emissão de carbono: Polimorfismo
✓ Calculadora do crescimento demográfico mundial
✓ Calculadora da economia do transporte solidário
✓ Calculadora da taxa de frequência cardíaca
✓ Computadorização dos registros de saúde
✓ Impondo privacidade com criptografia
✓ Ecofont
✓ Professor de digitação: Aprimorando uma habilidade crucial na era da informática
✓ Telas com fontes grandes para pessoas com problemas de visão
✓ Cozinhando com ingredientes mais saudáveis
✓ Scanner de spam
✓ Scanner de phishing
✓ Projeto de acessibilidade: Síntese da fala
✓ Projeto de acessibilidade: Reconhecimento da fala
✓ Projeto: Simulador robótico Simbad
✓ Serviço Web de scanner de spam
✓ Serviço Web de SMS
✓ Serviço Web de neutralidade de sexos

- Ajustamos o estudo de caso opcional do projeto orientado a objetos/UML 2 do caixa eletrônico (ATM) e o reorganizamos em dois capítulos opcionais (12 e 13) que apresentam o projeto do caixa eletrônico e a implementação completa do código. O ATM é um bom exemplo de negócio com o qual os alunos podem se relacionar. Na nossa experiência, ensinar esses dois capítulos como uma unidade ajuda os estudantes a associar muitos dos conceitos orientados a objetos que eles aprendem nos capítulos 1–10. Um conceito-chave na programação orientada a objetos é a interação entre objetos. Na maioria dos livros escolares de programação, os exemplos de código criam e utilizam um ou dois objetos. O ATM dá aos estudantes a oportunidade de estudar interações de *muitos* objetos que fornecem a funcionalidade de um sistema substancial. Os capítulos 12 e 13 fornecem soluções completas para *todos* os exercícios relacionados. Na edição anterior, o estudo de caso foi distribuído ao longo dos capítulos 2–8, 10 e um apêndice. Para professores que desejam abordar o estudo de caso de uma maneira distribuída, para cada seção nos capítulos 12 e 13, indicamos depois de qual capítulo anterior a seção pode ser abordada.
- Reforçamos nossa pedagogia de introdução antecipada de classes e objetos dando bastante atenção à orientação dos professores universitários nas nossas equipes de revisão para assegurar que alcançamos o nível conceitual certo. O tratamento da POO (Programação Orientada a Objetos) é claro e acessível. Introduzimos os conceitos básicos e a terminologia da tecnologia de objetos no Capítulo 1. Os estudantes desenvolvem suas primeiras classes e objetos personalizados no Capítulo 3. Apresentar objetos e classes, logo no início, faz com que os estudantes "pensem sobre objetos" imediatamente e dominem esses conceitos mais a fundo.
- Reordenamos nossa apresentação das estruturas de dados. Agora começamos com a classe genérica `ArrayList` no Capítulo 7. Como os estudantes entenderão os conceitos genéricos básicos bem no início do livro, nossas discussões posteriores sobre estruturas de dados fornecem um tratamento mais profundo das coleções genéricas — mostrando como utilizar as coleções internas da API do Java. Mostramos, então, como implementar métodos e classes genéricos. Por fim, mostramos como construir estruturas de dados genéricas personalizadas.
- Adicionamos a cobertura do Java Web Start e Java Network Launch Protocol (JNLP), que permitem que applets *e* aplicativos sejam carregados por um navegador Web. Além disso, o usuário pode instalá-los como atalhos na área de trabalho para executá-los, no futuro, sem revisitar o site Web. Os programas também podem solicitar a permissão do usuário para acessar recursos do sistema local, tais como arquivos — permitindo que você desenvolva applets e aplicativos mais robustos, executados de uma maneira segura utilizando o modelo de segurança de caixa de areia do Java, que se aplica ao código baixado da Internet.
- Reordenamos vários capítulos para facilitar o ensino do livro em módulos. O fluxograma de dependências (página xxiii) foi atualizado para refletir a nova modularização.
- Adicionamos muitos links à documentação on-line, para que os estudantes possam aprender mais sobre uma classe ou um tópico, e adicionamos muitos links para os Resource Centers relacionados ao Java da Deitel, disponíveis em `www.deitel.com/ResourceCenters.html`.
- O Capítulo 7 agora abrange a classe `Arrays` — que contém métodos para realizar manipulações de array comuns — e a classe `ArrayList` — que implementa uma estrutura de dados parecida a um array dinamicamente redimensionável. Isso segue a nossa filosofia *de utilizar classes existentes antes de aprender como definir suas próprias classes*.
- Agora, introduzimos a classe `BigInteger` para valores inteiros, arbitrariamente grandes, no Capítulo 18, "Recursão".

- Ajustamos cuidadosamente todos os capítulos com ênfase maior na clareza e simplicidade, eliminando redundâncias, reduzindo o número de páginas (esta nova edição tem 90 páginas a menos do que a anterior) e aprimorando a pedagogia e a organização modular.
- Substituímos todos os usos da `StringTokenizer` pelo método recomendável `String split` por todo o livro. A classe `StringTokenizer` ainda é discutida, principalmente para compatibilidade com versões anteriores com código legado.
- Incluímos uma lista em ordem alfabética dos termos importantes definidos em cada capítulo com o número de página da ocorrência que define o termo. Ocorrências de definições também são destacadas no índice com um número de página em **vermelho escuro**.

Tudo isso foi cuidadosamente revisado por 24 eminentes acadêmicos e desenvolvedores do setor que trabalharam conosco neste livro.

Acreditamos que este livro e seu material de suporte propiciarão a estudantes e profissionais uma experiência educacional em Java informativa, interessante, desafiadora e divertida. Fornecemos um conjunto de materiais subsidiário que ajudará os professores a maximizar a experiência de aprendizagem dos seus alunos.

Ao ler o livro, se tiver dúvidas, envie um e-mail para `deitel@deitel.com`; responderemos prontamente. Para atualizações deste livro e o status de todo o software Java de suporte, e para as notícias mais recentes sobre todas as publicações e serviços Deitel, visite `www.deitel.com`. Registre-se em `www.deitel.com/newsletter/subscribe.html` para receber por e-mail o boletim informativo gratuito *Deitel® Buzz On-line* (em inglês) e conheça nossa crescente lista sobre Java e recursos relacionados do Resource Centers em `www.deitel.com/ResourceCenters.html`. Toda semana anunciamos nossos mais recentes recursos no boletim informativo.

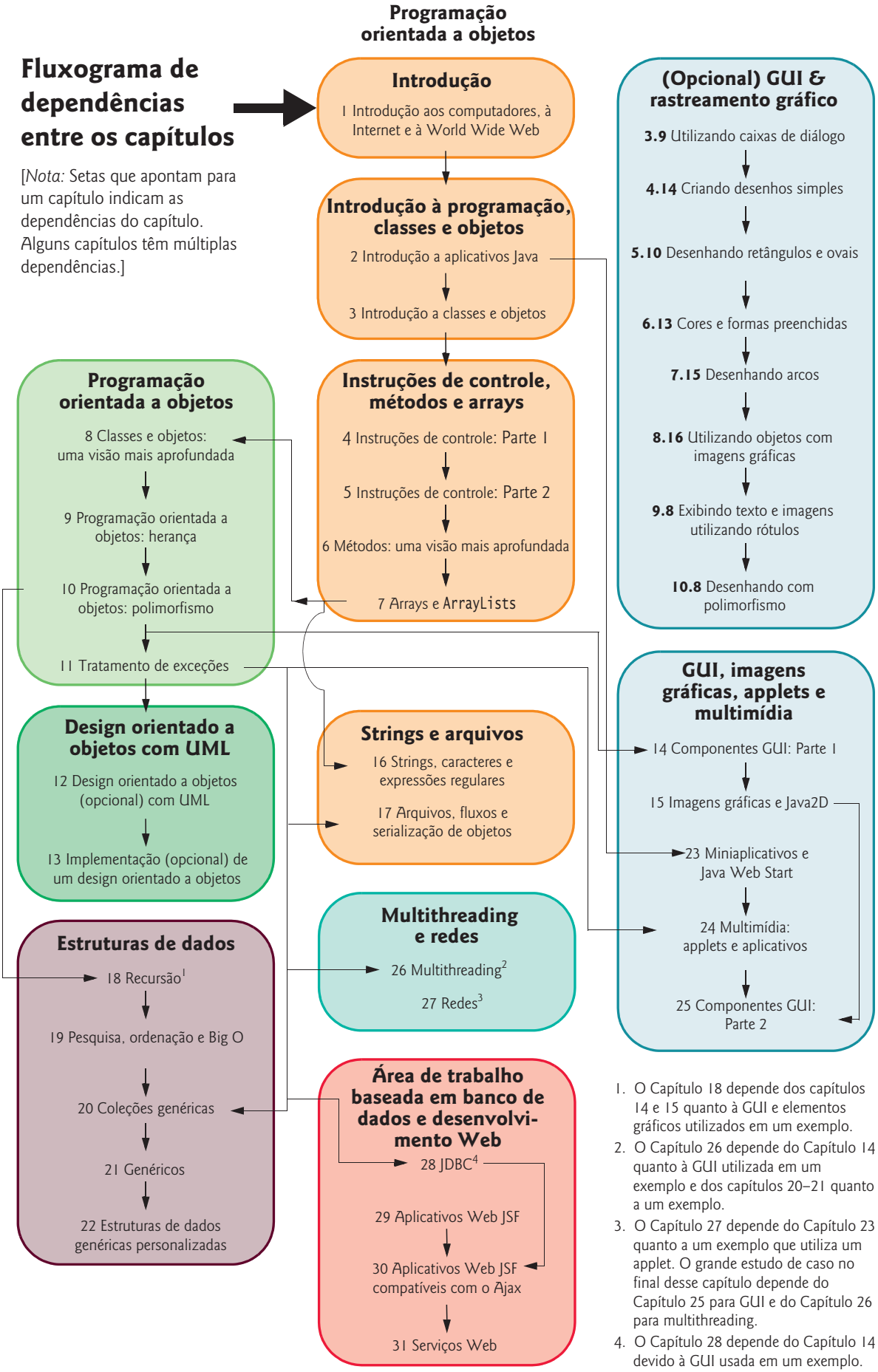
Outros recursos

Outros recursos desta oitava edição:

- Alinhamos a apresentação com as recomendações curriculares da ACM/IEEE e do Computer Science Advanced Placement Examination.
- A apresentação antecipada de classes e objetos fornece estudos de caso sobre as classes `Time`, `Employee` e `GradeBook` discutidas em várias seções e capítulos, introduzindo gradualmente conceitos OO mais profundos.
- Professores que ministram cursos introdutórios têm uma ampla escolha quanto ao volume de material sobre GUIs e elementos fluxogramas a abranger — desde nenhuma sequência introdutória, ou dez breves seções, até um tratamento detalhado nos capítulos 14, 15 e 25 e no Apêndice I.
- Nossa programação orientada a objetos e as apresentações de design utilizam a *UML™* (*Unified Modeling Language™*) — a linguagem gráfica padrão do setor para modelagem de sistemas orientados a objetos.
- Fornecemos vários estudos de caso substanciais de programação Web orientada a objetos.
- O Capítulo 28, "Acesso a bancos de dados com o JDBC", abrange o JDBC 4 e utiliza os sistemas de gerenciamento de bancos de dados Java DB/Apache Derby e MySQL. O capítulo apresenta um estudo de caso OO sobre o desenvolvimento de um catálogo de endereços voltado a banco de dados que demonstra instruções preparadas e a descoberta automática de drivers do JDBC 4.
- O Capítulo 29, "Aplicativos Web JavaServer™ Faces" e o Capítulo 30, "Aplicativos Web JavaServer™ Faces compatíveis com Ajax", introduzem a tecnologia JavaServer Faces (JSF) e utiliza-a com o NetBeans 6.5 para construir aplicativos Web rápida e facilmente. O Capítulo 29 inclui exemplos da criação de GUIs de aplicativos Web, tratamento de eventos, formulários de validação e rastreamento de sessão. O Capítulo 30 discute o desenvolvimento de aplicativos Web compatíveis com o Ajax, utilizando a tecnologia JavaServer Faces. Ele apresenta um aplicativo Web multicamada do tipo catálogo de endereços baseado em banco de dados que permite aos usuários adicionar e pesquisar contatos. Esse aplicativo compatível com o Ajax dá ao leitor uma boa ideia do desenvolvimento de softwares baseado na Web 2.0. O aplicativo utiliza componentes JSF compatíveis com o Ajax para sugerir nomes de contato enquanto o usuário digita um nome a localizar.
- O Capítulo 31, "Serviços Web", utiliza uma abordagem baseada em ferramentas para criar e consumir serviços Web baseados em SOAP *e* REST. Os estudos de caso incluem o desenvolvimento de serviços Web de reservas aéreas e do jogo Vinte-e-um.
- Utilizamos uma nova abordagem baseada em ferramentas para desenvolver rapidamente aplicativos Web; todas as ferramentas estão disponíveis gratuitamente para download.
- Fornecemos o 100+ Resource Centers (`www.deitel.com/resourcecenters.html`) para dar suporte aos nossos leitores acadêmicos e profissionais. Seus tópicos incluem Java SE 6, Java, avaliação e certificação Java, padrões de design Java, Java EE 5, sistemas de pesquisa de código e sites de código, programação de jogos, projetos de programação e muito mais. Para receber por e-mail o boletim informativo gratuito *Deitel® Buzz On-line,* registre-se em `www.deitel.com/newsletter/subscribe.html` — toda semana anunciamos nossos Resource Centers mais recentes e incluímos outros itens de interesse para nossos leitores.
- Discutimos conceitos-chave da comunidade de engenharia de software, como Web 2.0, Ajax, SaaS (Software as a Service), serviços Web, software de código-fonte aberto, padrões de design, mashups, refatoração, desenvolvimento ágil de softwares, prototipagem rápida etc.
- Reformulamos completamente o Capítulo 26, "Multithreading" [agradecemos especialmente a Brian Goetz e Joseph Bowbeer — coautores do *Java Concurrency in Practice,* Addison-Wesley, 2006].

Fluxograma de dependências entre os capítulos

[*Nota:* Setas que apontam para um capítulo indicam as dependências do capítulo. Alguns capítulos têm múltiplas dependências.]



1. O Capítulo 18 depende dos capítulos 14 e 15 quanto à GUI e elementos gráficos utilizados em um exemplo.
2. O Capítulo 26 depende do Capítulo 14 quanto à GUI utilizada em um exemplo e dos capítulos 20–21 quanto a um exemplo.
3. O Capítulo 27 depende do Capítulo 23 quanto a um exemplo que utiliza um applet. O grande estudo de caso no final desse capítulo depende do Capítulo 25 para GUI e do Capítulo 26 para multithreading.
4. O Capítulo 28 depende do Capítulo 14 devido à GUI usada em um exemplo.

- Discutimos a classe `SwingWorker` para desenvolver interfaces com o usuário baseadas em multithreading.
- Discutimos o gerenciador de layout `GroupLayout` no contexto da ferramenta de design de GUI no IDE do NetBeans.
- Apresentamos capacidades de ordenação e filtragem do `JTable` que permitem ao usuário reordenar os dados.
- Discutimos a classe `StringBuilder`, que tem um desempenho melhor do que `StringBuffer` em aplicativos sem threading.
- Apresentamos anotações, que reduzem significativamente a quantidade de código que você tem de escrever para construir aplicativos.

Estudo de caso opcional Utilizando UML 2 para desenvolver um design de ATM orientado a objetos

A UML 2 tornou-se a linguagem de modelagem gráfica preferida para projetar sistemas orientados a objetos. Utilizamos diagramas de atividade UML (em vez de fluxogramas) para demonstrar o fluxo de controle em cada uma das instruções de controle Java e utilizamos diagramas de classe UML para representar visualmente classes e seus relacionamentos de herança.

Incluímos um estudo de caso opcional (mas altamente recomendável) sobre o projeto orientado a objetos utilizando a UML. O estudo de caso foi revisado por meio de várias edições, por profissionais do setor e uma equipe eminente de acadêmicos OOD/UML, incluindo líderes no segmento da Rational (os criadores da UML) e do Object Management Group (responsável pela evolução da UML). No estudo de caso, projetamos e implementamos integralmente o software para um sistema de caixa eletrônico (Automatic Teller Machine — ATM) simples. O estudo de caso opcional de engenharia de softwares nos capítulos 12 e 13 apresenta uma introdução cuidadosamente compassada ao projeto orientado a objetos utilizando a UML.

Introduzimos um subconjunto simples e conciso da UML 2 e então orientamos o leitor para uma primeira experiência de design destinada a principiantes. O estudo de caso não é um exercício; em vez disso, é uma experiência de aprendizagem de ponta a ponta que termina com uma revisão detalhada do código Java completo. Os capítulos 12 e 13 ajudam os alunos a desenvolver um projeto orientado a objetos para complementar os conceitos da programação orientada a objetos que eles aprenderam nos capítulos de 1–11.

No final do Capítulo 1, introduzimos conceitos básicos e terminologia relacionada ao OOD. No Capítulo 12, consideramos questões mais substanciais, à medida que empreendemos um problema desafiador com as técnicas OOD. Analisamos um documento de requisitos típicos que especifica um sistema a ser construído, determinamos os objetos necessários para implementar esse sistema, determinamos os atributos que esses objetos precisam ter, determinamos os comportamentos que esses objetos precisam exibir e especificamos como os objetos devem interagir um com o outro para atender aos requisitos do sistema. No Capítulo 13, incluímos uma implementação de código Java completa do sistema orientado a objetos que projetamos no Capítulo 12.

Esse estudo de caso ajuda a preparar os alunos para os tipos de projetos substanciais que eles encontrarão na indústria. Empregamos um processo de projeto orientado a objetos incremental cuidadosamente desenvolvido para produzir um modelo de UML 2 para nosso sistema ATM. A partir desse projeto, criamos uma rica implementação Java funcional utilizando noções-chave da programação orientada a objeto, incluindo classes, objetos, encapsulamento, visibilidade, composição, herança e polimorfismo.

Fluxograma de dependências

O fluxograma da página anterior mostra as dependências entre os capítulos para ajudar os professores a planejar seus planos de estudos. Este livro é apropriado para vários cursos de programação em vários níveis, mais notavelmente cursos e sequências de cursos de Ciência da Computação 1 e 2 nas disciplinas relacionadas.

O livro tem uma organização claramente delineada e modular. Os capítulos 1–11 e 14–17 formam uma sequência de programação elementar acessível com uma sólida introdução à programação orientada a objetos. Os capítulos 12–13 opcionais formam uma introdução acessível ao projeto orientado a objetos com a UML. Os tópicos sobre GUI e rastreamento gráfico junto com os capítulos 14, 15, 23, 24 e 25 formam uma sequência substancial sobre GUI, imagens gráficas e multimídia. Os capítulos 18–22 formam uma boa sequência de estruturas de dados. Os capítulos 26–27 formam uma sólida introdução a multithreading e redes Internet. Os capítulos 28–31 formam uma sequência clara do desenvolvimento Web que usa intensamente banco de dados.

Ajuda para planos de estudo

Teremos toda satisfação em ajudar os professores a criar planos de estudos baseados neste livro. Você pode entrar em contato conosco por e-mail (`clientes@pearsoned.com`).

A abordagem de ensino

Este livro contém uma rica coleção de exemplos. O livro concentra-se nos princípios da boa engenharia de software e destaca principalmente a clareza da programação. Ensinamos por meio de exemplos. Somos educadores que ensinam linguagens de programação de ponta e tópicos relacionados a softwares nos setores governamental, industrial, militar e acadêmico no mundo todo.

Abordagem Live-Code. Este livro está repleto de exemplos de códigos funcionais. Isso significa que cada novo conceito é apresentado no contexto de um aplicativo Java totalmente funcional, seguido imediatamente por uma ou mais execuções reais que mostram as entradas e saídas do programa.

Sintaxe colorida. Por questões de legibilidade, utilizamos uma sintaxe colorida para todos os códigos Java, semelhante à maneira da maioria dos ambientes de desenvolvimento integrado Java e editores de código, que utilizam cores nos códigos. Nossas convenções para cores de sintaxe incluem:

```
Comentários aparecem em verde
Palavras-chave aparecem em azul-escuro
Erros aparecem em vermelho
Constantes e valores literais aparecem em azul claro
Todos os outros códigos aparecem em preto
```

Destaque de código. Realçamos em amarelo os segmentos de código mais importantes.

Uso de fontes para ênfase. Inserimos os termos-chave e a referência de página do índice para cada ocorrência definidora em texto em negrito em **vermelho escuro** para facilitar a referência. Enfatizamos os componentes na tela com a fonte **Helvetica** em negrito (por exemplo, o menu **File**) e enfatizamos o texto do programa Java na fonte `Lucida` (por exemplo, `int x = 5;`).

Acesso Web. Todos os exemplos de código-fonte deste livro estão disponíveis para download em:

```
www.pearson.com.br/deitel
```

Objetivos. Cada capítulo inicia com uma apresentação de objetivos.

Citações. Os objetivos da aprendizagem são seguidos por citações. Esperamos que você aprecie relacionar esses exemplos ao material do capítulo.

Ilustrações/figuras. Inúmeros gráficos, tabelas, desenhos a traço, programas e saída de programa foram incluídos. Modelamos o fluxo de controle em instruções de controle com diagramas de atividade UML. Os diagramas de classes UML modelam os campos, construtores e métodos das classes. Fazemos uso extenso de seis tipos de diagrama UML principais no estudo de caso opcional do ATM, baseado no OOD/UML 2.

Dicas de programação. Incluímos dicas de programação para ajudá-lo a focalizar aspectos importantes do desenvolvimento do programa. Essas dicas e práticas representam o melhor que reunimos a partir de sete décadas combinadas de programação e experiência pedagógica.

Boa prática de programação

As Boas práticas de programação *chamam atenção a técnicas que irão ajudá-lo a criar programas que são mais claros, mais compreensíveis e mais fáceis de manter.*

Erro de programação comum

Indicar esses erros de programação comuns reduz a probabilidade de que eles aconteçam.

Dica de prevenção de erro

Essas dicas contêm sugestões para expor bugs e removê-los dos seus programas; muitos descrevem aspectos do Java que evitam que bugs apareçam nos programas.

Dica de desempenho

Essas dicas destacam oportunidades para fazer seus programas executarem mais rapidamente ou minimizar a quantidade de memória que eles ocupam.

Dica de portabilidade

As dicas de portabilidade ajudam a escrever código que poderá ser executado em diferentes plataformas.

Observação de engenharia de software

As Observações de engenharia de software *destacam questões arquitetônicas e de projeto que afetam a construção de sistemas de software, especialmente sistemas de larga escala.*

Observação sobre a aparência e funcionamento

Observações sobre a aparência e funcionamento *destacam as convenções da interface gráfica com o usuário. Essas observações ajudam a criar interfaces gráficas atraentes e amigáveis ao usuário que seguem as normas da indústria.*

Seção de resumo. Cada capítulo termina com uma seção de resumo que recapitula seu conteúdo e as transições para o próximo capítulo.

Lista de tópicos de resumo. Cada capítulo termina com dispositivos pedagógicos adicionais. Apresentamos um resumo do capítulo, sessão por sessão, no estilo de lista itemizada.

Terminologia. Incluímos uma lista em ordem alfabética dos termos importantes definidos em cada capítulo com o número de página da ocorrência que define o termo. Ocorrências de definições também são destacadas no índice com um número de página em **vermelho escuro**.

Exercícios e respostas de autorrevisão. Extensos exercícios e respostas de autorrevisão são incluídos para autoaprendizagem. Todos os exercícios no estudo de caso opcional do ATM estão totalmente resolvidos.

Exercícios. Cada capítulo termina com um conjunto substancial de exercícios, incluindo recapitular resumidamente a terminologia e conceitos importantes; identificar erros nos exemplos de código; escrever instruções de programa individuais; escrever pequenas partes dos métodos e classes Java; escrever métodos e classes e programas Java completos; e criar projetos de conclusão de curso. Os instrutores podem utilizar esses exercícios para formar deveres de casa, pequenos questionários, exames importantes e projetos de conclusão. [NOTA: **Não nos escreva solicitando acesso ao Resource Center da Pearson Instructor. O acesso é limitado estritamente a professores universitários que utilizam o livro em suas aulas. Professores só podem obter acesso por meio dos seus representantes na Pearson.**] Certifique-se de verificar em nosso Programming Projects Resource Center (`http://www.deitel.com/ProgrammingProjects/`) vários exercícios adicionais e possibilidades de projeto.

Milhares de entradas de índice. Incluímos um índice extenso, que é especialmente útil ao utilizar o livro como uma referência.

Recursos para o aluno incluídos neste livro

Há muitas ferramentas de desenvolvimento Java disponíveis para compra, mas você não precisa de nenhuma delas para começar a trabalhar com o Java. Para sistemas Windows, todos os softwares que você precisará para este livro estão disponíveis gratuitamente para o download na Web ou no CD que o acompanha. Para outras plataformas, todos os softwares que você precisará para este livro estão, na sua maioria, disponíveis gratuitamente para download a partir da Web. Escrevemos a maioria dos exemplos neste livro utilizando o Java Standard Edition Development Kit (JDK) 6. A versão atual do JDK (e separadamente sua documentação) pode ser baixada do site Web Java da Sun, `java.sun.com/javase/downloads/index.jsp`. Os usuários do Mac OS X podem fazer o download do Java em `developer.apple.com/java`. Em vários capítulos, também utilizamos o IDE do NetBeans. O NetBeans está disponível como um pacote com o JDK do site Web Java da Sun precedente, ou você pode fazer o download dele separadamente em `www.netbeans.org/downloads/index.html`. O Eclipse pode ser baixado de `www.eclipse.org/downloads/`. O MySQL pode ser baixado de `http://dev.mysql.com/downloads/`, e MySQL Connector/J em `http://dev.mysql.com/downloads/connector/j/5.1.html`. Você pode encontrar recursos adicionais e downloads de softwares no nosso Resource Center do Java SE 6 em:

```
www.deitel.com/JavaSE6Mustang/
```

O CD que acompanha este livro contém versões dos seguintes pacotes de software para uso no Microsoft® Windows®:

- O Java™ SE Development Kit (JDK) 6 Update 11 — que foi utilizado para criar e testar todos os programas no livro.
- O IDE do Eclipse para desenvolvedores Java EE versão 3.4.1.
- NetBeans™ IDE Version 6.5 All Bundle.
- MySQL® 5.0 Community Server, versão 5.0.67.
- MySQL® Connector/J, versão 5.1.7.

O NetBeans e o Eclipse são ambientes de desenvolvimento integrado (IDEs) para desenvolver todos os tipos de aplicativos Java. O MySQL e o Conector/J MySQL são fornecidos para os aplicativos de banco de dados nos capítulos 28–31. Todas essas ferramentas também podem ser baixadas para outras plataformas, como discutido em "Antes de você começar", depois deste Prefácio.

O CD também contém uma página Web com links para o site Web da Deitel & Associates, Inc. e para o site Web da Pearson. Essa página da Web pode ser carregada em um navegador da Web para acesso rápido a todos os recursos.

Companion Website

O site de apoio desta oitava edição brasileira (`www.pearson.com.br/deitel`) oferece:

- Para estudantes
  - ✓ exercícios de múltipla escolha
  - ✓ código-fonte dos exemplos apresentados no livro (em inglês)
- Para professores[1]
  - ✓ manual de soluções (em inglês)
  - ✓ apresentações em PowerPoint

Cursos avançados de ciência da computação

Este livro é adequado para ensino avançado de ciência da computação e também para preparar os alunos para realizar os exames avançados.

Boletim *Deitel® Buzz On-line* gratuito via e-mail

Toda semana, o *Deitel® Buzz On-line* anuncia o(s) nosso(s) Resource Center(s) mais recente(s) e inclui comentários sobre tendências e desenvolvimentos do setor, links para artigos gratuitos e recursos oferecidos por nossos livros publicados e futuras publicações, agendas de lançamento de produtos, erratas, desafios, depoimentos de leitores e usuários, informações sobre nossos cursos corporativos voltados a professores e muitos mais. Também é um bom modo de manter-se atualizado sobre este livro. Para cadastrar-se, visite:

```
www.deitel.com/newsletter/subscribe.html
```

Deitel on-line Resource Centers

O site Web `www.deitel.com` fornece mais de 100 Resource Centers (em inglês) sobre vários temas, incluindo linguagens de programação, desenvolvimento de software, Web 2.0, negócios on-line e projetos de código-fonte aberto — veja a lista dos Resource Centers em `www.deitel.com/ResourceCenters.html`. Os Resource Centers evoluíram a partir de pesquisas que fazemos para dar suporte aos nossos livros e empreendimentos comerciais. Encontramos muitos recursos excepcionais on-line, incluindo tutoriais, documentação, downloads de software, artigos, blogs, podcasts, vídeos, exemplos de código, livros, e-books e mais — a maioria deles é gratuita. Toda semana anunciamos nossos Resource Centers no nosso boletim informativo, o *Deitel® Buzz On-line* (`www.deitel.com/newsletter/subscribe.html`). Alguns dos Resource Centers que você pode achar útil ao estudar este livro são Java SE 6, Java, avaliação e certificação em Java, padrões de design Java, Java EE 5, sistemas de pesquisa de código e sites de código, de jogos, programação de jogos, programação em geral e muitos outros.

Agradecimentos

É um prazer reconhecer os esforços das pessoas cujos nomes não aparecem na capa, mas cujo trabalho árduo, cooperação, amizade e compreensão foram cruciais para a produção do livro. Muitas pessoas na Deitel & Associates, Inc., dedicaram longas horas a este projeto — agradecemos especialmente a Abbey Deitel e Barbara Deitel.

Também gostaríamos de agradecer aos participantes do nosso Honors Internship Program que contribuíram para esta nova edição — Nicholas Doiron, bacharel em engenharia elétrica e de computação pela Carnegie Mellon University; e Matthew Pearson, bacharel em ciência da computação pela Cornell University.

Tivemos a felicidade de trabalhar neste projeto com uma equipe talentosa e dedicada de profissionais de publicação da Pearson. Apreciamos os esforços extraordinários de Marcia Horton, diretora editorial de engenharia e ciência da computação da Pearson. Carole Snyder e Dolores Mars fizeram um trabalho extraordinário na contratação da equipe de revisão do livro e gerenciamento do processo de revisão. Francesco Santalucia (um artista independente) e Kristine Carney, da Pearson, fizeram um trabalho maravilhoso na criação da capa do livro — fornecemos o conceito e eles o tornaram uma realidade. Scott Disanno e Robert Engelhardt fizeram um maravilhoso trabalho no gerenciamento da produção do livro. Erin Davis, nosso gerente de marketing, e Margaret Waples, sua chefe, fizeram um excelente trabalho para o marketing do livro em canais acadêmicos e profissionais.

Revisores desta edição

Queremos reconhecer os esforços dos nossos revisores. Seguindo uma agenda apertada, eles escrutinaram o texto e os programas, e forneceram inúmeras sugestões para aprimorar a exatidão e a completa abrangência da apresentação:

[1] Este material é de uso exclusivo de professores e protegido por senha. Para adquirir sua senha, contate seu representante ou envie um e-mail para o endereço `universitarios@pearson.com`.

Revisores da Microsystems

- Lance Andersen
- Soundararajan Angusamy
- Lawrence Prem Kumar
- Simon Ritter
- Sang Shin
- Alexander Zuev

Revisores acadêmicos

- William E. Duncan (Louisiana State University)
- Diana Franklin (University of California, Santa Barbara)
- Edward F. Gehringer (North Carolina State University)
- Ric Heishman (George Mason University)
- Patty Kraft (San Diego State University)
- Manjeet Rege, Ph.D. (Rochester Institute of Technology)
- Tim Margush (University of Akron)
- Sue McFarland Metzger (Villanova University)
- Shyamal Mitra (The University of Texas at Austin)
- Susan Rodger (Duke University)
- Amr Sabry (Indiana University)
- Monica Sweat (Georgia Tech)

Revisores do setor

- Joseph Bowbeer (Consultor)
- Peter Pilgrim (Lloyds TSB)
- José Antonio González Seco (Parlamento de Andalusia)
- S. Sivakumar (Astra Infotech Private Limited)
- Raghavan "Rags" Srinivas (Intuit)
- S. Sivakumar (Astra Infotech Private Limited)

Bem, aí está! O Java é uma linguagem de programação poderosa que ajuda a escrever programas de maneira rápida e eficiente. É perfeitamente adequado ao âmbito do desenvolvimento de sistemas corporativos para ajudar organizações a construir seus sistemas de informação de negócios e missões cruciais. À medida que você ler este livro, apreciaríamos sinceramente seus comentários, críticas, correções e sugestões para melhorar o texto. Envie qualquer correspondência para:

```
universitarios@pearson.com
```

Postaremos correções e esclarecimentos (em inglês) em:

```
www.deitel.com/books/jHTP8/
```

Esperamos que você aprecie ler este livro tanto quanto apreciamos escrevê-lo!
Paul J. Deitel
Dr. Harvey M. Deitel

Sobre os autores

Paul J. Deitel, executivo-chefe de tecnologia da Deitel & Associates, Inc., é graduado pela Sloan School of Management do MIT, onde estudou Tecnologia da Informação. Ele tem as certificações Java Certified Programmer e Java Certified Developer e foi nomeado pela Sun Microsystems como um Java Champion. Por meio da Deitel & Associates, ministrou cursos de programação sobre Java, C, C++, C#, Visual Basic e Internet para clientes da indústria, incluindo Cisco, IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA no Kennedy Space Center, National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, Cable-Data Systems, Nortel Networks, Puma, iRobot, Invensys e muitos mais. Deu palestras sobre Java e C++ para o Boston Chapter da Association for Computing Machinery. Ele e seu coautor, Dr. Harvey M. Deitel, são os autores de livros-texto sobre linguagem de computação líderes de vendas no mundo todo.

Dr. Harvey M. Deitel, executivo-chefe da Deitel & Associates, Inc., tem 48 anos de experiência acadêmica e empresarial no campo da computação. O Dr. Deitel obteve os graus de B.S e M.S. do MIT e um Ph.D. da Boston University. Ele tem 20 anos de experiência em ensino universitário, incluindo nomeação definitiva e servindo como o chefe do Departamento de Ciência da Computação no Boston College antes de fundar a Deitel & Associates, Inc. com seu filho, Paul J. Deitel. Ele e Paul são os coautores de dezenas de livros e pacotes de multimídia e atualmente estão escrevendo vários outros. Com traduções publicadas em chinês tradicional, chinês simplificado, japonês, alemão, russo, espanhol, coreano, francês, polonês, italiano, português, grego, urdu e turco, os textos da Deitel ganharam reconhecimento internacional. O Dr. Deitel proferiu centenas de seminários profissionais para importantes corporações, instituições acadêmicas, organizações governamentais e órgãos militares.

Sobre a Deitel & Associates, Inc.

A Deitel & Associates, Inc. é internacionalmente reconhecida como uma organização corporativa na área de treinamento e autoria especializada em linguagens de programação de computador, tecnologia de software para Internet e Web, treinamento em tecnologia orientada a objetos e desenvolvimento de negócios na Internet por meio da sua Web 2.0 Internet Business Initiative. A empresa fornece cursos voltados a professores nas principais linguagens de programação e plataformas, tais como Java™, C++, C, Visual C#®, Visual Basic®, Visual C++®, XML®, Python®, tecnologia orientada a objetos, programação via Internet e Web e uma crescente lista de cursos adicionais relacionados à programação e desenvolvimento de softwares. Os fundadores da Deitel & Associates, Inc. são Paul J. Deitel e o Dr. Harvey M. Deitel. Os clientes da empresa incluem muitas das maiores empresas de computador do mundo, agências governamentais, áreas do serviço militar e instituições acadêmicas. Por meio da sua parceria de 33 anos na área de publicação com a Prentice Hall/Pearson, a Deitel & Associates, Inc. publica livros universitários e livros profissionais de ponta na área de programação, os cursos *Cyber Classrooms* com multimídia interativa, os cursos *Live-Lessons* baseados em DVD, cursos em vídeo baseados na Web e conteúdo eletrônico para os mais populares sistemas de gerenciamento de cursos. A Deitel & Associates e os autores podem ser contatados via e-mail em:

```
deitel@deitel.com
```

Para aprender mais sobre a Deitel & Associates, Inc., suas publicações e seu programa *Dive Into*® Series Corporate Training ministrado nas premissas dos clientes em todo o mundo, visite:

```
www.deitel.com/training/
```

e inscreva-se para receber o boletim *Deitel*® *Buzz On-line* gratuito via e-mail em:

```
www.deitel.com/newsletter/subscribe.html
```

Conheça a crescente lista dos Deitel Resource Centers em:

```
www.deitel.com/resourcecenters.html
```

Para compra de publicações Deitel, visite:

```
www.deitel.com/books/index.html
```

Antes de você começar

Esta seção contém informações que você deve saber antes de utilizar este livro e instruções para assegurar que seu computador esteja configurado apropriadamente para o uso com este livro. Postaremos atualizações (se houver alguma) para a seção "Antes de você começar" no site Web do livro (em inglês):

```
www.deitel.com/books/jhtp8/
```

Convenções de fontes e nomes

Utilizamos fontes para separar componentes na tela (como nomes de menu e itens de menu) e código ou comandos Java. Nossa convenção é enfatizar componentes na tela utilizando a fonte **Helvetica** em negrito sem serifas (por exemplo, menu **File**) e enfatizar código e comandos Java com uma fonte `Lucida` sem serifas (por exemplo, `System.out.println()`).

Requisitos de sistema de software e hardware

A interface para o conteúdo do CD que acompanha este livro é projetada para iniciar automaticamente por meio do arquivo `AUTORUN.EXE` (o CD é para uso nos sistemas Microsoft® Windows®). Se uma tela de abertura não aparecer quando você inserir o CD no computador, dê um clique duplo no arquivo `welcome.htm` para abrir a interface do CD Student, ou consulte o arquivo `readme.txt` no CD. Na página `welcome.htm`, clique no link **Software** na parte inferior da página para examinar os requisitos de sistema de hardware e software. Leia esses requisitos cuidadosamente antes de instalar o software do CD.

O CD que acompanha este livro contém versões dos seguintes pacotes de software para uso no Microsoft® Windows®:

- O Java™ SE Development Kit (JDK) 6 Update 11 — que foi utilizado para criar e testar todos os programas no livro.
- O IDE do Eclipse para desenvolvedores Java EE versão 3.4.1.
- NetBeans™ IDE Version 6.5 All Bundle.
- MySQL® 5.0 Community Server/v5.0.67.
- MySQL® Connector/J, versão 5.1.7.

O Netbeans e o Eclipse são ambientes de desenvolvimento integrado (IDEs) para desenvolver todos os tipos de aplicativos Java. O MySQL e o MySQL Connector/J são fornecidos para os aplicativos de banco de dados nos capítulos 28–31. Versões para outras plataformas de todas essas ferramentas também estão disponíveis para download, o que será discutido na próxima seção.

Instalando o software

O CD inclui o software requerido para compilar e executar os exemplos neste livro na plataforma Windows. Para outras plataformas, você pode baixar o software. Usuários do Mac OS X podem aprender como utilizar o Java em um Mac em

```
developer.apple.com/java/
```

Os usuários do Linux podem obter o Java SE Development Kit mais recente em

```
java.sun.com/javase/downloads/index.jsp
```

Instruções de instalação e instaladores

No lado esquerdo da página dos requisitos de software e hardware do CD há links para as instruções de instalação para cada pacote de software para o sistema operacional Windows. Cada página de instruções da instalação contém um link para instalador do pacote de software correspondente. Você também pode abrir esses instaladores diretamente da pasta `software` do CD. Siga as instruções de instalação à risca para cada pacote de software. Antes de executar os aplicativos discutidos neste livro ou construir seus próprios aplicativos, você deve instalar o Java Standard Edition Development Kit (JDK) 6 ou uma ferramenta de desenvolvimento Java, como o Netbeans (que exige o JDK) ou os ambientes de desenvolvimento integrado (IDEs) do Eclipse. O Netbeans é necessário para os capítulos 29–31. O Eclipse é opcional para uso com este livro. Fornecemos vídeos para ajudá-lo a começar a trabalhar com o Netbeans e Eclipse em:

```
www.prenhall.com/deitel_br
```

Esses vídeos discutem:

- Como criar programas de único arquivo.
- Como criar programas de múltiplos arquivos.
- Como os IDEs gerenciam arquivos, diretórios e pacotes Java.
- Como utilizar o depurador.
- Como utilizar bibliotecas Java de terceiros.

Downloads para outras plataformas

O software no CD também está disponível para outras plataformas:

- O Netbeans com o pacote JDK: `java.sun.com/javase/downloads/index.jsp`.
- O Netbeans sem o pacote JDK: `www.netbeans.org/downloads/index.html`.
- Eclipse: `www.eclipse.org/downloads/`.
- MySQL Community Edition: `dev.mysql.com/downloads/`.
- MySQL Connector/J: `dev.mysql.com/downloads/connector/j/5.1.html`.

Obtendo os exemplos de código

Os exemplos deste livro estão disponíveis para download em:

```
www.prenhall.com/deitel_br
```

Se você ainda não se registrou no nosso site Web em inglês, acesse `www.deitel.com` e clique no link **Register** abaixo do nosso logotipo no canto superior esquerdo da página. Preencha o formulário com suas informações. Não há nenhum custo para registrar-se e não compartilhamos suas informações com ninguém. Enviamos para você apenas e-mails de gerenciamento de conta a menos que você se registre separadamente para receber por e-mail nosso boletim informativo gratuito *Deitel® Buzz Online* em `www.deitel.com/newsletter/subscribe.html`. Depois de registrar-se, você receberá um e-mail de confirmação com seu código de verificação. *Você precisará desse código para cadastrar-se em* `www.deitel.com` *pela primeira vez*. Configure seu cliente de e-mail para permitir e-mails do `deitel.com` a fim de assegurar que o e-mail de confirmação não seja filtrado como lixo eletrônico.

Em seguida, acesse `www.deitel.com` e cadastre-se utilizando o link **Login** abaixo do nosso logotipo no canto superior esquerdo da página. Acesse `www.deitel.com/books/jhtp8/`. Clique no link **Examples** para fazer o download do arquivo `Examples.zip`. Anote o local onde você quer salvar o arquivo no seu computador. Esta é outra forma de obter acesso aos exemplos.

Supomos que os exemplos estejam localizados em `C:\Examples` no seu computador. Extraia o conteúdo do `Examples.zip` utilizando uma ferramenta como o WinZip (`www.winzip.com`) ou os recursos internos do Windows XP e Windows Vista (ou uma ferramenta semelhante em outras plataformas).

Configurando a variável de ambiente `PATH`

A variável de ambiente `PATH` no seu computador especifica em quais diretórios o computador pesquisa ao procurar aplicativos, como os aplicativos que permitem compilar e executar seus aplicativos Java (chamados `javac` e `java`, respectivamente). *Siga atentamente as instruções de instalação para o Java na sua plataforma a fim de certificar-se de que você configurou a variável de ambiente* `PATH` *corretamente.*

Se você não configurar a variável `PATH` corretamente, ao utilizar as ferramentas do JDK, você receberá uma mensagem como:

```
'java' is not recognized as an internal or external command, operable program or batch file.
```

Nesse caso, volte às instruções de instalação para configurar a variável PATH e verifique novamente seus passos. Se baixou uma versão mais recente do JDK, talvez seja necessário mudar o nome do diretório de instalação do JDK na variável PATH.

Configurando a variável de ambiente CLASSPATH

Se tentar executar um programa Java e receber uma mensagem como:

```
Exception in thread "main" java.lang.
NoClassDefFoundError: NomeDaClasse
```

então seu sistema tem um variável de ambiente CLASSPATH que deve ser modificada. Para corrigir esse problema, siga os passos da configuração da variável de ambiente PATH, localize a variável CLASSPATH, e então edite o valor da variável para incluir o diretório local — tipicamente representado por um ponto (.). No Windows adicione:

```
.;
```

ao início do valor CLASSPATH (sem espaços antes ou depois desses caracteres). Em outras plataformas, substitua o ponto-e-vírgula pelos caracteres separadores de caminho apropriados — muitas vezes um dois-pontos (:):

A nova interface Nimbus do Java

A partir da atualização 10 do Java SE 6, o Java é distribuído com uma interface nova, elegante e compatível com várias plataformas conhecida como Nimbus. Para programas com interfaces gráficas com o usuário, configuramos nossos sistemas para utilizar o Nimbus como a interface padrão.

Para configurar o Nimbus como o padrão para todos os aplicativos Java, você precisa criar um arquivo de texto chamado `swing.properties` na pasta `lib` tanto da sua pasta de instalação do JDK como da sua pasta de instalação do JRE. Insira a seguinte linha do código no arquivo:

```
swing.defaultlaf=com.sun.java.swing.plafnimbus.NimbusLookAndFeel
```

Para mais informações sobre a localização dessas pastas de instalação, visite `java.sun.com/javase/6/webnotes/install/index.html`. [*Nota*: além do JRE autônomo, há um JRE aninhado na pasta de instalação do seu JDK. Se estiver utilizando um IDE que depende do JDK (por exemplo, Netbeans), talvez você também precise inserir o arquivo `swing.properties` na pasta `lib` aninhada na pasta `jre`.]

Agora você está pronto para começar seus estudos do Java com este livro. Esperamos que você goste dele!

Nossa vida é desperdiçada em detalhes... Simplifique, simplifique.
— **Henry David Thoreau**

O principal mérito da língua é a clareza.
— **Galeno**

My object all sublime
I shall achieve in time.
(*Devo alcançar a tempo meu sublime objetivo.*)
— **W. S. Gilbert**

Ele tinha um talento maravilhoso para empacotar bem os pensamentos e torná-los portáteis.
— **Thomas B. Macaulay**

Meu Deus, entre os dois, acho que o mais difícil de entender é o intérprete!
— **Richard Brinsley Sheridan**

O homem ainda é o computador mais extraordinário.
— **John F. Kennedy**

1

Introdução aos computadores, à Internet e à World Wide Web

Objetivos

Neste capítulo, você aprenderá:

- Os conceitos básicos de rede, hardware e software de computador.
- Os conceitos básicos de tecnologia de objeto, como classes, objetos, atributos, comportamentos, encapsulamento e herança.
- Os diferentes tipos de linguagens de programação e quais linguagens são mais amplamente utilizadas.
- Um ambiente de desenvolvimento de programa Java típico.
- O papel do Java no desenvolvimento de aplicativos cliente/servidor distribuídas para a Internet e a Web.
- A história da UML — a linguagem de design orientado a objetos padrão no setor.
- A história da Internet e da World Wide Web até a Web 2.0.
- A fazer test-drive de aplicativos Java.
- Algumas das principais tecnologias de software mais recentes.

Sumário

1.1 Introdução
1.2 Computadores: hardware e software
1.3 Organização do computador
1.4 Primeiros sistemas operacionais
1.5 Computação pessoal distribuída e computação cliente/servidor
1.6 A Internet e a World Wide Web
1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível
1.8 História do C e do C++
1.9 História do Java
1.10 Bibliotecas de classe do Java
1.11 Fortran, Cobol, Pascal e Ada
1.12 Basic, Visual Basic, Visual C++, C# e .NET
1.13 Ambiente típico de desenvolvimento Java
1.14 Notas sobre o Java e este livro
1.15 Testando um aplicativo Java
1.16 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML
1.17 Web 2.0
1.18 Tecnologias de software
1.19 Conclusão
1.20 Recursos da Web

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

1.1 Introdução

Bem vindo ao Java — uma poderosa linguagem de programação de computador que é divertida para iniciantes aprenderem e adequada para programadores experientes utilizarem na construção de sistemas de informações empresariais importantes. A oitava edição de *Java — Como programar* é uma ferramenta de aprendizagem efetiva para cada uma dessas audiências.

Pedagogia

O livro enfatiza basicamente como alcançar a *clareza* do programa pelas técnicas comprovadas da *programação orientada a objetos*. Os não programadores aprenderão programação da maneira certa desde o início. A apresentação é clara, simples e ricamente ilustrada. Inclui centenas de programas Java e mostra as saídas produzidas quando eles são executados em um computador. Ensinamos recursos de Java no contexto de programas funcionais completos — chamamos isso de **abordagem *live-code***. Os programas de exemplo podem ser baixados de `www.deitel.com/books/jhtp8/` ou `www.prenhall.com/deitel_br` (consulte a Seção "Antes de você começar" depois do Prefácio).

Fundamentos

Os primeiros capítulos introduzem princípios básicos de informática, programação de computadores e a linguagem de programação Java, fornecendo uma base sólida para o tratamento mais profundo de Java nos capítulos posteriores. Programadores experientes tendem a ler os primeiros capítulos rapidamente e a considerar rigoroso e desafiante o tratamento de Java nos capítulos seguintes.

A maioria das pessoas está familiarizada com as extraordinárias tarefas que os computadores realizam. Utilizando este manual, você aprenderá a escrever instruções que fazem com que os computadores realizem essas tarefas. Software (isto é, as instruções que você escreve); hardware (isto é, os computadores). O Java, desenvolvido pela Sun Microsystems, é uma das linguagens mais populares da atualidade para o desenvolvimento de software.

Evolução da computação e da programação

O uso de computadores está aumentando em quase todos os campos de trabalho. Os custos da computação estão caindo drasticamente, devido aos rápidos avanços nas tecnologias de hardware e software. Os computadores que ocupavam grandes salas e custavam milhões de dólares algumas décadas atrás agora podem ser gravados em chips de silício menores que uma unha, ao custo de apenas alguns poucos dólares. Felizmente, o silício é um dos materiais mais abundantes na Terra — é um ingrediente da areia comum. A tecnologia do chip de silício tornou a computação tão econômica que mais de um bilhão de computadores de uso geral estão em uso em todo o mundo, auxiliando pessoas no comércio, na indústria, no governo e na vida pessoal. O número pode facilmente dobrar nos próximos poucos anos.

Ao longo da sua formação acadêmica e profissional, muitos programadores aprenderam uma metodologia conhecida como programação estruturada. Você aprenderá a programação estruturada e uma metodologia empolgante e mais recente, a **programação orientada a objetos**. Por que ensinamos ambas? Atualmente, a orientação a objeto é a principal metodologia de programação utilizada pelos programadores. Você irá criar e trabalhar com muitos objetos de software neste texto. Mas descobrirá que a estrutura interna deles costuma ser construída com técnicas de programação estruturada. Além disso, a lógica de manipular objetos é ocasionalmente expressa com programação estruturada.

Linguagem preferida para aplicativos em rede

O Java tornou-se a linguagem preferida para implementar aplicativos baseados na Internet e software para dispositivos que se comunicam por uma rede. Equipamentos de som estéreo e outros dispositivos domésticos muitas vezes são conectados em rede pela tecnologia Java. Atualmente, existem bilhões de celulares e dispositivos portáteis compatíveis com Java! O Java é a linguagem preferida para atender às necessidades de programação de muitas organizações.

O Java evoluiu tão rapidamente que esta oitava edição do livro — baseada na **Java Standard Edition (Java SE)** 6, atualização 11 — foi publicada exatamente 13 anos depois da primeira edição. O Java é utilizado para um espectro de aplicações tão amplo que ele tem duas outras versões. A **Java Enterprise Edition (Java EE)** é adequada para desenvolver aplicativos distribuídos em rede em larga escala e aplicativos baseados na Web — discutiremos vários recursos do Java EE mais adiante. A **Java Micro Edition (Java ME)** é voltada para o desenvolvimento de aplicativos de pequenos dispositivos com limitações de memória, como telefones celulares, pagers e PDAs.

Como entrar em contato com os autores

Se você quiser se comunicar conosco, envie um e-mail para `deitel@deitel.com`. Responderemos prontamente. Para manter-se atualizado com o desenvolvimento do Java na Deitel & Associates, registre-se em nosso boletim gratuito distribuído por e-mail, *The Deitel® Buzz Online*, em:

```
www.deitel.com/newsletter/subscribe.html
```

Uma grande quantidade de material adicional sobre o Java pode ser encontrada em nossos Java Resource Centers, uma lista de recursos Java que cresce continuamente em `www.deitel.com/ResourceCenters.html`. Esperamos que você goste de aprender com esta nova edição de *Java — Como programar*.

1.2 Computadores: hardware e software

Um **computador** é um dispositivo que pode realizar cálculos e tomar decisões lógicas fenomenalmente mais rápido do que os seres humanos. Boa parte dos computadores pessoais de hoje em dia podem realizar bilhões de cálculos em um segundo. Uma pessoa operando uma calculadora não pode realizar esse número de cálculos em uma vida inteira. (Questões a ponderar: como você saberia se a pessoa somou todos esses números corretamente? Como você saberia se o computador somou os números corretamente?) **Supercomputadores** já realizam *milhares de trilhões* (*quatrilhões*) de instruções por segundo! Para colocar isso em perspectiva, um computador de quatrilhões de instruções por segundo pode realizar mais de 100.000 cálculos por segundo para cada pessoa no planeta!

Os computadores processam **dados** sob o controle de conjuntos de instruções chamados **programas de computador**. Esses programas orientam o computador por meio de conjuntos ordenados de ações especificadas por pessoas chamadas **programadores de computador**.

Um computador consiste em vários dispositivos conhecidos como **hardware** (por exemplo, o teclado, tela, mouse, discos, memória, unidades de DVD, CD-ROM e de processamento). Os programas que executam em um computador são referidos como **software**. Os custos de hardware têm caído significativamente nos últimos anos, a ponto de os computadores pessoais terem se tornado um bem de consumo popular. Neste livro, você aprenderá metodologias comprovadas que podem reduzir custos de desenvolvimento de software — a programação orientada a objetos e (em nosso estudo de caso opcional de engenharia de software nos capítulos 12–13) o design orientado a objetos.

1.3 Organização do computador

Independentemente das diferenças na aparência física, praticamente todos os computadores podem ser divididos em várias **unidades lógicas** ou seções:

1. ***Unidade de entrada***. Essa seção de "recebimento" obtém informações (dados e programas de computador) de **dispositivos de entrada** e as coloca à disposição de outras unidades para serem processadas. A maioria das informações é inserida em computadores por meio de dispositivos de entrada, como teclados e mouse. Informações também podem ser inseridas de muitos outros modos, incluindo falar com um computador, digitalização de imagens e leitura de códigos de barra, leitura a partir de dispositivos de armazenamento secundário (como unidades de disco rígido, unidades de CD, unidades de DVD e unidades USB) e os computadores também recebem informações da Internet (por exemplo, ao fazer o download de vídeos do YouTube™, e-books da Amazon etc).
2. ***Unidade de saída***. Essa seção de "entrega" pega as informações que o computador processou e as coloca em vários **dispositivos de saída** para torná-las disponíveis para serem utilizadas fora do computador. Hoje, a maioria das informações produzidas a partir dos computadores é exibida em telas, impressas em papel, reproduzidas em players de áudio (como os populares iPods da Apple), ou utilizadas para controlar outros dispositivos. Os computadores também podem gerar saída de suas informações para redes, como a Internet.
3. ***Unidade de memória***. Essa seção de "armazenamento" de acesso rápido e relativamente baixa capacidade retém as informações que foram inseridas por meio da unidade de entrada, tornando-as imediatamente disponíveis para processamento quando necessário. A unidade de memória também retém informações processadas até que elas possam ser colocadas em dispositivos de saída pela unidade de saída. As informações na unidade de memória são **voláteis** — em geral, são perdidas quando o computador é desligado. A unidade de memória costuma ser chamada de **memória** ou **memória principal**.

4. ***Unidade de aritmética e lógica* (*Arithmetic and Logic Unit* — ALU)**. Essa seção de "produção" realiza cálculos, como adição, subtração, multiplicação e divisão. Essa seção também contém os mecanismos de decisão que permitem ao computador, por exemplo, comparar dois itens da unidade de memória para determinar se são iguais ou não. Nos sistemas atuais, a ALU normalmente é implementada como parte da unidade lógica, a CPU.
5. **Unidade central de processamento (*Central Processing Unit* — CPU)**. Essa "seção administrativa" coordena e supervisiona a operação das outras seções. A CPU instrui a unidade de entrada sobre quando as informações devem ser lidas e transferidas para a unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas em cálculos e instrui a unidade de saída sobre quando enviar as informações da unidade de memória para certos dispositivos de saída. Muitos computadores de hoje têm múltiplas CPUs e, portanto, podem realizar muitas operações simultaneamente — esses computadores são chamados de **multiprocessadores**. Um **processador de múltiplos núcleos** (ou **multi-core**) implementa o multiprocessamento em um único chip de circuito integrado — por exemplo um processador de dois núcleos (ou dual-core) tem duas CPUs e um processador de quatro núcleos (ou quad-core) tem quatro.
6. **Unidade de armazenamento secundária.** Essa é a seção de "armazenamento" de longo prazo e alta capacidade. Programas ou dados que não são utilizados ativamente pelas outras unidades, em geral, são colocados em dispositivos de armazenamento secundário (por exemplo, unidades de disco) até que sejam necessários, possivelmente horas, dias, meses ou até mesmo anos mais tarde. Portanto, diz-se que as informações nos dispositivos de armazenamento secundário são **persistentes** — são preservadas mesmo quando a energia elétrica é desligada. As informações de armazenamento secundário exigem muito mais tempo para serem acessadas do que as informações na memória principal, mas o custo por unidade de armazenamento secundário é muito menor que o da memória principal. Exemplos de dispositivos de armazenamento secundário incluem CDs, DVDs e unidades flash (às vezes chamadas cartões de memória), que podem armazenar centenas de milhões a bilhões de caracteres.

1.4 Primeiros sistemas operacionais

Os primeiros computadores podiam realizar apenas um **trabalho** ou **tarefa** por vez. Isso muitas vezes é chamado **processamento em lotes** de um único usuário e era a regra na década de 1950. O computador executava um único programa ao processar dados em grupos ou **lotes**. Os usuários geralmente submetiam seus trabalhos a um centro de processamento de dados em unidades de cartões perfurados e muitas vezes esperavam horas ou mesmo dias antes que as listagens estivessem prontas.

Sistemas operacionais de software foram desenvolvidos para tornar o uso dos computadores mais conveniente. Os primeiros sistemas operacionais facilitavam e aceleravam a transição entre trabalhos, aumentando a quantidade de trabalho, ou **throughput**, que os computadores poderiam processar em um dado tempo.

Quando os computadores tornaram-se mais poderosos, tornou-se evidente que o processamento em lotes de um único usuário era ineficiente, porque uma grande quantidade de tempo era gasta esperando dispositivos de entrada/saída lentos completarem suas tarefas. A ideia foi concebida para que muitos trabalhos ou tarefas pudessem compartilhar os recursos do computador para alcançar melhor utilização. Isso é chamado de **multiprogramação**. A multiprogramação envolve a operação simultânea de muitos trabalhos que competem para compartilhar os recursos do computador.

Na década de 1960, vários grupos na indústria e universidades foram os pioneiros dos sistemas operacionais de **compartilhamento de tempo**. O compartilhamento de tempo é um caso especial da multiprogramação em que usuários acessam o computador por meio de terminais, em geral dispositivos com teclados e telas. Dúzias ou até mesmo centenas de usuários compartilham o computador de uma vez. Na verdade, o computador não executa todos os trabalhos simultaneamente. Em vez disso, ele executa uma pequena parte do trabalho de um usuário e, em seguida, atende o próximo usuário, talvez fornecendo serviço para cada usuário várias vezes por segundo. Portanto, os programas dos usuários *parecem* estar executando simultaneamente. Uma vantagem do compartilhamento de tempo é que as solicitações de usuário recebem respostas quase imediatas.

1.5 Computação pessoal distribuída e computação cliente/servidor

Em 1977, a Apple Computer popularizou a **computação pessoal**. Os computadores tornaram-se tão econômicos que as pessoas podiam comprá-los para utilização pessoal ou profissional. Em 1981, a IBM, o maior fornecedor de computadores do mundo, lançou o IBM Personal Computer. Isso legitimou rapidamente a computação pessoal em negócios, indústria e organizações governamentais.

Esses computadores eram unidades "independentes" — as pessoas transportavam discos de um lado para o outro entre si para o compartilhamento de informações (um modo de transferência frequentemente chamado de "rede peão"). Embora computadores pessoais antigos não fossem suficientemente poderosos para suportar vários usuários simultâneos, essas máquinas podiam ser interconectadas em redes de computadores, às vezes por linhas telefônicas e às vezes em **redes locais (*Local Area Networks* — *LANs*)** dentro de uma organização. Isso levou ao fenômeno da **computação distribuída**, em que a computação de uma organização, em vez de ser realizada somente em alguma instalação central, é distribuída por redes para os locais em que o trabalho da organização é realizado. Os computadores pessoais eram poderosos o bastante para tratar os requisitos de computação de usuários individuais bem como as tarefas básicas de comunicação de passar eletronicamente informações entre computadores.

Hoje os computadores pessoais são tão poderosos quanto as máquinas de milhões de dólares de apenas poucas décadas atrás. As máquinas desktop mais poderosas — chamadas **estações de trabalho** — fornecem a usuários individuais capacidades enormes. As informações

são facilmente compartilhadas por redes de computadores, onde computadores chamados **servidores** armazenam dados que podem ser utilizados por computadores **clientes** distribuídos por toda a rede — daí o termo **computação cliente/servidor**. O Java tornou-se amplamente utilizado para escrever software de rede de computador e aplicativos distribuídos cliente/servidor. Os sistemas operacionais populares de hoje, como o Linux, o Mac OS X (pronuncia-se "O-S dez") da Apple e o Microsoft Windows, fornecem os tipos de capacidades necessárias atualmente.

1.6 A Internet e a World Wide Web

A **Internet** — uma rede global de computadores — tem suas raízes na década de1960, quando o Departamento de Defesa dos Estados Unidos disponibilizou recursos financeiros. Originalmente projetada para conectar os principais sistemas de computadores de cerca de uma dúzia de universidades e organizações de pesquisa, a Internet atualmente é acessível a bilhões de computadores e dispositivos controlados por computadores no mundo todo.

Com o surgimento da **World Wide Web** — que permite aos usuários de computador localizar e visualizar documentos baseados em multimídia sobre quase qualquer assunto pela Internet — a Internet explodiu, tornando-se um dos principais mecanismos de comunicação do mundo.

Antigamente, a maioria dos aplicativos de computador executava em computadores que não se comunicavam entre si. Hoje em dia, é possível escrever aplicativos que permitem que os computadores se comuniquem por todo o mundo. A Internet, combinando tecnologias de computação e comunicação, torna o trabalho das pessoas mais fácil. Ela torna informações acessíveis mundialmente de forma instantânea e conveniente. Permite aos indivíduos e empresas de pequeno porte local obter exposição mundial. Ela está mudando a maneira como os negócios são feitos. As pessoas podem procurar os melhores preços em praticamente todos os produtos ou serviços. Comunidades de interesse especial podem permanecer em contato entre si. Os pesquisadores podem tornar-se instantaneamente cientes dos últimos avanços científicos.

Java — Como programar apresenta técnicas de programação que permitem aos aplicativos Java utilizar a Internet e a Web para interagir com outros aplicativos. Essas capacidades e outras permitem desenvolver o tipo de aplicativos distribuídos de nível corporativo utilizados no setor hoje em dia. Os aplicativos Java podem ser escritos para executar portatilmente em todo tipo de computador importante, reduzindo significativamente o tempo e custo de desenvolvimento de sistemas.

1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível

Os programadores escrevem instruções em várias linguagens de programação, algumas diretamente compreensíveis por computadores e, outras, requerendo passos intermediários de **tradução**. Centenas dessas linguagens estão em uso atualmente. Essas linguagens podem ser divididas em três tipos gerais:

1. linguagens de máquina
2. linguagens assembly
3. linguagens de alto nível

Qualquer computador pode entender diretamente somente sua própria **linguagem de máquina**. Essa é a "linguagem natural" do computador, definida pelo seu design de hardware. As linguagens de máquina consistem geralmente em strings de números (em última instância reduzidas a 1s e 0s) que instruem os computadores a realizar suas operações mais elementares uma de cada vez. As linguagens de máquina são **dependentes de máquina** (uma linguagem particular de máquina pode ser utilizada apenas em um tipo de computador). Essas linguagens são complicadas para seres humanos. Por exemplo, eis uma seção de um primeiro programa de linguagem de máquina que adiciona o pagamento de horas extras à base de pagamentos e armazena o resultado como salário bruto:

```
+1300042774
+1400593419
+1200274027
```

A programação na linguagem de máquina era um processo muito lento e tedioso para a maioria dos programadores. Em vez de utilizar as strings de números que os computadores poderiam entender diretamente, os programadores começaram a utilizar abreviações em inglês para representar operações elementares. Essas abreviações formaram a base de **linguagens assembly**. **Programas tradutores** chamados **assemblers** foram desenvolvidos para converter os primeiros programas de linguagem assembly em linguagem de máquina a velocidades de computador. A seção a seguir de um programa de linguagem assembly também soma os ganhos em horas extras ao salário de base e armazena o resultado no salário bruto:

```
load        basepay
add         overpay
store       grosspay
```

Embora tal código seja mais claro para humanos, ele é incompreensível para computadores até ser traduzido em linguagem de máquina.

O uso de computador aumentou rapidamente com o advento de linguagens assembly, mas os programadores ainda tinham de utilizar muitas instruções para realizar até as tarefas mais simples. Para acelerar o processo de programação, foram desenvolvidas **linguagens de alto nível** em que instruções únicas poderiam ser escritas para realizar tarefas substanciais. Os programas tradutores chamados **compila-**

dores convertem os programas de linguagem de alto nível em linguagem de máquina. Linguagens de alto nível permitem aos programadores escrever instruções que se parecem com o inglês cotidiano e contêm notações matemáticas comumente utilizadas. Um programa de folha de pagamento escrito em uma linguagem de alto nível poderia conter uma instrução assim:

```
grossPay = basePay + overTimePay
```

Do ponto de vista do programador, as linguagens de alto nível são preferíveis às linguagens de máquina e às linguagens assembly. O C, C++, as linguagens .NET da Microsoft (por exemplo, Visual Basic, Visual C++ e C#) estão entre as linguagens de programação de alto nível mais usadas; o Java é de longe a linguagem *mais amplamente* utilizada.

O processo de compilação de um programa de linguagem de alto nível em linguagem de máquina pode consumir uma quantidade considerável de tempo do computador. Os programas **interpretadores** foram desenvolvidos para executar programas de linguagem de alto nível diretamente (sem o tempo de espera da compilação), embora mais lentos do que a execução de programas compilados. Discutiremos mais sobre como os interpretadores funcionam na Seção 1.13, onde você aprenderá que o Java utiliza uma combinação inteligente de compilação e interpretação para executar programas. Os exercícios 7.35–7.37 (na Seção especial "Construindo seu próprio computador") orientam pelo processo da construção de um programa interpretador.

1.8 História do C e do C++

O Java evoluiu a partir do C++, que evoluiu do C, que evoluiu a partir do BCPL e do B. O BCPL foi desenvolvido em 1967 por Martin Richards como uma linguagem para escrever software de sistemas operacionais e compiladores. Ken Thompson modelou muitos recursos em sua linguagem B com base nas suas contrapartes em BCPL, utilizando o B para criar versões anteriores do sistema operacional UNIX na Bell Laboratories em 1970.

A linguagem C — originalmente implementada em 1972 — foi desenvolvida a partir da linguagem B criada por Dennis Ritchie nos Bell Laboratories. Ela inicialmente tornou-se amplamente conhecida como a linguagem de desenvolvimento do sistema operacional UNIX. Hoje, a maior parte do código para sistemas operacionais de uso geral (por exemplo, aqueles encontrados em notebooks, áreas de trabalho, estações de trabalho e pequenos servidores) é escrita em C ou C++.

O C++, uma extensão do C, foi desenvolvido por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ fornece vários recursos que "sofisticam" a linguagem C, mas, sobretudo, fornece capacidades para a *programação orientada a objetos* (discutida em mais detalhes na Seção 1.16 e em todo este livro). O C ++ é uma linguagem híbrida — é possível programar tanto no estilo C como no estilo orientado a objetos, ou em ambos.

Construir software de maneira rápida, correta e econômica continua sendo um objetivo difícil de alcançar numa época em que a demanda por novos e mais poderosos programas de software cresce vertiginosamente. *Objetos*, ou mais precisamente — como veremos na Seção 1.16 — as classes a partir das quais os objetos se originam, são componentes de software essencialmente reutilizáveis. Há objetos data, objetos tempo, objetos áudio, objetos vídeo, objetos automóveis, objetos pessoas e assim por diante. De fato, praticamente todos os substantivos podem ser representados como um objeto de software em termos de **atributos** (por exemplo, nome, cor e tamanho) e **comportamentos** (por exemplo, calcular, mover e comunicar). Os desenvolvedores de software estão descobrindo que utilizar uma abordagem de implementação de design orientado a objetos modular pode tornar grupos de desenvolvimento de software muito mais produtivos do que era possível com as primeiras técnicas populares, como a programação estruturada. Programas orientados a objetos são frequentemente mais fáceis de entender, corrigir e modificar. O Java é a linguagem de programação orientada a objetos mais amplamente utilizada do mundo.

1.9 História do Java

A contribuição mais importante da revolução do microprocessador até essa data é que ele tornou possível o desenvolvimento de computadores pessoais, que agora contam com mais de um bilhão em todo o mundo. Os computadores pessoais afetaram profundamente a vida das pessoas e a maneira que as organizações conduzem e gerenciam seus negócios.

Os microprocessadores têm um impacto profundo em dispositivos eletrônicos inteligentes de consumo popular. Reconhecendo isso, a Sun Microsystems, em 1991, financiou um projeto de pesquisa corporativa interna que resultou em uma linguagem baseada em C++ que seu criador, James Gosling, chamou de Oak em homenagem a uma árvore de carvalho vista por sua janela na Sun. Descobriu-se mais tarde que já havia uma linguagem de computador com esse nome. Quando uma equipe da Sun visitou uma cafeteria local, o nome Java (cidade de origem de um tipo de café importado) foi sugerido; e o nome pegou.

O projeto de pesquisa passou por algumas dificuldades. O mercado para dispositivos eletrônicos inteligentes destinados ao consumidor final não estava se desenvolvendo tão rapidamente como a Sun tinha previsto. Por uma feliz casualidade, a Web explodiu em popularidade em 1993 e a Sun viu o potencial de utilizar o Java para adicionar **conteúdo dinâmico**, como interatividade e animações, às páginas da Web. Isso deu nova vida ao projeto.

A Sun anunciou o Java formalmente em uma conferência do setor em maio de 1995. O Java chamou a atenção da comunidade de negócios por causa do enorme interesse na Web. O Java é agora utilizado para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores da Web (os computadores que fornecem o conteúdo que vemos em nossos navegadores da Web), fornecer aplicativos para dispositivos voltados para o consumo popular (como telefones celulares, pagers e PDAs) e para muitos outros propósitos.

1.10 Bibliotecas de classe do Java

Programas Java consistem em partes chamadas classes. As classes incluem partes chamadas **métodos** que realizam tarefas e retornam informações quando as tarefas são concluídas. Você pode criar cada parte necessária para formar seus programas Java. Entretanto, a maioria dos programadores Java tira proveito das ricas coleções de classes existentes nas **bibliotecas de classe Java**, que também são conhecidas como **Java APIs (Application Programming Interfaces)**. Portanto, na realidade há dois aspectos para aprender o "mundo" do Java. O primeiro aspecto é a própria linguagem Java, de modo que você possa programar suas próprias classes, o segundo são as classes nas extensas bibliotecas de classe Java. Por todo este livro discutimos muitas classes da Java API preexistentes. Para baixar a documentação da Java API, acesse `java.sun.com/javase/downloads/`, role para baixo até a Seção Additional Resources e clique no botão Download à direita de Java SE 6 Documentation.

Observação de engenharia de software 1.1
Utilize uma abordagem de bloco de construção para criar seus programas. Evite reinventar a roda — utilize partes existentes onde quer que possível. Essa reutilização de software é um benefício fundamental da programação orientada a objetos.

Incluímos muitas **Observações de engenharia de software** por todo o livro para explicar conceitos que afetam e aprimoram a arquitetura total e a qualidade de sistemas de software. Também destacamos outros tipos de dicas, incluindo:

- **Boas práticas de programação** (para ajudá-lo a escrever programas que são mais claros, mais compreensíveis, mais fáceis de manter e mais fáceis de testar e **depurar** — isto é, remover erros de programação).
- **Erros de programação comuns** (problemas a observar e evitar).
- **Dicas de desempenho** (técnicas para escrever programas que executam mais rapidamente e utilizam menos memória).
- **Dicas de portabilidade** (técnicas para ajudá-lo a escrever programas que podem executar, com pouca ou nenhuma modificação, em diferentes computadores — essas dicas também incluem observações gerais de como o Java alcança seu alto grau de portabilidade).
- **Dicas de prevenção de erros** (técnicas para remover bugs dos seus programas e, mais importante, antes de tudo, técnicas para escrever programas sem bugs).
- **Observações sobre a interface** (técnicas para ajudá-lo a projetar a "aparência" e o "comportamento" das interfaces com o usuário dos seus aplicativos em termos da aparência e facilidade de uso).

Muitas dessas técnicas são apenas diretrizes. Sem dúvida, você desenvolverá seu próprio estilo de programação preferido.

Observação de engenharia de software 1.2
Ao programar em Java, você geralmente utilizará os seguintes blocos de construção: classes e métodos de bibliotecas de classe, classes e métodos que você mesmo cria e classes e métodos que outros criam e tornam disponíveis para você.

A vantagem de criar suas próprias classes e métodos é que você sabe exatamente como eles funcionam e pode examinar o código Java. A desvantagem é o consumo de tempo e o esforço potencialmente complexo que é necessário.

Dica de desempenho 1.1
Usar as classes e métodos da Java API em vez de escrever suas próprias versões pode melhorar o desempenho de programa, porque eles são cuidadosamente escritos para executar de modo eficiente. Essa técnica também diminui o tempo de desenvolvimento de programa.

Dica de portabilidade 1.1
Utilizar as classes e métodos da Java API em vez de escrever suas próprias versões melhora a portabilidade de programa, porque esses são incluídos em cada implementação Java.

1.11 Fortran, Cobol, Pascal e Ada

Centenas de linguagens de alto nível foram desenvolvidas, mas somente algumas alcançaram ampla aceitação. O **Fortran** (*FORmula TRANslator*) foi uma linguagem desenvolvida pela IBM Corporation em meados da década de 1950 para ser utilizada em aplicativos científicos de engenharia que exigem complexos cálculos matemáticos. O Fortran ainda é amplamente utilizado em aplicativos de engenharia.

O **Cobol (*COmmon Business Oriented Language*)** foi desenvolvido no final da década de 1950 por fabricantes de computadores e usuários de computadores do governo norte-americano e da indústria. O Cobol é utilizado para aplicativos comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados. Grande parte do software utilizado em grandes empresas varejistas ainda é programada em Cobol.

Durante a década de 1960, muitos grandes esforços no desenvolvimento de software encontraram diversas dificuldades. Em geral, os prazos de entrega de software atrasavam, os custos excediam enormemente os orçamentos e os produtos finais não eram confiáveis. As pessoas começaram a perceber que o desenvolvimento de software era uma atividade muito mais complexa do que haviam imaginado. Uma pesquisa feita na década de 1960 resultou na evolução da **programação estruturada** — uma abordagem disciplinada para escrever programas mais claros, mais fáceis de testar e depurar e mais fáceis de modificar do que os grandes programas produzidos com a técnica anterior.

Um dos resultados mais tangíveis dessa pesquisa foi o desenvolvimento da linguagem de programação Pascal pelo professor Nicklaus Wirth em 1971. Batizada em homenagem a Blaise Pascal, matemático e filósofo do século XVII, essa linguagem foi adotada para ensino de programação estruturada em ambientes acadêmicos e rapidamente se tornou a linguagem de programação preferida na maioria das universidades.

A linguagem de programação **Ada** foi desenvolvida sob o patrocínio do Departamento de Defesa dos Estados Unidos durante a década de 1970 e o início da década de 1980. O DOD queria que uma única linguagem atendesse a maioria de suas necessidades. A linguagem foi batizada como Ada, em homenagem a Lady Ada Lovelace, filha do poeta Lord Byron. Lady Lovelace é considerada a primeira pessoa a escrever um programa de computador do mundo no início do século XIX (para o dispositivo mecânico de computação conhecido como Máquina Analítica, projetado por Charles Babbage). Uma capacidade importante da linguagem Ada, chamada **multitarefa**, permite que os programadores especifiquem quantas atividades em um programa devem ocorrer paralelamente. O Java, por meio de uma técnica chamada *multithreading* (discutida no Capítulo 26), também permite aos programadores escrever programas com atividades paralelas.

1.12 Basic, Visual Basic, Visual C++, C# e .NET

A linguagem de programação **Basic** (Beginner's All-Purpose Symbolic Instruction Code) foi desenvolvida em meados da década de 1960, no Dartmouth College, como um meio de escrever programas simples. O principal propósito de Basic foi familiarizar os iniciantes com as técnicas de programação.

A linguagem **Visual Basic** da Microsoft foi introduzida no início de 1990 para simplificar o desenvolvimento de aplicativos Microsoft Windows e é uma das linguagens de programação mais populares do mundo.

As últimas ferramentas de desenvolvimento da Microsoft fazem parte de sua estratégia de escopo corporativo para integrar a Internet e a Web em aplicativos de computador. Essa estratégia é implementada na **plataforma .NET** da Microsoft, que fornece aos desenvolvedores as capacidades necessárias para criar e executar aplicativos de computador que possam executar em computadores distribuídos através da Internet. As três principais linguagens de programação da Microsoft são **Visual Basic** (baseada no Basic original), **Visual C++** (baseada no C++) e **C#** (baseada no C++ e no Java, e desenvolvida especificamente para a plataforma .NET). Os desenvolvedores que utilizam .NET podem escrever componentes de software na linguagem com que se sentem à vontade, então formar aplicativos combinando esses componentes com outros escritos em qualquer linguagem de .NET.

1.13 Ambiente típico de desenvolvimento Java

Agora explicamos os passos comumente seguidos na criação e execução de um aplicativo Java que utiliza um ambiente de desenvolvimento Java (ilustrado na Figura 1.1).

Em geral, programas Java passam por cinco fases — edição, compilação, carregamento, verificação e execução. Discutimos essas fases no contexto do Java SE Development Kit 6 (JDK6) da Sun Microsystems, Inc. Você pode baixar o JDK mais recente e sua documentação de `java.sun.com/javase/downloads/`. *Siga atentamente as instruções de instalação do JDK fornecidas* na *Seção* "Antes de você começar" *deste livro a fim de garantir a correta configuração do computador para compilar e executar programas Java.* Também é recomendável visitar o New to Java Center da Sun em:

```
java.sun.com/new2java/
```

[*Nota:* esse site da Web fornece instruções de instalação para Windows, Linux e Mac OS X. Se você não estiver utilizando um desses sistemas operacionais, consulte a documentação do ambiente Java do seu sistema ou pergunte a seu instrutor como realizar essas tarefas com base no sistema operacional do seu computador. Se tiver problemas com esse link ou com quaisquer outros referidos neste livro, verifique erratas e nos informe por e-mail em `deitel@deitel.com`.]

Fase 1: criando um programa

A Fase 1 consiste em editar um arquivo com um **programa editor** (em geral, conhecido simplesmente como um **editor**). Você digita um programa Java (em geral referido como **código-fonte**) utilizando o editor, faz quaisquer correções necessárias e salva o programa em um dispositivo de armazenamento secundário, como sua unidade de disco. Um nome de arquivo que termina com a **extensão `.java`** indica que o arquivo contém o código-fonte Java. Supomos que você saiba como editar um arquivo.

Dois editores amplamente utilizados em sistemas Linux são o `vi` e o `emacs`. No Windows, o Notepad será suficiente. Também há muitos editores freeware e shareware disponíveis online, incluindo EditPlus (`www.editplus.com`), TextPad (`www.textpad.com`) e jEdit (`www.jedit.org`).

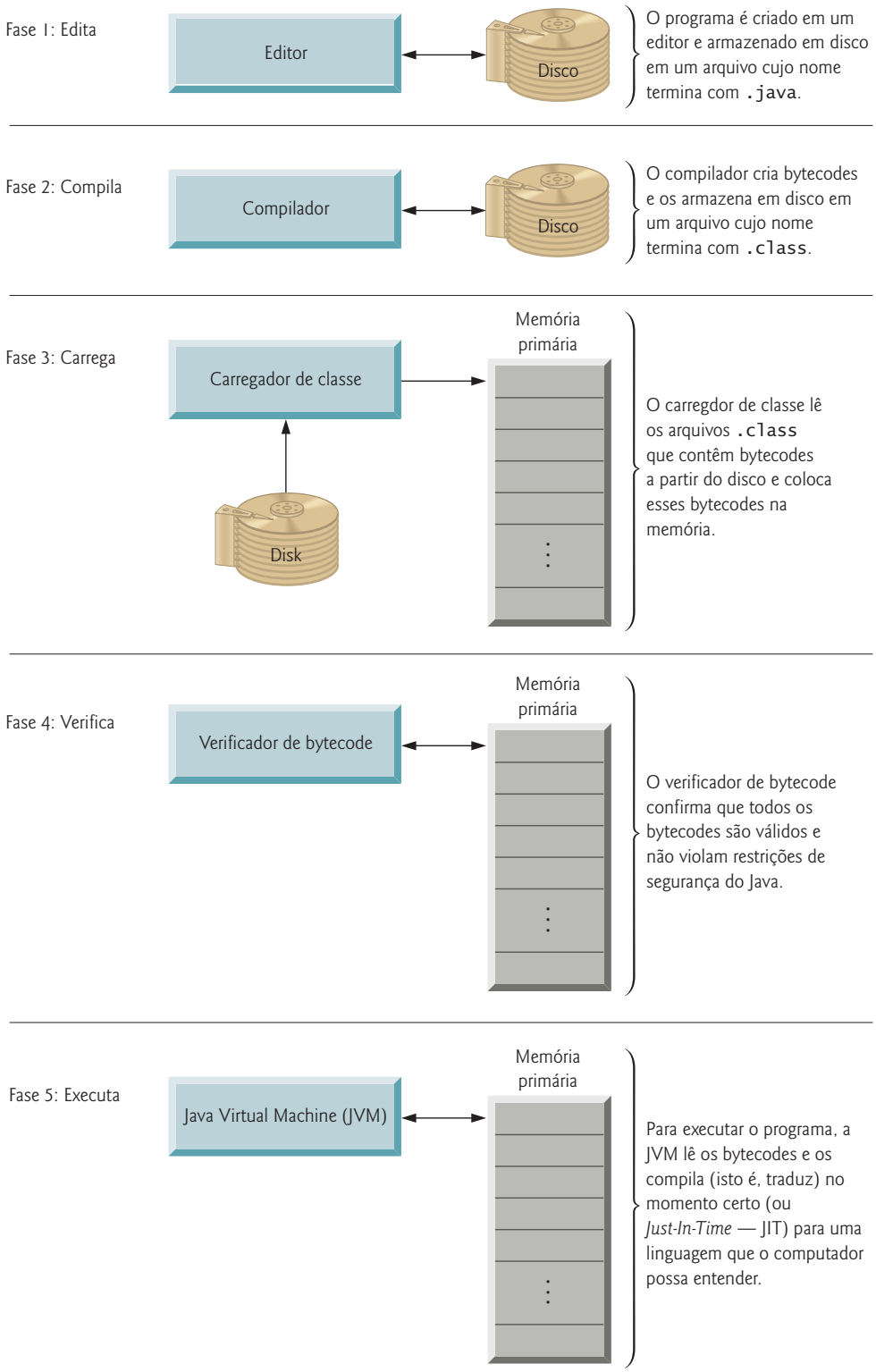


Figura 1.1 | Ambiente típico de desenvolvimento Java.

Para organizações que desenvolvem sistemas de informações substanciais, **ambientes de desenvolvimento integrado** (***Integrated Development Environments* — IDEs**) estão disponíveis a partir de muitos fornecedores de software importantes. IDEs fornecem ferramentas que suportam o processo de desenvolvimento de software, incluindo editores para escrever e editar programas e depuradores para localizar **erros de lógica** — erros que fazem os programas executar incorretamente. IDEs populares são Eclipse (`www.eclipse.org`), NetBeans (`www.netbeans.org`), JBuilder (`www.codegear.com`), JCreator (`www.jcreator.com`), BlueJ (`www.blueJ.org`) e jGRASP (`www.jgrasp.org`).

Fase 2: compilando um programa Java em bytecodes

Na Fase 2, utilize o comando (o **compilador Java**) para **compilar** um programa. Por exemplo, para compilar um programa chamado `Welcome.java`, você digitaria:

```
javac Welcome.java
```

na janela de comando do seu sistema (isto é, o **Prompt de Comando** no Windows, o **prompt de shell** no Linux ou **aplicativo Terminal** no Mac OS X). Se o programa compilar, o compilador produz um arquivo **.class** chamado `Welcome.class` que contém a versão compilada do programa.

O compilador Java converte o código-fonte Java em **bytecodes** que representam as tarefas a serem executadas na fase de execução (Fase 5). Os bytecodes são executados pela **Java Virtual Machine** (**JVM**) — uma parte do JDK e a base da plataforma Java. A **máquina virtual** (**Virtual Machine — VM**) é um aplicativo de software que simula um computador, mas oculta o sistema operacional e o hardware subjacentes dos programas que interagem com ela. Se a mesma VM for implementada nas várias plataformas de computador, os aplicativos que ela executa podem ser utilizados em todas essas plataformas. A JVM é uma das máquinas virtuais mais amplamente utilizadas. O .NET da Microsoft utiliza uma arquitetura de máquina virtual semelhante.

Ao contrário da linguagem de máquina, que é dependente do hardware específico de computador, os bytecodes são independentes de plataforma — eles não dependem de uma plataforma de hardware particular. Portanto, os bytecodes do Java são **portáveis** — sem recompilar o código-fonte, os mesmos bytecodes podem executar em qualquer plataforma contendo uma JVM que entende a versão do Java em que os bytecodes foram compilados. A JVM é invocada pelo comando `java`. Por exemplo, para executar um aplicativo Java chamado `Welcome`, você digitaria o comando:

```
java Welcome
```

em uma janela de comando para invocar a JVM, que então iniciaria os passos necessários para executar o aplicativo. Isso inicia a Fase 3.

Fase 3: carregando um programa na memória

Na Fase 3, a JVM armazena o programa na memória para executá-lo — isso é conhecido como **carregamento**. O **carregador de classe** da JVM pega os arquivos `.class` que contêm os bytecodes do programa e transfere-os para a memória primária. O carregador de classe também carrega qualquer arquivo `.class` fornecido pelo Java que seu programa utiliza. Os arquivos `.class` podem ser carregados a partir de um disco em seu sistema ou em uma rede (por exemplo, sua unidade local ou rede corporativa ou a Internet).

Fase 4: verificação de bytecode

Na Fase 4, enquanto as classes são carregadas, o **verificador de bytecode** examina seus bytecodes para assegurar que eles são válidos e não violam restrições de segurança do Java. O Java impõe uma forte segurança para certificar-se de que os programas Java que chegam pela rede não danificam os arquivos ou o sistema (como vírus e vermes de computador).

Fase 5: execução

Na Fase 5, a JVM executa os bytecodes do programa, realizando assim as ações especificadas pelo programa. Nas primeiras versões do Java, a JVM era simplesmente um interpretador para bytecodes Java. Isso fazia com que a maioria dos programas Java executasse lentamente porque a JVM interpretava e executava um bytecode por vez. Em geral, as JVMs atuais executam bytecodes utilizando uma combinação de interpretação e a chamada **compilação Just-In-Time (JIT)**. Nesse processo, a JVM analisa os bytecodes à medida que eles são interpretados, procurando **hot spots (pontos ativos)** — partes dos bytecodes que executam com frequência. Para essas partes, um **compilador Just-In-Time** (**JIT**) — conhecido como **compilador Java HotSpot** — traduz os bytecodes para a linguagem de máquina de um computador subjacente. Quando a JVM encontra novamente essas partes compiladas, o código de linguagem de máquina mais rápido é executado. Portanto, os programas Java na realidade passam por duas fases de compilação — uma em que código-fonte é traduzido em bytecodes (para a portabilidade entre JVMs em diferentes plataformas de computador) e uma segunda em que, durante a execução, os bytecodes são traduzidos em linguagem de máquina para o computador real em que o programa é executado.

Problemas que podem ocorrer em tempo de execução

Os programas podem não funcionar na primeira tentativa. Cada uma das fases anteriores pode falhar por causa de vários erros que discutiremos por todo este livro. Por exemplo, um programa executável talvez tente realizar uma operação de divisão por zero (uma operação ilegal para a aritmética de número inteiro em Java). Isso faria o programa Java imprimir uma mensagem de erro. Se isso ocorresse, você teria de retornar à fase de edição, fazer as correções necessárias e passar novamente pelas demais fases para determinar se as correções resolveram o(s) problema(s). [*Nota:* a maioria dos programas Java realiza entrada ou saída de dados. Quando dizemos que um programa exibe uma mensagem, normalmente queremos dizer que ele exibe essa mensagem na tela do computador. As mensagens e outros dados podem ser enviados para outros dispositivos de saída, como discos e impressoras ou até mesmo uma rede para transmissão para outros computadores.]

Erro comum de programação 1.1

Erros como divisão por zero ocorrem durante a execução de um programa, por isso são chamados de ***erros de runtime*** *ou* ***erros de tempo de execução****.* ***Erros de tempo de execução fatais*** *fazem com que os programas sejam imediatamente encerrados sem terem realizado seu trabalho com sucesso.* ***Erros de tempo de execução não fatais*** *permitem que os programas executem até sua conclusão, produzindo frequentemente resultados incorretos.*

1.14 Notas sobre o Java e este livro

O Java é uma poderosa linguagem de programação. Os programadores experientes às vezes sentem-se orgulhosos por criarem algum uso exótico, distorcido e complexo de uma linguagem. Essa é uma prática de programação pobre. Ela torna programas mais difíceis de ler, mais propensos a se comportar estranhamente, mais difíceis de testar e depurar e mais difíceis de adaptar em caso de alteração de requisitos. Este livro enfatiza a *clareza.* Eis a seguir a nossa primeira dica de Boa prática de programação.

Boa prática de programação 1.1

Escreva seus programas Java de uma maneira simples e direta. Isso é às vezes chamado KIS (“Keep It Simple”, ou seja, “mantenha a coisa simples”). Não “estenda” a linguagem tentando usos bizarros.

Você viu antes que o Java é uma linguagem portável e que programas escritos em Java podem executar em muitos computadores diferentes. Em geral, a *portabilidade é um objetivo ilusório.*

Dica de portabilidade 1.2

Embora seja mais fácil escrever programas portáveis em Java do que na maioria das outras linguagens de programação, diferenças entre compiladores, JVMs e computadores podem tornar a portabilidade difícil de alcançar. Simplesmente escrever programas em Java não garante portabilidade.

Dica de prevenção de erro 1.1

Sempre teste os programas Java em todos os sistemas em que você quiser executá-los, para assegurar que funcionarão corretamente para o público-alvo.

Você pode encontrar uma versão baseada na Web da documentação da Java API em `java.sun.com/javase/6/docs/api/index.html`, ou fazer o download dessa documentação no seu computador em `java.sun.com/javase/downloads/`. Para detalhes técnicos adicionais sobre muitos aspectos do desenvolvimento Java, visite `java.sun.com/reference/docs/index.html`.

Boa prática de programação 1.2

Leia a documentação da versão do Java que você está utilizando. Consulte-a frequentemente para certificar-se de que você está ciente da rica coleção de recursos Java e de que está utilizando esses recursos corretamente.

1.15 Testando um aplicativo Java

Nesta seção, você irá executar e interagir com seu primeiro aplicativo Java. Você começará executando um aplicativo ATM (Automatic Teller Machine — caixa eletrônico), que simula as transações que acontecem quando você utiliza um caixa eletrônico (por exemplo, para fazer saques e depósitos e obter saldos de sua conta bancária). Você aprenderá como construir esse aplicativo no estudo de caso opcional orientado a objetos incluído nos capítulos 12–13. A Figura 1.10 no final desta seção sugere outros aplicativos interessantes com os quais você também poderia querer fazer um test-drive. Para o objetivo desta seção, supomos que você esteja executando o Microsoft Windows.[1]

Nos próximos passos, você executará o aplicativo e realizará várias transações. Os elementos e a funcionalidade que você vê aqui são típicos daquilo que você aprenderá a programar neste livro. [*Nota:* usamos fontes para distinguir entre os aspectos que você vê em uma tela (por exemplo, o **Command Prompt**) e elementos que não estão diretamente relacionados a uma tela. Nossa convenção é enfatizar os recursos de tela como títulos e menus (por exemplo, o menu **File**) em uma fonte **Helvetica sem serifa** e seminegrito e, para enfatizar os elementos de

[1] Em `www.deitel.com/books/jhtp8/` fornecemos uma versão do Linux para esse test-drive. Também fornecemos links para vídeos que o ajudam a começar a trabalhar com vários ambientes de desenvolvimento integrados populares (IDEs), incluindo o Eclipse e NetBeans.

não tela, como nomes de arquivo ou entrada (por exemplo, `ProgramName.java`), em uma fonte `Lucida sem serifa`. Como você já percebeu, a ocorrência que define cada termo-chave no texto é especificada em **vermelho escuro**. Nas figuras desta seção, destacamos em amarelo a entrada do usuário requerida por cada passo e indicamos as partes significativas do aplicativo. Para tornar esses recursos mais visíveis, alteramos a cor do fundo das janelas Prompt de Comando para branco e a cor do primeiro plano para preto.]

1. ***Verificando sua configuração.*** Leia a Seção "Antes de você começar" do livro para confirmar se você configurou o Java corretamente no seu computador e se copiou os exemplos do livro para sua unidade de disco rígido.
2. ***Localizando o aplicativo concluído.*** Abra a janela Prompt de Comando. Mude para o diretório do aplicativo ATM digitando `cd C:\examples\ch01\ATM` e, em seguida, pressionando Enter (Figura 1.2). O comando `cd` é utilizado para mudar de diretório.

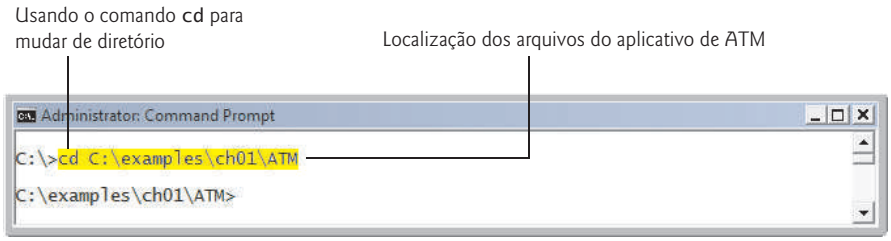


Figura 1.2 | Abrindo um Prompt de Comando e mudando de diretório.

3. ***Executando o aplicativo ATM.*** Digite o comando `java ATMCaseStudy` (Figura 1.3) e pressione *Enter*. Lembre-se de que o comando `java`, seguido pelo nome do arquivo `.class` do aplicativo (nesse caso, `ATMCaseStudy`), executa o aplicativo. Especificando a extensão `.class` ao utilizar o comando `java` resulta em um erro. [*Nota:* comandos Java diferenciam letras maiúsculas de minúsculas. É importante digitar o nome desse aplicativo com as letras A, T e M maiúsculas em "ATM", a letra C maiúscula em "Case" e um S maiúsculo em "Study". Caso contrário, o aplicativo não executará.] Se receber a mensagem de erro "`Exception in thread "main" java.lang.NoClass-DefFoundError: ATMCaseStudy`", seu sistema tem um problema de `CLASSPATH`. Consulte na Seção "Antes de você começar" do livro instruções para ajudá-lo a corrigir esse problema.

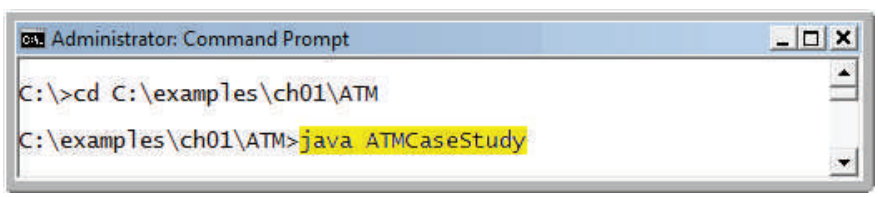


Figura 1.3 | Utilizando o comando `java` para executar o aplicativo ATM.

4. ***Inserindo o número de uma conta.*** Quando o aplicativo executa pela primeira vez, ele exibe uma saudação "`Welcome!`" e um prompt pedindo o número de uma conta. Digite `12345` no prompt "`Please enter your account number:`" (Figura 1.4) e pressione *Enter*.

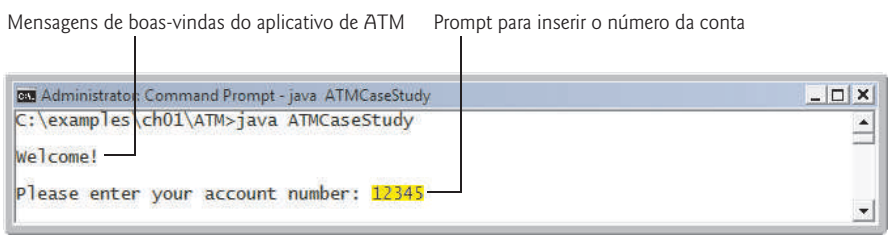


Figura 1.4 | Solicitando o número de uma conta ao usuário.

5. ***Inserindo um PIN.*** Depois que o número de uma conta válida é inserido, o aplicativo exibe o prompt "`Enter your PIN:`". Digite "`54321`" como seu PIN (Personal Identification Number) válido e pressione *Enter*. O menu principal do ATM contendo uma lista de opções será exibido (Figura 1.5).

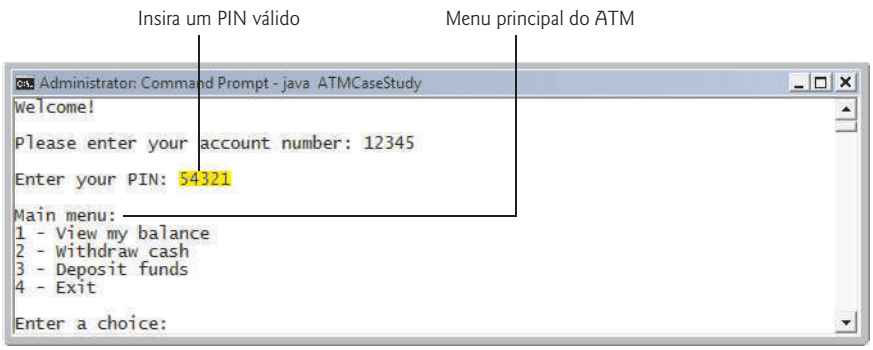


Figura 1.5 | Inserindo um número de PIN válido e exibindo o menu principal do aplicativo ATM.

6. ***Visualizando o saldo da conta.*** Selecione a opção `1`, "`View my balance`", no menu ATM (Figura 1.6). O aplicativo exibe então dois números — o `Available balance` (`$1,000.00`) e o `Total balance` (`$1,200.00`). O saldo disponível é a quantia máxima de dinheiro na sua conta que está disponível para saque em uma dada hora. Em alguns casos, certos fundos, como depósitos recentes, não estão imediatamente disponíveis para o usuário retirar, então o saldo disponível pode ser menor do que o saldo total, como é mostrado aqui. Depois que as informações do saldo da conta são mostradas, o menu principal do aplicativo é exibido novamente.

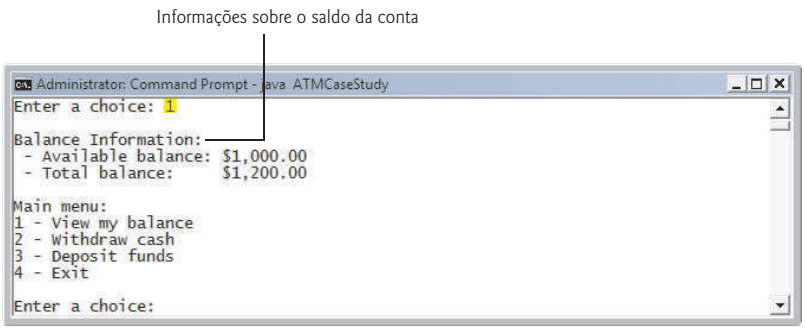


Figura 1.6 | O aplicativo ATM exibindo as informações de saldo da conta do usuário.

7. ***Retirando dinheiro da conta.*** Selecione opção `2`, "`Withdraw cash`", no menu do aplicativo. Em seguida, uma lista das quantias em dólar (Figura 1.7) é apresentada (por exemplo, `20`, `40`, `60`, `100` e `200`). Você também tem a opção de cancelar a transação e retornar ao menu principal. Retire US$ 100 selecionando a opção `4`. O aplicativo exibe "`Please take your cash now.`" e retorna ao menu principal. [*Nota:* infelizmente, esse aplicativo apenas simula o comportamento de um ATM real e, portanto, não libera realmente dinheiro.]

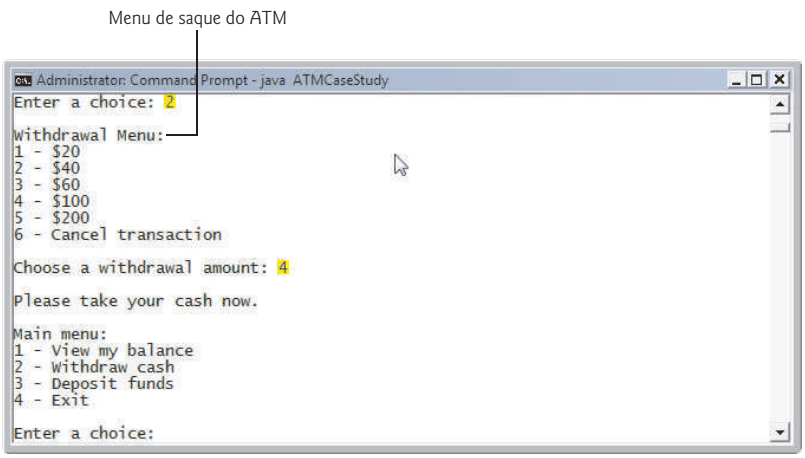


Figura 1.7 | Retirando dinheiro da conta e retornando ao menu principal.

8. ***Confirmando que as informações de conta foram atualizadas.*** A partir do menu principal, selecione novamente a opção `1` para visualizar o saldo atual da sua conta (Figura 1.8). Observe que tanto o saldo disponível como o saldo total foram atualizados para refletir a transação de retirada.

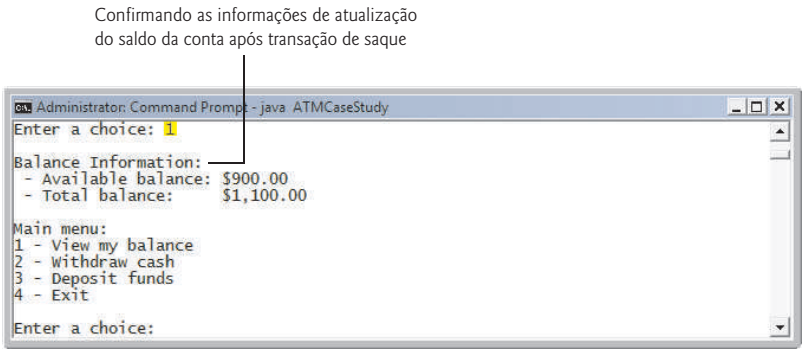


Figura 1.8 | Verificando o novo saldo.

9. ***Concluindo a transação.*** Para concluir a sessão ATM atual, selecione opção 4, "`Exit`", no menu principal (Figura 1.9). O ATM sairá do sistema e exibirá uma mensagem de despedida para o usuário. O aplicativo então retornará ao seu prompt original pedindo o número de conta do próximo usuário.

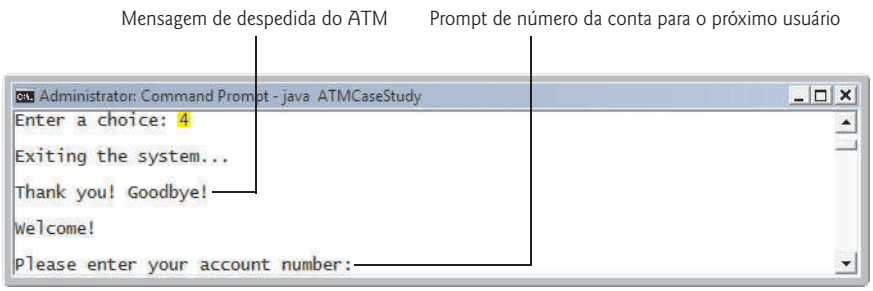


Figura 1.9 | Concluindo uma sessão de transação de um ATM.

10. ***Saindo do aplicativo ATM e fechando a janela* Command Prompt.** A maioria dos aplicativos fornece uma opção para sair e retornar ao diretório Command Prompt a partir do qual o aplicativo foi executado. Um caixa eletrônico real não fornece ao usuário a opção de desligar a máquina. Em vez disso, quando um usuário tiver concluído todas as transações desejadas e escolher a opção de menu para sair, o caixa eletrônico reinicia e exibe um prompt solicitando o número de conta do próximo usuário. Como a Figura 1.9 ilustra, o aplicativo ATM aqui se comporta de maneira semelhante. Escolher a opção de menu de sair encerra somente a sessão de ATM do usuário atual, não o aplicativo ATM inteiro. Para sair realmente do aplicativo ATM, clique no botão close (x) no canto superior direito da janela Command Prompt. Fechar a janela faz com que o aplicativo em execução termine.

Aplicativos sugeridos para um test-drive neste livro

A Figura 1.10 lista algumas das centenas de aplicativos encontrados nos exemplos e exercícios do livro. Esses programas introduzem muitos dos recursos poderosos e divertidos do Java. Execute esses programas para ver um número maior dos tipos de aplicativos que você aprenderá a construir neste livro. A pasta de exemplos deste capítulo contém todos os arquivos necessários para executar cada aplicativo. Simplesmente digite os comandos listados na Figura 1.10 em uma janela Prompt de Comando.

Nome do aplicativo	Localização do capítulo	Comandos a serem executados
Tic-Tac-Toe (jogo da velha)	Capítulo 8	`cd C:\examples\ch01\Tic-Tac-Toe java TicTacToeTest`
Jogo de adivinhação	Capítulo 14	`cd C:\examples\ch01\GuessGame java GuessGame`
Animador de logotipo	Capítulo 24	`cd C:\examples\ch01\LogoAnimator java LogoAnimator`
Bola que rebate	Capítulo 26	`cd C:\examples\ch01\BouncingBall java BouncingBall`

Figura 1.10 | Alguns dos aplicativos sugeridos para um test-drive neste livro.

1.16 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML

Agora vamos começar nossa primeira introdução à orientação a objetos, uma maneira natural de pensar o mundo e escrever programas de computador. Os capítulos 12–13 apresentam uma introdução cuidadosamente compassada ao design orientado a objetos. Nosso objetivo aqui é ajudá-lo a desenvolver uma maneira de pensar orientada a objetos e o introduzir a **Unified Modeling Language™ (UML™)** — uma linguagem gráfica que permite que as pessoas projetem sistemas de software para utilizar uma notação padrão da indústria para representá-las.

Nesta única seção necessária, introduzimos a terminologia e os conceitos orientados a objetos. As seções opcionais nos capítulos 12–13 apresentam um design orientado a objetos e a implementação do software de um caixa eletrônico simples. As seções 12.2–12.7 do estudo de caso de engenharia de software:

- analisam um típico documento de requisitos que descreve um sistema de software (o ATM) a ser construído;
- determinam os objetos necessários para implementar o sistema;
- determinam os atributos que os objetos terão;
- determinam os comportamentos que esses objetos exibirão;
- especificam como os objetos interagem para atender os requisitos de sistema.

As seções 13.2–13.3 modificam e aprimoram o design apresentado nas seções 12.2–12.7. A Seção 13.4 contém uma implementação funcional do Java completa do sistema de software do ATM orientado a objetos.

Você experimentará uma introdução concisa e sólida ao design orientado a objetos com a UML. Além disso, você aperfeiçoará suas habilidades de leitura de código passeando pela completa implementação Java do ATM cuidadosamente escrita e bem documentada na Seção 13.4.

Conceitos básicos da tecnologia de objetos

Iniciamos nossa introdução à orientação a objetos com uma terminologia-chave. Onde quer que você olhe no mundo real, você vê **objetos** — pessoas, animais, plantas, carros, aviões, edifícios, computadores e assim por diante. Os humanos pensam em termos de objetos. Telefones, residências, sinais de trânsito, micro-ondas e refrigeradores são alguns outros objetos. Programas de computador, como os programas Java que você verá neste livro e aqueles que você escreverá, são compostos de muitos objetos de software interativos.

Às vezes dividimos os objetos em duas categorias: animados e inanimados. Os animados são, em certo sentido, objetos "vivos" — eles se movem e fazem coisas. Por outro lado, os objetos inanimados não se movem por conta própria. Objetos de ambos os tipos, porém, têm algumas coisas em comum. Todos eles têm **atributos** (por exemplo, tamanho, forma, cor e peso) e todos exibem **comportamentos** (por exemplo, uma bola rola, rebate, infla e murcha; o bebê chora, dorme, engatinha, anda e pisca; um carro acelera, freia e desvia; uma toalha absorve água). Estudaremos os tipos de atributos e comportamentos dos objetos de software.

Os humanos aprendem sobre os objetos existentes estudando seus atributos e observando seus comportamentos. Diferentes objetos podem ter atributos semelhantes e podem exibir comportamentos semelhantes. É possível fazer comparações, por exemplo, entre bebês e adultos, e entre humanos e chimpanzés.

O **design orientado a objetos (*Object-Oriented Design* — OOD)** modela software em termos semelhantes àqueles que as pessoas utilizam para descrever objetos do mundo real. Ele tira proveito de relacionamentos de **classe**, em que os objetos de certa classe, como uma classe de veículos, têm as mesmas características — carros, caminhões, patins têm muito em comum. O OOD também tira proveito dos relacionamentos de **herança**, dos quais as classes de objetos novas são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessas mesmas classes. Um objeto de classe "conversível" certamente tem as características da classe mais geral "automóvel", mas, mais especificamente, seu capô sobe e desce.

O design orientado a objetos fornece uma maneira natural e intuitiva de visualizar o processo de design de software — a saber, modelar objetos por seus atributos e comportamentos assim como descrevemos objetos do mundo real. A OOD também modela a comunicação entre objetos. Assim como as pessoas trocam mensagens entre si (por exemplo, um sargento manda um soldado permanecer em atenção), os objetos também se comunicam via mensagens. Um objeto conta bancária pode receber uma mensagem para reduzir seu saldo em certa quantia porque o cliente retirou essa quantia em dinheiro.

O OOD **encapsula** (isto é, empacota) atributos e **operações** (comportamentos) em objetos — os atributos e operações de um objeto estão intimamente ligados. Os objetos têm a propriedade de **ocultamento de informações**. Isso significa que podem saber como se comunicar com outros por meio de **interfaces** bem definidas, mas normalmente eles não têm permissão para saber como os outros objetos são implementados — os detalhes de implementação são ocultados dentro dos próprios objetos. Na verdade podemos dirigir um carro, por exemplo, sem conhecer os detalhes de como motores, transmissões, freios e sistemas de escapamento funcionam internamente — contanto que saibamos utilizar o acelerador, o freio, a roda e assim por diante. O ocultamento de informações, como veremos, é crucial à boa engenharia de software.

Linguagens como Java são linguagens **orientadas a objeto**. A programação nessa linguagem, chamada **programação orientada a objetos (*Object-Oriented Programming* — OOP)**, permite-lhe implementar um design orientado a objetos como um sistema funcional. Linguagens como o C, por outro lado, são **procedurais**, então a programação tende a ser **orientada para a ação**. No C, a unidade de programação é a **função**. Grupos de ações que realizam alguma tarefa comum são reunidos em funções e as funções são agrupadas para formar programas. No Java, a unidade de programação é a classe a partir da qual os objetos por fim são **instanciados** (criados). Classes Java contêm **métodos** (que implementam operações e são semelhantes a funções na linguagem C) bem como **campos** (que implementam atributos).

Programadores em Java concentram-se na criação de classes. Cada classe contém campos e o conjunto de métodos que manipulam os campos e fornecem serviços aos **clientes** (isto é, outras classes que utilizam a classe). O programador utiliza classes existentes como blocos de construção para construir novas classes.

As classes estão para os objetos assim como as plantas arquitetônicas estão para as casas. Assim como podemos construir muitas casas a partir de uma planta, podemos instanciar (criar) muitos objetos a partir de uma classe. Você não pode fazer refeições na cozinha de uma planta; isso só é possível em uma cozinha real.

As classes podem ter relacionamentos com outras classes. Por exemplo, em um design orientado a objetos de um banco, a classe "caixa de banco" precisa se relacionar com a classe "cliente", a classe "gaveta de dinheiro", a classe "cofre" etc. Esses relacionamentos são chamados de **associações**.

Empacotar software como classes possibilita que os sistemas de software futuros **reutilizem** as classes. Grupos de classes relacionadas são frequentemente empacotados como **componentes** reutilizáveis. Assim como corretores de imóveis costumam dizer que os três fatores mais importantes que afetam o preço dos imóveis são "localização, localização e localização", as pessoas na comunidade de software costumam dizer que os três fatores mais importantes que afetam o futuro do desenvolvimento de software são "reutilização, reutilização e reutilização". A reutilização de classes existentes ao construir novas classes e programas economiza tempo e esforço. A reutilização também ajuda-lhe a construir sistemas mais confiáveis e eficientes, porque classes e componentes existentes costumam passar por extensos testes, depuração e ajuste de desempenho.

Com a tecnologia de objetos, você pode construir grande parte do software necessário combinando classes, exatamente como fabricantes de automóveis combinam partes intercambiáveis. Cada nova classe que você criar terá o potencial de se tornar um "ativo de software" que você e outros programadores podem utilizar para acelerar e aprimorar a qualidade de seus esforços futuros no desenvolvimento de software.

Introdução à análise e design orientado a objetos (Object-Oriented Analysis and Design — OOAD)

Logo você estará escrevendo programas em Java. Como criar o código para seus programas? Talvez, como muitos programadores, você simplesmente ligará seu computador e começará a digitar. Essa abordagem pode funcionar para pequenos programas (como os apresentados nos primeiros capítulos do livro), mas e se você fosse contratado para criar um sistema de software para controlar milhares de caixas eletrônicos para um banco importante? Ou se você fosse trabalhar em uma equipe de 1.000 desenvolvedores de software para construir o próximo sistema de controle de tráfego aéreo dos Estados Unidos? Para projetos tão grandes e complexos, você não sentaria e simplesmente começaria a escrever programas.

Para criar as melhores soluções, você deve seguir um processo detalhado para **analisar** os **requisitos** do seu projeto (isto é, determinar *o que* o sistema deve fazer) e desenvolver um **design** que atenda esses requisitos (isto é, decidir *como* o sistema deve fazê-lo). Idealmente, você passaria por esse processo e revisaria cuidadosamente o projeto (ou teria seu projeto revisado por outros profissionais de software) antes de escrever qualquer código. Se esse processo envolve analisar e projetar o sistema de um ponto de vista orientado a objetos, ele é chamado de **processo de análise e design orientado a objetos (*Object-Oriented Analysis and Design* — OOAD)**. Programadores experientes sabem que a análise e o design podem poupar muitas horas, ajudando a evitar uma abordagem de desenvolvimento de sistemas precariamente planejada que teria de ser abandonada no meio do caminho da implementação, possivelmente desperdiçando tempo, dinheiro e esforço consideráveis.

O OOAD é o termo genérico para o processo de análise de um problema e desenvolvimento de uma abordagem para resolvê-lo. Pequenos problemas, como os discutidos nesses primeiros poucos capítulos, não exigem um processo exaustivo de OOAD. Talvez seja suficiente escrever o **pseudocódigo** antes de começarmos a escreve o código Java. O pseudocódigo é um meio informal de expressar a lógica do programa. Na realidade, o pseudocódigo não é uma linguagem de programação, mas pode ser utilizado como um esboço de orientação ao escrever o código. Introduzimos o pseudocódigo no Capítulo 4.

Uma vez que os problemas e os grupos de pessoas que os resolvem aumentam em tamanho, os métodos OOAD tornam-se mais adequados do que o pseudocódigo. Idealmente, um grupo deve estabelecer um acordo comum sobre um processo rigorosamente definido para resolver seu problema e sobre uma maneira uniforme de comunicar os resultados desse processo para os outros. Embora existam muitos processos OOAD diferentes, uma única linguagem gráfica para comunicar os resultados de *qualquer* processo OOAD veio a ser amplamente utilizada. Essa linguagem, conhecida como Unified Modeling Language (UML), foi desenvolvida em meados da década de 1990 sob a direção inicial de três metodologistas de software — Grady Booch, James Rumbaugh e Ivar Jacobson.

História da UML

Na década de 1980, um número crescente de organizações começou a utilizar OOP para construir seus aplicativos e desenvolveu-se a necessidade de um processo OOAD padrão. Muitos metodologistas — incluindo Booch, Rumbaugh e Jacobson — produziram e promoveram individualmente processos separados para satisfazer essa necessidade. Cada processo tinha sua própria notação ou "linguagem" (na forma de diagramas gráficos) para transportar os resultados de análise e design.

Por volta do início da década de 1990, diferentes organizações e até divisões dentro de uma mesma organização estavam utilizando seus próprios processos e notações. Ao mesmo tempo, essas organizações também queriam utilizar ferramentas de software que suportassem seus processos particulares. Os fornecedores de software acharam difícil e custoso fornecer ferramentas para tantos processos. Uma notação padrão e processos padrão eram necessários.

Em 1994, James Rumbaugh associou-se a Grady Booch na Rational Software Corporation (agora uma divisão da IBM) e os dois começaram a trabalhar para unificar seus populares processos. Eles logo se associaram a Ivar Jacobson. Em 1996, o grupo liberou as primeiras versões da UML para a comunidade de engenharia de software e feedback solicitado. Mais ou menos na mesma época, uma organização conhecida como **Object Management Group™ (OMG™)** solicitou ideias e sugestões para uma linguagem de modelagem comum. O OMG (`www.omg.org`) é uma organização sem fins lucrativos que promove a padronização de tecnologias orientadas a objetos publicando diretrizes e especificações, como a UML. Várias corporações — entre elas HP, IBM, Microsoft, Oracle e Rational Software — já haviam reconhecido a necessidade de uma linguagem de modelagem comum. Em resposta à solicitação de propostas do OMG, essas empresas formaram o **UML Partners** — o consórcio que desenvolveu a UML versão 1.1 e a sugeriu para o OMG. Este aceitou a proposta e, em 1997, assumiu a responsabilidade pela manutenção e revisão constantes da UML. Apresentamos a terminologia e notação da UML 2 por todo este livro.

O que é a UML?

A UML é agora o esquema de representação gráfica mais amplamente utilizado para modelar sistemas orientados a objetos. Ela de fato unificou os vários esquemas de notação populares. Aqueles que projetam sistemas utilizam a linguagem (na forma de diagramas) para modelar seus sistemas.

Um recurso atraente da UML é sua flexibilidade. Ela é **extensível** (isto é, capaz de ser aprimorada com novos recursos) e é independente de qualquer processo OOAD particular. Os modeladores de UML são livres para utilizar vários processos para projetar sistemas, mas agora todos os desenvolvedores podem expressar seus designs com um conjunto padrão de notações gráficas.

A UML é uma linguagem gráfica complexa, rica em recursos. Apresentamos nossos primeiros diagramas UML nos capítulo 3–4. No nosso estudo de caso (opcional) de engenharia de software do ATM nos capítulos 12–13, apresentamos um subconjunto simples e conciso dos recursos da UML. Então utilizamos esse subconjunto para guiá-lo pela primeira experiência de design com a UML destinada a programadores iniciantes em tecnologia orientada a objetos em cursos de programação do primeiro ou segundo semestre. Para informações adicionais sobre a UML, visite nosso UML Resource Center em `www.deitel.com/UML/`.

Seção 1.16 Exercícios de autorrevisão

1.1 Liste três exemplos de objetos do mundo real que não mencionamos. Para cada objeto, liste vários atributos e comportamentos.

1.2 Pseudocódigo é _____.
a) outro termo para OOAD
b) uma linguagem de programação utilizada para exibir diagramas de UML
c) uma maneira informal de expressar a lógica do programa
d) um esquema de representação gráfica para modelar sistemas orientados a objetos

1.3 A UML é utilizada principalmente para _____.
a) testar sistemas orientados a objetos
b) projetar sistemas orientados a objetos
c) implementar sistemas orientados a objetos
d) a e b são alternativas corretas

Respostas da Seção 1.16 Exercícios de autorrevisão

1.1 [*Nota:* as respostas podem variar.] a) Os atributos de uma televisão incluem o tamanho da tela, o número de cores que ela pode exibir, o canal atual e o volume atual. Uma televisão liga e desliga, muda de canal, exibe vídeo e reproduz sons. b) Os atributos de uma cafeteira incluem o volume máximo de água que ela pode conter, o tempo necessário para preparar um bule de café e a temperatura da chapa de aquecimento sob o bule de café. Uma cafeteira liga e desliga, prepara o café e o mantém aquecido. c) Os atributos de uma tartaruga incluem sua idade, o tamanho do casco e o peso. Uma tartaruga caminha, protege-se dentro seu casco, sai de seu casco e alimenta-se de vegetais.

1.2 c.

1.3 b.

1.17 Web 2.0

A Web literalmente explodiu entre a metade e o final dos anos 1990, mas o fracasso econômico do "ponto com" trouxe tempos difíceis no início dos anos 2000. O ressurgimento que começou mais ou menos em 2004 foi chamado **Web 2.0**. A primeira conferência Web 2.0 aconteceu em 2004. Um ano depois, o termo "Web 2.0" acumulou aproximadamente 10 milhões de hits no sistema de pesquisa **Google**, aumentado para 60 milhões um ano mais tarde. O Google é amplamente considerado como a empresa símbolo da Web 2.0. Outras são: Craigslist (listas de classificados gratuitos), Flickr (compartilhamento de fotos), del.icio.us (social bookmarking), YouTube (compartilhamento de vídeos),

MySpace e FaceBook (serviços de rede social), Salesforce (software profissional oferecido como serviços online), Second Life (um mundo virtual), Skype (telefonia via Internet) e Wikipedia (uma enciclopédia online gratuita).

Na Deitel & Associates, lançamos nossa **Internet Business Initiative** baseada na Web 2.0 em 2005. Pesquisamos as tecnologias-chave da Web 2.0 e a utilizamos para construir negócios na Internet. Compartilhamos nossas pesquisas na forma de Resource Centers em `www.deitel.com/resourcecenters.html`. A cada semana, anunciamos os Resource Centers mais recentes no nosso boletim informativo, *Deitel® Buzz Online* (`www.deitel.com/newsletter/subscribe.html`). Cada um lista vários links para principalmente conteúdo e softwares gratuitos na Internet.

Incluímos neste livro um tratamento substancial dos **serviços Web** (Capítulo 31) e introduzimos a nova metodologia de desenvolvimento de aplicativos dos **mashups** nos quais você pode desenvolver rapidamente aplicativos poderosos e intrigantes combinando serviços Web complementares e outras formas de **feeds** de informações de duas ou mais organizações. Um mashup popular é `www.housingmaps.com`, que combina as listagens de classificados de imóveis fornecidas por `www.craigslist.org` com as capacidades de mapeamento do **Google Maps** para oferecer mapas que mostram as localizações de apartamentos para locação em uma dada área.

O **Ajax** é uma das principais tecnologias da Web 2.0. Embora o uso do termo tenha explodido em 2005, ele apenas identifica um grupo de tecnologias e técnicas de programação em uso desde o final dos anos 1990. O Ajax ajuda aplicativos baseados na Internet a ter um desempenho comparável ao dos aplicativos desktop — uma tarefa difícil, dado que esses aplicativos sofrem atrasos de transmissão à medida que os dados são transmitidos entre seu computador e outros computadores na Internet. Utilizando o Ajax, aplicativos como o Google Maps alcançaram um excelente desempenho e se aproximaram da aparência e do funcionamento dos aplicativos desktop. Embora não discutamos a programação Ajax "bruta" (que é bastante complexa) neste texto, mostramos no Capítulo 30 como construir aplicativos compatíveis com o Ajax utilizando componentes JavaServer Faces (JSF) compatíveis com o Ajax.

Blogs são sites Web (atualmente mais de 100 milhões) que se parecem com diários online, com as entradas mais recentes aparecendo primeiro. Bloggers postam rapidamente suas opiniões sobre notícias, lançamentos de produtos, candidatos políticos, questões controversas e praticamente tudo mais. A coleção de todos os blogs e a comunidade de criação de blogs é chamada **blogosfera** e está se tornando cada vez mais influente. **Technorati** e **Google Blog Search** são dois dos sistemas de pesquisa de blog populares.

RSS feeds permitem que sites incluam informações aos assinantes. Um uso comum de RSS feeds é disponibilizar as postagens mais recentes nos blogs para assinantes. Os fluxos de informação RSS na Internet estão crescendo exponencialmente.

A **Web 3.0** é outro nome para a próxima geração da Web, também chamada **Web semântica**. A Web 1.0 era quase puramente baseada em HTML. A Web 2.0 cada vez mais utiliza a XML, especialmente em tecnologias como RSS feeds e serviços Web. A Web 3.0 utilizará intensamente a XML, criando "uma teia de significados". Se você é um estudante procurando um artigo de pesquisa ou tema de tese, ou um empresário procurando oportunidades de negócios, verifique nosso Web 3.0 Resource Center.

Para acompanhar os últimos avanços na Web 2.0, leia `www.techcrunch.com` e `www.slashdot.org` e verifique a lista crescente dos Resource Centers relacionados à Internet e Web em `www.deitel.com/resourcecenters.html`.

1.18 Tecnologias de software

Nesta seção, discutimos alguns jargões da área de engenharia de software que você ouvirá na comunidade de desenvolvimento de softwares. Criamos Resource Centers sobre a maioria desses tópicos, com mais a caminho.

O **Agile Software Development** é um grupo de metodologias que tenta implementar softwares rapidamente utilizando menos recursos do que as metodologias anteriores. Verifique a Agile Alliance (`www.agilealliance.org`) e o Agile Manifesto (`www.agilemanifesto.org`).

A **refatoração** envolve a reformulação do código para torná-lo mais claro e mais fácil de manter e, ao mesmo tempo, preservar sua funcionalidade. Ela é amplamente empregada com metodologias de desenvolvimento ágeis. Há muitas ferramentas de refatoração disponíveis para criar as partes principais da reformulação automaticamente.

Padrões de projeto (**design patterns**) são arquiteturas testadas para construir softwares orientados a objetos flexíveis e que podem ser mantidos (ver o Apêndice Q). O campo dos padrões de projeto tenta enumerar aqueles padrões recorrentes, encorajando os designers de software a reutilizá-los para desenvolver softwares de melhor qualidade em menos tempo, dinheiro e esforço.

A **programação de jogos** é uma carreira interessante e desafiadora para desenvolvedores de software. O negócio de jogos de computador já é maior do que o negócio de filmes inéditos. Cursos universitários e até disciplinas agora dedicam-se às técnicas sofisticadas utilizadas na programação de jogos. Verifique nossos Resource Centers de projetos de programação e programação de jogos.

Softwares de código-fonte aberto são um estilo de desenvolvimento de softwares que se diferencia do desenvolvimento proprietário que dominou os primeiros anos dos softwares. Com o desenvolvimento de softwares de código-fonte aberto, pessoas e empresas contribuem com seus esforços para desenvolver, manter e evoluir os softwares em troca do direito de utilizar esses softwares para objetivos próprios, em geral gratuitamente. Software de código-fonte aberto normalmente é realizado por um grupo de pessoas bem maior do que softwares proprietários, assim erros muitas vezes são removidos mais rapidamente. Código-fonte aberto também estimula mais inovações. A Sun transformou sua implementação do kit de desenvolvimento Java em código-fonte aberto. Algumas organizações sobre as quais você ouvirá muito na comunidade de código-fonte aberto são a Eclipse Foundation (o IDE Eclipse é popular para o desenvolvimento de softwares Java), a Mozilla Foundation (criadores do navegador Web Firefox), a Apache Software Foundation (criadores do servidor Web Apache) e SourceForge (que fornece as ferramentas para gerenciar projetos de código-fonte aberto e atualmente tem mais de 100.000 projetos de código-fonte aberto em desenvolvimento).

O **Linux** é um sistema operacional de código-fonte aberto e um dos maiores sucessos do movimento de código-fonte aberto. O **MySQL** é um sistema de gerenciamento de bancos de dados de código-fonte aberto. O **PHP** é a linguagem Internet para criação de scripts de código-fonte aberto do lado do servidor mais popular para desenvolver aplicativos baseados na Internet. **LAMP** é um acrônimo para o conjunto de tecnologias de código-fonte aberto que muitos desenvolvedores utilizavam para construir aplicativos Web — ele significa Linux, Apache, MySQL e PHP (ou Perl ou Python — duas outras linguagens utilizadas para objetivos semelhantes).

O **Ruby on Rails** combina a linguagem de criação de scripts Ruby com a estrutura de aplicativo Web do Rails desenvolvida pela empresa 37Signals. Seu livro, *Getting Real*, é uma leitura obrigatória para os atuais desenvolvedores de aplicativo Web; leia-o gratuitamente em `gettingreal.37signals.com/toc.php`. Muitos desenvolvedores Ruby on Rails informaram ganhos de produtividade significativos em relação ao uso de outras linguagens ao desenvolver aplicativos Web que utilizam intensamente banco de dados.

Softwares geralmente são vistos como um produto; a maioria dos softwares ainda é oferecida dessa forma. Se quiser executar um aplicativo, você compra um pacote de software de um fornecedor de softwares. Você então instala esse software no computador e executa-o conforme necessário. À medida que novas versões do software aparecem, você faz uma atualização do seu software, frequentemente com custos significativos. Esse processo pode tornar-se difícil para organizações com dezenas de milhares de sistemas que devem ser mantidos em um conjunto diverso de equipamentos de informática. Com o **Software As A Service (SAAS)** os softwares executam em servidores em outros locais na Internet. Quando esse servidor é atualizado, todos os clientes no mundo inteiro veem as novas capacidades; nenhuma instalação local é necessária. Você acessa o serviço por um navegador. Navegadores são bem portáteis, portanto você pode executar os mesmos aplicativos em diferentes tipos de computadores a partir de qualquer lugar no mundo. Salesforce.com, Google e o Office Live e Windows Live da Microsoft oferecem SAAS.

1.19 Conclusão

Este capítulo introduziu os conceitos básicos de hardware, software e comunicação e os conceitos básicos da tecnologia orientada a objetos, incluindo classes, objetos, atributos, comportamentos, encapsulamento e herança. Discutimos os diferentes tipos de linguagens de programação e quais linguagens específicas são mais amplamente utilizadas. Você aprendeu os passos para criar e executar um aplicativo Java utilizando o JDK 6 da Sun. O capítulo explorou a história da Internet e da World Wide Web até o fenômeno Web 2.0, e o papel do Java para desenvolver aplicativos cliente/servidor distribuídos para a Internet e Web. Você também aprendeu a história e o propósito da UML — a linguagem gráfica padrão da indústria para modelar sistemas de software orientados a objetos. Você fez um test-drive de um ou mais aplicativos Java de exemplo semelhantes àqueles que você aprenderá a programar neste livro. Discutimos algumas importantes tecnologias de software recentes.

No Capítulo 2, você criará seus primeiros aplicativos Java. Você verá exemplos que mostram como os programas exibem mensagens e obtêm informações do usuário para processamento. Você utilizará tipos de dados primitivos e operadores aritméticos do Java. Você utilizará operadores relacionais e de igualdade do Java para escrever instruções simples de tomada de decisão. Analisaremos e explicaremos cada exemplo para ajudar a facilitar a introdução à programação de Java.

1.20 Recursos da Web

Esta seção lista muitos sites úteis à medida que você aprende o Java. Os sites incluem recursos do Java, ferramentas de desenvolvimento em Java para alunos e profissionais e, além disso, nossos próprios sites da Web em que você pode encontrar downloads e recursos associados com este livro.

Deitel & Associates Websites

`www.deitel.com`
Contém atualizações, correções e recursos adicionais para todas as publicações Deitel.

`www.deitel.com/newsletter/subscribe.html`
Assine e receba gratuitamente por e-mail o boletim informativo *Deitel® Buzz Online* para acompanhar o programa de publicação da Deitel & Associates, incluindo atualizações e erratas para este livro.

`www.deitel.com/books/jhtp8/` e `www.prenhall.com/deitel_br`
O primeio é o site da Deitel & Associates (em inglês) e o segundo é o site da Prentice Hall (em português) para este livro. Aqui você encontrará links para os exemplos do livro e outros recursos.

Centros de Recursos Relacionados a Java da Deitel & Associates

`www.deitel.com/Java/`
Nosso Java Resource Center focaliza uma quantidade enorme de conteúdo gratuito sobre o Java disponível online. Inicie sua pesquisa aqui para ferramentas, ambientes de desenvolvimento integrados (IDEs), recursos, downloads, tutoriais, documentação, livros, e-books, periódicos, artigos, blogs e mais que irão ajudá-lo a desenvolver aplicativos Java.

`www.deitel.com/JavaSE6Mustang/`
Nosso Java SE 6 (Mustang) Resource Center é seu guia para a versão mais recente do Java. O site inclui os melhores recursos que encontramos on-line para ajudá-lo a começar a trabalhar com o desenvolvimento Java SE 6.

`www.deitel.com/CodeSearchEngines/`
Nossos Code Search Engines e Code Sites Resource Center incluem o uso de recursos que os desenvolvedores utilizam para localizar código-fonte online.

`www.deitel.com/ProgrammingProjects/`
Nosso Programming Projects Resource Center é seu guia online para excelentes ideias relacionadas a projetos de programação de alunos.

Editores e ambientes de desenvolvimento integrado

`www.eclipse.org`
O ambiente de desenvolvimento Eclipse pode ser utilizado para desenvolver código em qualquer linguagem de programação. Você pode fazer o download do ambiente e vários plug-ins Java para desenvolver seus programas Java.

`www.netbeans.org`
O IDE do NetBeans é uma das ferramentas de desenvolvimento Java livremente distribuídas mais amplamente utilizada.

`www.blueJ.org`
BlueJ é uma ferramenta gratuita projetada a fim de ajudar a ensinar o Java orientado a objetos para novos programadores.

`www.jgrasp.org`
jGRASP é um ambiente de desenvolvimento integrado "mais leve" desenvolvido na Auburn University. A ferramenta também exibe representações visuais dos programas Java para auxiliar a compreensão.

`www.jcreator.com`
JCreator é um IDE do Java popular. O JCreator Lite Edition está disponível como um download gratuito. Também há uma versão de avaliação de 30 dias do Creator Pro Edition.

`www.textpad.com`
TextPad é um editor que também permite compilar e executar seus programas Java.

`www.editplus.com`
EditPlus é um editor que pode ser configurado para compilar e executar seus programas Java.

`www.jedit.org`
jEdit é um editor escrito em Java e pode ser utilizado com muitas linguagens de programação.

Resumo

Seção 1.1 Introdução

- O Java tornou-se a linguagem preferida para implementar aplicativos baseados na Internet e software para dispositivos que se comunicam em uma rede.
- O Java Enterprise Edition (Java EE) é adequado para desenvolver aplicativos em rede distribuídos e em grande escala e também aplicativos baseados na Web.
- A Java Micro Edition (Java ME) é voltada para o desenvolvimento de aplicativos de pequenos dispositivos com limitações de memória, como telefones celulares, pagers e PDAs.

Seção 1.2 Computadores: hardware e software

- Um computador é um dispositivo que pode realizar cálculos e tomar decisões lógicas fenomenalmente mais rápido do que os seres humanos.
- Os computadores processam dados sob o controle de conjuntos de instruções chamados programas de computador. Os programas guiam os computadores ao longo das ações especificadas por pessoas chamadas *programadores*.
- Um computador consiste em vários dispositivos conhecidos como hardware. Os programas que executam em um computador são chamados de software.

Seção 1.3 Organização do computador

- Praticamente cada computador pode ser visto como sendo dividido em seis unidades lógicas ou seções.
- A unidade de entrada obtém informações a partir dos dispositivos de entrada e disponibiliza essas informações a outras unidades para que possam ser processadas.
- A unidade de saída pega as informações que o computador processou e as coloca em vários dispositivos de saída para tornar as informações disponíveis para serem utilizadas fora do computador.
- A unidade de memória é a seção de armazenamento de relativamente baixa capacidade e acesso rápido do computador. Ela retém as informações que foram inseridas por meio da unidade de entrada, tornando-as imediatamente disponíveis para processamento quando necessário. Ela também retém informações processadas até que elas possam ser colocadas em dispositivos de saída pela unidade de saída.
- A unidade aritmética e lógica (ALU) é responsável por realizar cálculos (tais como adição, subtração, multiplicação e divisão) e tomar decisões.
- A unidade central de processamento (CPU) coordena e supervisiona a operação das outras seções. A CPU diz à unidade de entrada quando as informações devem ser lidas e transferidas para a unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas em cálculos e instrui a unidade de saída sobre quando enviar as informações da unidade de memória para certos dispositivos de saída.
- Os multiprocessadores têm múltiplas CPUs e, consequentemente, podem realizar várias operações simultaneamente.
- Um processador de múltiplos núcleos (ou multi-core) implementa o multiprocessamento em um único chip de circuito integrado — por exemplo um processador de dois núcleos (ou dual-core) tem duas CPUs e um processador de quatro núcleos (ou quad-core) tem quatro.
- A unidade de armazenamento secundário é a seção de armazenamento de alta capacidade e longo prazo do computador. Programas ou dados não ativamente usados por outras unidades normalmente são colocados em dispositivos de armazenamento secundário até que sejam necessários.

Seção 1.4 Primeiros sistemas operacionais

- Os primeiros computadores podiam realizar apenas um **trabalho** ou **tarefa** por vez.
- Os sistemas operacionais foram desenvolvidos para tornar o uso dos computadores mais conveniente.
- A multiprogramação envolve a operação simultânea de muitos trabalhos.
- Com o compartilhamento de tempo, o computador executa uma pequena parte do trabalho de um usuário, e, em seguida, atende o próximo usuário, talvez fornecendo serviço para cada usuário várias vezes por segundo.

Seção 1.5 Computação pessoal distribuída e computação cliente/servidor

- Em 1977, a Apple Computer popularizou a computação pessoal.
- Em 1981, a IBM, o maior fornecedor de computadores no mundo, introduziu o computador pessoal IBM, que rapidamente legitimou a computação pessoal nos negócios, indústrias e governos.
- Na computação distribuída, em vez de ser executada apenas em um computador central, a computação é distribuída ao longo de redes até os sites onde o trabalho da organização é realizado.
- Os servidores armazenam dados que podem ser utilizados por computadores clientes distribuídos por toda a rede, daí o termo computação cliente/servidor.
- O Java tornou-se amplamente utilizado para escrever software de rede de computador e aplicativos distribuídos cliente/servidor.

Seção 1.6 A Internet e a World Wide Web

- A Internet é acessível por mais de um bilhão de computadores e dispositivos controlados por computador.
- Com a introdução da World Wide Web, a Internet tornou-se um dos principais mecanismos de comunicação do mundo.

Seção 1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível

- Qualquer computador pode entender diretamente somente sua própria linguagem de máquina.
- A linguagem de máquina é a "linguagem natural" de um computador.
- As linguagens de máquina consistem geralmente em strings de números (em última instância reduzidas a 1s e 0s) que instruem os computadores a realizar suas operações mais elementares uma de cada vez.
- Qualquer computador pode entender diretamente apenas sua própria linguagem de máquina.
- Os programadores começaram a utilizar abreviações como aquelas do idioma inglês para representar operações elementares. Essas abreviações formaram a base de linguagens assembly.
- Programas tradutores chamados assemblers foram desenvolvidos para converter os primeiros programas de linguagem assembly em linguagem de máquina a velocidades de computador.
- Para acelerar o processo de programação, foram desenvolvidas linguagens de alto nível em que instruções únicas poderiam ser escritas para realizar tarefas substanciais.
- Os programas tradutores chamados compiladores convertem os programas de linguagem de alto nível em linguagem de máquina.

- Linguagens de alto nível permitem aos programadores escrever instruções que se parecem com o inglês cotidiano e contêm notações matemáticas comumente utilizadas.
- O Java é de longe a linguagem de alto nível mais amplamente usada.
- O processo de compilação de um programa de linguagem de alto nível em linguagem de máquina pode consumir uma quantidade considerável de tempo do computador. Os programas interpretadores foram desenvolvidos para executar programas de linguagem de alto nível diretamente (sem o tempo de espera da compilação), embora mais lentos do que a execução de programas compilados.

Seção 1.8 História do C e do C++

- O Java evoluiu do C++, que evoluiu do C, que evoluiu do BCPL e do B.
- A linguagem C foi desenvolvida por Dennis Ritchie na Bell Laboratories. Inicialmente, ela tornou-se muito conhecida como a linguagem de desenvolvimento do sistema operacional UNIX.
- O C++, uma extensão do C, foi desenvolvido por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ fornece alguns recursos que sofisticam a linguagem C e a capacidade para a programação orientada a objetos.

Seção 1.9 História do Java

- O Java é usado para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores da Web, fornecer aplicativos para dispositivos de consumo popular e para muitos outros fins.
- Programas Java consistem em partes chamadas classes. As classes incluem partes chamadas métodos que realizam tarefas e retornam informações quando as tarefas são concluídas.

Seção 1.10 Bibliotecas de classe do Java

- A maioria dos programadores Java tira proveito das ricas coleções de classes existentes nas bibliotecas de classe Java, também conhecidas como Java APIs (Application Programming Interfaces).
- A vantagem de criar suas próprias classes e métodos é que você sabe como eles funcionam e pode examinar o código. A desvantagem é o esforço demorado e potencialmente complexo.

Seção 1.11 Fortran, Cobol, Pascal e Ada

- A linguagem Fortran (*FORmula TRANslator*) foi desenvolvida pela IBM Corporation em meados da década de 1950 para utilização em aplicativos científicos e de engenharia que exigem complexos cálculos matemáticos.
- O Cobol (*COmmon Business Oriented Language*) é utilizado para aplicativos comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados.
- Pesquisas nos anos 1960 resultaram na programação estruturada — uma abordagem para escrever programas que são mais claros e mais fáceis de testar, depurar e modificar do que aqueles produzidos com as técnicas anteriores.
- O Pascal foi projetado para ensinar a programação estruturada em ambientes acadêmicos e rapidamente tornou-se a linguagem de programação preferida na maioria das universidades.
- A linguagem de programação Ada foi desenvolvida com o patrocínio do U.S. Department Of Defense (DOD) para atender à maioria das suas necessidades. Uma capacidade do Ada chamada multitarefa permite aos programadores especificar que atividades devem ocorrer em paralelo. O Java, por meio de uma técnica chamada *multithreading*, também permite que os programadores escrevam programas com atividades paralelas.

Seção 1.12 Basic, Visual Basic, Visual C++, C# e .NET

- O Basic foi desenvolvido em meados dos anos 60 para escrever programas simples.
- A linguagem Visual Basic da Microsoft simplifica o desenvolvimento de aplicativos Windows.
- A plataforma .NET da Microsoft integra a Internet e a Web a aplicativos de computador.

Seção 1.13 Ambiente típico de desenvolvimento Java

- Os programas Java normalmente passam por cinco fases — edição, compilação, carga, verificação e execução.
- A fase 1 consiste na edição de um arquivo com um editor. Você digita um programa utilizando o editor, faz correções e salva o programa em um dispositivo de armazenamento secundário, como sua unidade de disco rígido.
- Um nome de arquivo que termina com a extensão `.java` indica que o arquivo contém o código-fonte Java.
- Ambientes de desenvolvimento integrados (IDEs) fornecem ferramentas que dão suporte ao desenvolvimento de softwares, incluindo editores para escrever e editar programas e depuradores para localizar erros de lógica.
- Na Fase 2, o programador usa o comando `javac` para compilar um programa.
- Se um programa compilar, o compilador produz um arquivo `.class` que contém o programa compilado.
- O compilador Java converte o código-fonte Java em bytecodes que representam as tarefas a ser executadas. Bytecodes são executados pela Java Virtual Machine (JVM).

- Na Fase 3, carregamento, o carregador de classe seleciona os arquivos `.class` que contêm os bytecodes do programa e transfere-os para a memória primária.
- Na Fase 4, enquanto as classes são carregadas, o verificador de bytecode examina seus bytecodes para assegurar que eles são válidos e não violam restrições de segurança do Java.
- Na Fase 5, a JVM executa os bytecodes do programa.

Seção 1.14 Notas sobre o Java e este livro

- Portabilidade é um objetivo vago. Simplesmente escrever programas em Java não garante portabilidade.

Seção 1.15 Testando um aplicativo Java

- Nesta seção, você executou e interagiu com seu primeiro aplicativo Java.

Seção 1.16 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML

- A Unified Modeling Language (UML) é uma linguagem gráfica que permite que as pessoas construam sistemas para representar seus designs orientados a objetos em uma notação comum.
- O design orientado a objetos (*Object-Oriented Design* — OOD) modela componentes de software em termos de objetos do mundo real.
- Objetos têm a propriedade de ocultamento de informações — os objetos de uma classe normalmente não sabem como os objetos de outras classes são implementados.
- A Programação Orientada a Objetos (POO) implementa designs orientados a objetos.
- Os programadores Java concentram-se em criar seus próprios tipos definidos pelo usuário chamados classes. Cada classe contém dados bem como métodos que manipulam esses dados e fornecem serviços aos clientes.
- Os componentes de dados de uma classe são atributos ou campos; os componentes de operação são métodos.
- Classes podem ter relacionamentos com outras classes; esses relacionamentos são chamados associações.
- Empacotar software como classes possibilita que os sistemas de software futuros reutilizem as classes.
- Uma instância de uma classe é chamada de objeto.
- O processo de analisar e projetar um sistema a partir de um ponto de vista orientado a objetos é chamado análise orientada a objetos e design (*Object-Oriented Analysis and Design* — OOAD).

Seção 1.17 Web 2.0

- O ressurgimento da Web, que começou mais ou menos em 2004, foi chamado Web 2.0.
- Mashups permitem desenvolver rapidamente aplicativos poderosos e intrigantes combinando serviços Web complementares e outras formas de feeds de informações de duas ou mais organizações.
- O Ajax ajuda aplicativos baseados na Internet a ter um desempenho comparável ao dos aplicativos desktop — uma tarefa difícil, dado que esses aplicativos sofrem atrasos de transmissão à medida que os dados são transmitidos entre seu computador e outros computadores na Internet.
- Blogs são sites Web que se parecem com diários online, com as entradas mais recentes aparecendo primeiro.
- RSS feeds permitem que sites incluam informações aos assinantes.
- A Web 3.0 é outro nome para a próxima geração da Web, também chamada Web semântica. A Web 3.0 utilizará intensamente a XML, criando "uma teia de significados".

Seção 1.18 Tecnologias de software

- O Agile Software Development é um grupo de metodologias que tenta implementar softwares rapidamente utilizando menos recursos do que as metodologias anteriores.
- A refatoração envolve a reformulação do código para torná-lo mais claro e mais fácil de manter e, ao mesmo tempo, preservar sua funcionalidade.
- Padrões de design são arquiteturas testadas para construir softwares orientados a objetos flexíveis e que podem ser mantidos.
- A programação de jogos é uma carreira interessante e desafiadora para desenvolvedores de software. Cursos universitários e mesmo especializações dedicam-se às técnicas de software sofisticadas utilizadas na programação de jogos.
- Com o desenvolvimento de softwares de código-fonte aberto, pessoas e empresas contribuem com seus esforços para desenvolver, manter e evoluir os softwares em troca do direito de utilizar esses softwares para objetivos próprios, em geral gratuitamente. Código de código-fonte aberto geralmente é realizado por um grupo de pessoas bem maior do que softwares proprietários, assim erros muitas vezes são removidos mais rapidamente. Código de código-fonte aberto também estimula mais inovações.
- O Linux é um sistema operacional de código-fonte aberto e um dos maiores sucessos do movimento de código-fonte aberto. O MySQL é um sistema de gerenciamento de bancos de dados de código-fonte aberto. O PHP é a linguagem Internet para criação de scripts de código-fonte aberto do lado do servidor mais popular para desenvolver aplicativos baseados na Internet. LAMP é um acrônimo para o conjunto de tecnologias de código-fonte aberto que muitos desenvolvedores utilizavam para construir aplicativos Web — ele significa Linux, Apache, MySQL e PHP (ou Perl ou Python — duas outras linguagens utilizadas para objetivos semelhantes).

- O Ruby on Rails combina a linguagem de criação de scripts Ruby com a estrutura de aplicativo Web do Rails desenvolvida pela empresa 37Signals. Muitos desenvolvedores Ruby in Rails informaram ganhos de produtividade significativos em relação ao uso de outras linguagens ao desenvolver aplicativos Web que utilizam intensamente banco de dados.
- Com o Software As A Service (SAAS), os softwares executam em servidores em outros locais na Internet. Você acessa o serviço por um navegador. Navegadores são bem portáteis, portanto você pode executar os mesmos aplicativos em diferentes tipos de computadores a partir de qualquer lugar no mundo.

Terminologia

Ada, linguagem de programação, 8
Agile Software Development, 18
Ajax (Asynchronous JavaScript and XML), 18
ALU (*Arithmetic and Logic Unit*), 4
ambiente de desenvolvimento integrado, 9
analisar os requisitos de um projeto, 16
análise e design orientado a objetos (*Object-Oriented Analysis and Design* — OOAD), 16
aplicativo de terminal (Mac OS X), 10
assembler, 5
associação, 16
associação na UML, 16
Basic (Beginner's All-Purpose Symbolic Instruction Code), 8
blog, 18
blogosfera, 18
boa prática de programação, 7
Bytecode, 10
campo, 16
carregador de classe, 10
carregando, 10
`.class`, arquivo, 10
classe, 7, 15
C#, linguagem de programação, 8
Cobol (*COmmon Business Oriented Language*), 7
código-fonte, 8
código-fonte aberto, software, 18
Command Prompt, 10
compartilhamento de tempo, 4
compilador, 5
compilar um programa, 10
componente, 6, 16
componente na UML, 16
computação cliente/servidor, 5
computação distribuída, 4
computação pessoal, 4
computador, 2
computador cliente, 5
conteúdo dinâmico, 6
CPU, 4
dados, 3
Deitel® Buzz Online, boletim, 19
dependente de máquina, 5
depurar, 7
design orientado a objetos (*Object-Oriented Design* — OOD), 15
Dica de desempenho, 7
Dica de portabilidade, 7
Dica de prevenção de erro, 7
dispositivos de entrada, 3
dispositivos de saída, 3
editor, 8
encapsulamento, 15
erro de lógica, 9
Erro de programação comum, 7
erro de tempo de execução, 11
erro fatal, 11
erro não fatal de tempo de execução, 11
estação de trabalho, 4
Fortran (*FORmula TRANslator*), 7
função, 16
Google, 17
hardware, 3, 5
herança, 15
IDE, 9
informações voláteis, 3
instanciar, 16
interface, 15
interface de programas aplicativos (*Applications Programming Interface* — API), 7
Internet, 5
interpretador, 6
`java`, comando, 10, 12
Java Application Programming Interface (Java API), 7
Java, biblioteca de classe, 7
Java, compilador, 10
Java Enterprise Edition (Java EE), 3
`.java`, extensão, 10
Java HotSpot, compilador, 10
Java Micro Edition (Java ME), 3
Java Standard Edition (Java SE), 3
Java Virtual Machine (JVM), 10, 11
JIT, compilador, 10
just-in-time, compilação, 10
Just-In-Time (JIT), compilador, 10
LAMP, 19
linguagem assembly, 5
linguagem de alto nível, 5
linguagem de máquina, 5
linguagem de programação procedural, 16
linguagem extensível, 17
linguagem orientada a objetos, 16
Linux, 5, 8, 19
live-code, abordagem, 2
máquina virtual (VM), 10
mashups, 18
memória, 3
memória principal, 3
método, 7, 16
multiprocessador, 4
multiprogramação, 4
multitarefa, 8
MySQL, 19
Object Management Group (OMG), 17
Object Management Group (`www.omg.org`), 17
Observações de engenharia de software, 7
Observações sobre a aparência e comportamento, 7
ocultamento de informações, 15
OOD (*Object-Oriented Design*), 15
OOP (*Object-Oriented Programming*), 16
Operação de um objeto, 15
orientado para a ação, 16
padrões de projeto (*design patterns*), 18
persistente, 4
PHP, 19
plataforma .NET, 8
pontos ativos no bytecode, 10
portável, 10
processador de múltiplos núcleos, 4
processamento em lote, 4
processo de análise, 16
Programação de jogos, 18
programação estruturada, 2, 3, 8
programa de computador, 3
programador, 3
programador de computador, 3
programa tradutor, 5
pseudocódigo, 16
rede local (*Local Area Network* — LAN), 4
refatorando, 18
requisitos, 16
RSS feeds, 18
Ruby on Rails, 19
serviços Web, 18
servidor, 5
shell, prompt no Linux, 10
sistema operacional, 4
Software As A Service (SAAS), 19
supercomputador, 3
tarefa, 4
Technorati, 18
throughput, 4
trabalho, 4
tradução, 5
UML, parceiros, 17
UML (Unified Modeling Language), 15
unidade central de processamento (*Central Processing Unit* — CPU), 4
Unidade de Aritmética e Lógica (ALU), 4
unidade de armazenamento secundária, 4
unidade de entrada, 3
unidade de memória, 3
unidade de saída, 3
unidade lógica, 4
Unified Modeling Language (UML), 15, 16
verificador de bytecode, 10
Visual Basic, linguagem de programação, 8
Visual C#, linguagem de programação, 8
Visual C++, linguagem de programação, 8
Web 2.0, 17
Web 3.0, 18
Web semântica, 18
World Wide Web (WWW), 5

Exercícios de autorrevisão

1.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) A empresa que popularizou a computação pessoal foi _____.

b) O computador que tornou computação pessoal legítima nos negócios e na indústria foi _____.

c) Os computadores processam dados sob o controle de conjuntos de instruções chamados _____.

d) As principais unidades lógicas do computador são _____, _____, _____, _____, _____ e _____.

e) Os três tipos de linguagens discutidas no capítulo são _____, _____ e _____.

f) Os programas que traduzem programas de linguagem de alto nível em linguagem de máquina são chamados _____.

g) O _____ permite aos usuários de computador localizar e visualizar documentos baseados em multimídia sobre aproximadamente qualquer assunto pela Internet.

h) _____ permite que um programa Java realize múltiplas atividades paralelamente.

1.2 Preencha as lacunas em cada uma das seguintes frases sobre o ambiente Java:

a) O comando _____ do JDK executa um aplicativo Java.

b) O comando _____ do JDK compila um programa Java.

c) Um arquivo de programa Java deve terminar com a extensão de arquivo _____.

d) Quando um programa Java é compilado, o arquivo produzido pelo compilador termina com a extensão de arquivo _____.

e) O arquivo produzido pelo compilador Java contém _____ que são executados pela Java Virtual Machine.

1.3 Preencha as lacunas de cada uma das sentenças a seguir (com base na Seção 1.16):

a) Os objetos têm a propriedade de _____ — embora os objetos possam saber comunicar entre si por interfaces bem definidas, normalmente não têm permissão de saber como outros objetos são implementados.

b) Os programadores Java concentram-se na criação de _____, que contém campos e o conjunto de métodos que manipulam esses campos e fornecem serviços para clientes.

c) As classes podem ter relacionamentos chamados _____ com outras classes.

d) O processo de analisar e projetar um sistema de um ponto de vista orientado a objetos é chamado _____.

e) O OOD tira proveito de relacionamentos de _____, nos quais novas classes de objetos são derivadas absorvendo características de classes existentes e, em seguida, adicionando características únicas dessas mesmas classes.

f) _____ é uma linguagem gráfica que permite que as pessoas que projetam sistemas de software utilizem uma notação padrão da indústria para representá-las.

g) O tamanho, forma, cor e peso de um objeto são considerados _____ da classe do objeto.

Respostas dos exercícios de autorrevisão

1.1 a) Apple. b) computador pessoal IBM. c) programas. d) unidade de entrada, unidade de saída, unidade de memória, unidade central de processamento, unidade aritmética e lógica e unidade de armazenamento secundário. e) linguagens de máquina, linguagens assembly e linguagens de alto nível. f) compiladores. g) World Wide Web. h) Multithreading.

1.2 a) `java`. b) `javac`. c) `.java`. d) `.class`. e) bytecodes.

1.3 a) ocultamento de informações. b) classes. c) associações. d) análise orientada a objetos e design (Object-Oriented Analysis and Design — OOAD). e) herança. f) Unified Modeling Language (UML). g) atributos.

Exercícios

1.4 Categorize cada um dos itens seguintes como hardware ou software:

a) CPU

b) Java, compilador

c) JVM

d) unidade de entrada

e) editor

1.5 Preencha as lacunas em cada uma das seguintes afirmações:

a) A unidade lógica do computador que recebe informações de fora do computador para utilização pelo computador é a _____.

b) O processo de instrução do computador para resolver um problema específico é chamado _____.

c) _____ é um tipo de linguagem de computador que utiliza abreviações em inglês para instruções de linguagem de máquina.

d) _____ é uma unidade lógica do computador que envia informações que já foram processadas pelo computador para vários dispositivos de modo que possam ser utilizadas fora do computador.

e) _____ e _____ são unidades lógicas do computador que retêm informações.

f) _____ é uma unidade lógica do computador que realiza cálculos.

g) _____ é uma unidade lógica do computador que toma decisões lógicas.

h) As linguagens mais convenientes para que programador escreva programas rápida e facilmente são as linguagens _____.

i) A única linguagem que um computador pode entender diretamente é a _____ do computador.

j) _____ é uma unidade lógica do computador que coordena as atividades de todas as outras unidades lógicas.

1.6 Qual é a diferença entre erros fatais e erros não fatais? Por que você poderia preferir que o programa sofresse um erro fatal em vez de um erro não fatal?

1.7 Preencha as lacunas em cada uma das seguintes afirmações:

a) _____ é agora utilizado para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores da Web, fornecer aplicativos para dispositivos de consumo popular e para muitos outros propósitos.

b) _____ foi projetada especificamente para a plataforma .NET para permitir aos programadores migrar facilmente para .NET.

c) Inicialmente, o _____ tornou-se muito conhecido como a linguagem de desenvolvimento do sistema operacional UNIX.

d) _____ foi desenvolvida no Dartmouth College em meados da década de 1960 como um meio de escrever programas simples.

e) _____ foi desenvolvida pela IBM Corporation em meados da década de 1950 para ser utilizada para aplicativos científicos e de engenharia que requerem complexos cálculos matemáticos.

f) _____ é utilizada para aplicativos comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados.

g) A linguagem de programação _____ foi desenvolvida por Bjarne Stroustrup no início dos anos 80 na Bell Laboratories.

1.8 Preencha as lacunas em cada uma das seguintes afirmações:

a) Os programas Java normalmente passam por cinco fases: _____, _____, _____, _____ e _____.

b) Um(a) _____ fornece muitas ferramentas que suportam o processo de desenvolvimento de software, como editores para escrever e editar programas, depuradores para localizar erros de lógica em programas e muitos outros recursos.

c) O comando `java` invoca _____, que executa programas Java.

d) Uma _____ é um aplicativo de software que simula um computador, mas oculta o sistema operacional e o hardware subjacentes dos programas que interagem com ela.

e) Um programa _____ pode executar em múltiplas plataformas.

f) O _____ transfere os arquivos `.class` contendo os bytecodes do programa para a memória principal.

g) O _____ examina bytecodes para assegurar que eles são válidos.

1.9 Explique as duas fases de compilação de programas Java.

1.10 Provavelmente você esteja usando no pulso um dos tipos mais comuns de objetos do mundo — um relógio. Discuta como cada um dos seguintes termos e conceitos se aplicam à noção de um relógio: objeto, atributos, comportamentos, classe, herança (considere, por exemplo, o alarme de um relógio), abstração, modelagem, mensagens, encapsulamento, interface e ocultamento de informações.

Fazendo a diferença

1.11 (***Test-drive: Calculadora de emissão de carbono***) Alguns cientistas acreditam que as emissões de carbono, especialmente da queima de combustíveis fósseis, contribuem significativamente para o aquecimento global e que isso pode ser combatido se as pessoas tomarem medidas para limitar o uso de combustíveis baseados no carbono. As organizações e pessoas estão cada vez mais preocupadas com suas "emissões de carbono". Sites Web, como o TerraPass:

`www.terrapass.com/carbon-footprint-calculator/`

e o Carbon Footprint:

`www.carbonfootprint.com/calculator.aspx`

fornecem calculadoras de emissão de carbono. Teste essas calculadoras para determinar as sua emissões de carbono. Os exercícios nos próximos capítulos solicitarão que você programe sua própria calculadora. Para preparar-se para isso, pesquise as fórmulas para calcular as emissões de carbono.

1.12 (***Test-drive: Calculadora de índice de massa corpórea***) De acordo com estimativas recentes, dois terços dos norte-americanos estão acima do peso e aproximadamente metade destes são obesos. Isso causa aumentos significativos de doenças como diabetes e doenças cardíacas. Para determinar se uma pessoa está acima do peso ou obesa, você pode utilizar uma medida chamada índice de massa corpórea (IMC). Os departamentos de assistência social e de saúde norte-americanos fornecem uma calculadora do IMC em `www.nhlbisupport.com/bmi/`. Utilize-o para calcular seu próprio IMC. Um exercício no Capítulo 2 solicitará que você programe sua própria calculadora de IMC. Para preparar-se para isso, pesquise as fórmulas para calcular o IMC.

1.13 (***Atributos dos veículos híbridos***) Neste capítulo você aprendeu os conceitos básicos das classes. Agora você começará a detalhar os aspectos da classe chamada "Veículo híbrido". Veículos híbridos estão se tornando cada vez mais populares, porque muitas vezes eles têm um desempenho por litro de combustível muito melhor do que veículos que só utilizam gasolina. Navegue pela Web e estude os recursos de quatro ou cinco dos atuais carros híbridos mais populares e então liste o maior número possível dos atributos relacionados a veículos híbridos. Por exemplo, atributos comuns incluem consumo urbano por litro de combustível e consumo em estradas. Também liste os atributos das baterias (tipo, peso etc.).

1.14 **(*Neutralidade de sexo*)** Muitas pessoas querem eliminar o machismo em todas as formas de comunicação. Você foi solicitado a criar um programa que pode processar um parágrafo do texto e substituir palavras de sexo específico por outras neutras. Supondo que você recebeu uma lista de palavras de sexo específico e suas substituições neutras em relação a ele (por exemplo, substitua "esposa" por "cônjuge", "homem" por "pessoa", "filha" por "criança" etc.), explique o procedimento que você utilizaria para ler um parágrafo do início ao fim do texto e fazer manualmente essas substituições. Pensando no idioma inglês, como seu procedimento poderia gerar um termo estranho como "woperchild", que de fato está listado no Urban Dictionary (`www.urbandictionary.com`)? No Capítulo 4, você aprenderá que um termo mais formal para "procedimento" é "algoritmo", e que um algoritmo especifica os passos a serem executados e a ordem para executá-los.

2

Que há em um simples nome? O que chamamos rosa,
sob outra designação teria igual perfume.
— William Shakespeare

Quando preciso tomar uma decisão, sempre pergunto, "O que seria mais divertido?"
— Peggy Walker

"Tome mais chá", a Lebre de Março disse para Alice, muito sinceramente. "Eu ainda não bebi nada", Alice respondeu em um tom ofendido: "não posso tomar mais". "Você quis dizer que não podes tomar menos", disse Leirão: "É muito fácil tomar mais do que não tomar nada".
— Lewis Carroll

Introdução aos aplicativos Java

Objetivos

Neste capítulo, você aprenderá:

- A escrever aplicativos Java simples.
- A utilizar instruções de entrada e saída.
- Os tipos primitivos do Java.
- Os conceitos básicos de memória.
- A utilizar operadores aritméticos.
- A precedência dos operadores aritméticos.
- A escrever instruções de tomada de decisão.
- A utilizar operadores relacionais de igualdade.

Sumário

2.1 Introdução
2.2 Nosso primeiro programa Java: imprimindo uma linha de texto
2.3 Modificando nosso primeiro programa Java
2.4 Exibindo texto com `printf`
2.5 Outro aplicativo: adicionando inteiros
2.6 Conceitos de memória
2.7 Aritmética
2.8 Tomada de decisão: operadores de igualdade e operadores relacionais
2.9 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

2.1 Introdução

Neste capítulo, apresentamos a programação de aplicativos Java com programas simples para uma introdução à criação de programas Java. Começamos com vários exemplos que exibem mensagens na tela e, então, demonstramos um programa que obtém dois números de um usuário, calcula sua soma e exibe o resultado. Você aprenderá a realizar cálculos aritméticos e salvar os resultados para utilização posterior. O último exemplo demonstra a tomada de decisão mostrando como comparar números e, em seguida, exibir mensagens baseadas nos resultados de comparação. Este capítulo utiliza as ferramentas do JDK para compilar e executar programas. Além disso, para ajudá-lo a começar a usar os populares ambientes de desenvolvimento integrados Eclipse e NetBeans, postamos vídeos em `www.deitel.com/books/jhtp8/`.

2.2 Nosso primeiro programa Java: imprimindo uma linha de texto

Um **aplicativo** Java é um programa de computador que é executado quando você utiliza o **comando `java`** para carregar a Java Virtual Machine (JVM). Vamos examinar um aplicativo simples que exibe uma linha de texto. (Mais adiante, nesta seção, discutiremos como compilar e executar um aplicativo.) O programa e sua saída são mostrados na Figura 2.1. A saída aparece no quadro no fim do programa. O programa ilustra vários recursos importantes da linguagem Java. Todo programa que apresentamos tem números de linha, que não fazem parte dos programas Java reais. Mais adiante, veremos que a linha 9 faz o trabalho real do programa — exibir a frase `Welcome to Java Programming!` na tela.

```
1  // Figura 2.1: Welcome1.java
2  // Programa de impressão de texto.
3
4  public class Welcome1
5  {
6     // método principal inicia a execução do aplicativo Java
7     public static void main( String[] args )
8     {
9        System.out.println( "Welcome to Java Programming!" );
10    } // fim do método main
11 } // fim da classe Welcome1
```

```
Welcome to Java Programming!
```

Figura 2.1 | Programa de impressão de texto.

Comentando programas

A linha 1

```
// Figura 2.1: Welcome1.java
```

começa com `//`, que indica que a linha é um **comentário**. Você insere comentários para **documentar programas** e aprimorar sua legibilidade. O compilador Java ignora os comentários, portanto eles não fazem o computador realizar nenhuma ação quando o programa é executado. Por convenção, iniciamos cada programa com um comentário indicando o número da figura e o nome do arquivo.

Um comentário que começa com `//` é um **comentário de fim de linha** — ele termina no fim da linha em que aparece. Um comentário de fim de linha também pode iniciar no meio de uma linha e continua até o fim dessa linha (como nas linhas 10–11). Outro tipo de comentário, chamado de **comentário tradicional**, pode ser distribuído por várias linhas como em:

```
/* Isso é um comentário tradicional. Ele
   pode ser dividido em várias linhas */
```

Esse tipo de comentário começa com **/*** e termina com ***/**. Todo o texto entre os delimitadores é ignorado pelo compilador. O Java incorporou comentários tradicionais e comentários de fim de linha das linguagens de programação C e C++, respectivamente. Neste livro, utilizamos somente comentários `//`.

O Java também fornece comentários no estilo **Javadoc** que são delimitados por **/**** e ***/**. Como ocorre com comentários tradicionais, todo o texto entre os delimitadores de comentário no estilo Javadoc é ignorado pelo compilador. Os comentários no estilo Javadoc permitem-lhe incorporar a documentação do programa diretamente aos seus programas. Esses comentários são o formato de documentação Java preferido na indústria. O **programa utilitário `javadoc`** (parte do Java SE Development Kit) lê comentários no estilo Javadoc e utiliza-os para preparar a documentação do seu programa no formato HTML. Demonstramos comentários de Javadoc e o utilitário `javadoc` no Apêndice M, "Criando documentação com o `javadoc`".

A linha 2

```
// Programa de impressão de texto.
```

é um comentário que descreve o propósito do programa.

Erro comum de programação 2.1

Esquecer um dos delimitadores de um comentário tradicional no estilo Javadoc causa um erro de sintaxe. A **sintaxe** *de uma linguagem de programação especifica as regras para criar programas apropriados nessa linguagem, assim como as regras de gramática de uma língua natural especificam a estrutura da frase. Um* **erro de sintaxe** *ocorre quando o compilador encontra o código que viola as regras da linguagem do Java (isto é, sua sintaxe). Nesse caso, o compilador emite uma mensagem de erro e impede o programa de compilar. Erros de sintaxe também são chamados* **erros de compilador**, **erros em tempo de compilação** *ou* **erros de compilação**, *porque o compilador detecta-os durante a fase de compilação.*

Boa prática de programação 2.1

Algumas organizações exigem que todo programa comece com um comentário que informa o objetivo e o autor do programa, a data e a hora em que o programa foi modificado pela última vez.

Utilizando linhas em branco

A linha 3 é uma linha em branco. As linhas em branco e caracteres de espaço tornam os programas mais fáceis de ler. Juntos, as linhas em branco, os espaços e as tabulações são conhecidos como **espaço em branco**. Espaços em branco são ignorados pelo compilador.

Boa prática de programação 2.2

Utilize linhas e espaços em branco para aprimorar a legibilidade do programa.

Declarando uma classe

A linha 4

```
public class Welcome1
```

inicia uma **declaração de classe** da classe `Welcome1`. Todo programa Java consiste em pelo menos uma classe que você (o programador) define. Essas são conhecidas como **classes definidas pelo programador**. A **palavra-chave `class`** introduz uma declaração de classe e é imediatamente seguida pelo **nome de classe** (`Welcome1`). **Palavras-chave** (às vezes chamadas de **palavras reservadas**) são reservadas para uso pelo Java e sempre escritas com todas as letras minúsculas. A lista completa de palavras-chave está no Apêndice C.

Por convenção, os nomes de classes iniciam com uma letra maiúscula e apresentam a letra inicial de cada palavra que eles incluem em maiúscula (por exemplo, `SampleClassName`). O nome de uma classe é um **identificador** — uma série de caracteres que consiste em letras, dígitos, sublinhados (_) e sinais de cifrão ($) que não iniciam com um dígito e não contêm espaços. Alguns identificadores válidos são `Bemvindo1`, `$valor`, `_valor`, `m_campoDeEntrada1` e `botao7`. O nome `7button` não é um identificador válido porque inicia com um dígito; e o nome `input field` não é um identificador válido porque contém um espaço. Normalmente, um identificador que não inicia com uma letra maiúscula não é um nome de classe. O Java faz **distinção entre letras maiúsculas e minúsculas** — letras maiúsculas e letras minúsculas são diferentes — assim, `a1` e `A1` são identificadores diferentes (mas ambos válidos).

Boa prática de programação 2.3

Por convenção, inicie o identificador de cada nome de classe com uma letra maiúscula e inicie cada palavra subsequente do identificador com uma letra maiúscula.

Erro comum de programação 2.2

O Java diferencia letras maiúsculas de minúsculas. O uso incorreto de letras maiúsculas e minúsculas para um identificador normalmente causa um erro de compilação.

Nos capítulos 2–7, cada classe que definimos inicia com a palavra-chave **public**. Por enquanto, simplesmente exigiremos essa palavra-chave. Para nosso aplicativo, o nome de arquivo é `Welcome1.java`. Você aprenderá mais sobre as classes `public` e não `public` no Capítulo 8.

Erro comum de programação 2.3

Um classe `public` deve ser colocada em um arquivo que tenha o mesmo nome da classe (tanto na ortografia como no uso de maiúsculas e minúsculas) mais a extensão `.java`; caso contrário, ocorre um erro de compilação. Por exemplo, a classe `public Welcome` deve ser colocada em um arquivo chamado `Welcome.java`.

Uma **chave esquerda** (na linha 5 desse programa), **{**, inicia o **corpo** de cada declaração de classe. Uma **chave direita** , **}**, correspondente (na linha 11) deve terminar cada declaração de classe. Observe que as linhas 6–10 estão recuadas. Esse recuo é uma das convenções de espaçamento mencionadas anteriormente.

Dica de prevenção de erro 2.1

Quando você digitar uma chave de abertura, ou chave esquerda, {, imediatamente digite a chave de fechamento, ou chave direita, }, e, então reposicione o cursor entre as chaves e dê um recuo para começar a digitação do corpo. Essa prática ajuda a evitar erros devidos à ausência das chaves. Muitos IDEs inserem os colchetes para você.

Boa prática de programação 2.4

Recue o corpo inteiro de cada declaração de classe por um "nível" entre a chave esquerda e a chave direita que delimitam o corpo da classe. Esse formato enfatiza a estrutura da declaração de classe e torna mais fácil sua leitura.

Boa prática de programação 2.5

Muitos IDEs inserem recuos ou indentações em todos os lugares certos. A tecla Tab também pode ser utilizada para criar recuos, mas as paradas de tabulação variam entre editores de textos. Recomendamos a utilização de três espaços para formar um nível de recuo.

Erro comum de programação 2.4

É um erro de sintaxe se chaves não ocorrerem em pares correspondentes.

Declarando um método

A linha 6

```
// método principal inicia a execução do aplicativo Java
```

é um comentário de fim de linha indicando o propósito das linhas 7–10 do programa. A linha 7:

```
public static void main( String[] args )
```

é o ponto de partida de cada aplicativo Java. Os **parênteses** depois do identificador `main` indicam que ele é um bloco de construção do programa chamado **método**. Declarações de classe Java normalmente contêm um ou mais métodos. Para um aplicativo Java, um dos métodos deve ser chamado `main` e deve ser definido como mostrado na linha 7; caso contrário, a JVM não executará o aplicativo. Os métodos realizam tarefas e podem retornar informações quando completam suas tarefas. A palavra-chave **`void`** indica que esse método não devolverá nenhuma informação. Mais tarde, veremos como um método pode retornar informações. Por enquanto, basta simular a primeira linha de `main` nos aplicativos Java. Na linha 7, a `String[] args` entre parênteses é uma parte requerida da declaração do método `main`. Discutiremos isso no Capítulo 7.

A chave esquerda, `{`, na linha 8 inicia o **corpo da definição de método**. Uma chave direita correspondente deve terminar o corpo da declaração do método (linha 10). Note que a linha 9 no corpo do método está recuada entre as chaves.

Boa prática de programação 2.6
Recue o corpo inteiro de cada declaração de método um "nível" entre as chaves que definem o corpo do método. Isso faz com que a estrutura do método se destaque, tornando a declaração do método mais fácil de ler.

Gerando saída com `System.out.println`

A linha 9

```
System.out.println( "Welcome to Java Programming!" );
```

instrui o computador a realizar uma ação — a saber, imprimir a **string** de caracteres contida entre as aspas duplas (mas não as próprias aspas duplas). Uma string às vezes é chamada de **string de caractere** ou **string literal**. Os caracteres de espaço em branco em strings não são ignorados pelo compilador. As strings não podem distribuir várias linhas de código.

O objeto **`System.out`** é conhecido como o **objeto de saída padrão**. Ele permite que aplicativos Java exibam strings na **janela de comando** a partir da qual o aplicativo de Java executa. Em versões recentes do Microsoft Windows, a janela de comando chama-se Prompt de Comando. No UNIX/Linux/Mac OS X, a janela de comando é chamada **janela terminal** ou **shell**. Muitos programadores chamam a janela de comando simplesmente de **linha de comando**.

O método **`System.out.println`** exibe (ou imprime) uma linha de texto na janela de comando. A string entre parênteses na linha 9 é o **argumento** para o método. Quando `System.out.println` completa sua tarefa, ele posiciona o cursor de saída (o local em que o próximo caractere será exibido) no começo da linha seguinte na janela de comando. (Esse movimento do cursor é semelhante ao pressionamento da tecla Enter ao digitar em um editor de textos — o cursor aparece no começo da próxima linha do documento.)

A linha 9 inteira, incluindo `System.out.println`, o argumento `"Welcome to Java Programming!"` entre parênteses e o **ponto-e-vírgula** (`;`), é uma **instrução**. A maioria das instruções acaba com um ponto-e-vírgula. Quando a instrução na linha 9 executa, ela exibe `Welcome to Java Programming!` na janela de comando. Em geral, um método contém uma ou várias instruções que realizam sua tarefa.

Erro comum de programação 2.5
Quando um ponto-e-vírgula é necessário para acabar uma instrução, omitir o ponto-e-vírgula é um erro de sintaxe.

Dica de prevenção de erro 2.2
Ao aprender a programar, às vezes é útil "quebrar" um programa funcional para você poder familiarizar-se com as mensagens de erro de sintaxe do compilador. Essas mensagens nem sempre declaram o problema exato no código. Quando encontrar essas mensagens de erro de sintaxe, você terá uma ideia do que causou o erro. [Tente remover um ponto-e-vírgula ou chave do programa da Figura 2.1 e, então, recompile o programa para ver as mensagens de erro geradas pela omissão.]

Dica de prevenção de erro 2.3
Quando o compilador informa um erro de sintaxe, o erro pode não estar na linha indicada pela mensagem de erro. Primeiro, verifique a linha em que o erro foi informado. Se essa linha não contiver erros de sintaxe, verifique as várias linhas anteriores.

Utilizando comentários de fim de linha em chaves de fechamento para melhorar a legibilidade

Incluímos um comentário de fim de linha depois de uma chave de fechamento que termina uma **declaração de método** e depois de uma chave de fechamento que termina uma declaração de classe. Por exemplo, a linha 10

```
} // fim do método main
```

indica a chave de fechamento do método `main`, e a linha 11

```
} // fim da classe Welcome1
```

indica a chave de fechamento da classe `Welcome1`. Cada comentário indica o método ou classe que a chave direita termina.

Boa prática de programação 2.7
Para aprimorar a legibilidade de programa, coloque depois da chave de fechamento do corpo de uma declaração de método ou de classe um comentário que indica a declaração de método ou de classe à qual a chave pertence.

Compilando e executando seu primeiro aplicativo Java

Estamos agora prontos para compilar e executar nosso programa. Supomos que você esteja usando o Java SE Development Kit (JDK) 6 Update 10 da Sun Microsystems, não um IDE. Nossos Java Resource Centers em `www.deitel.com/ResourceCenters.html` fornecem links para tutoriais que o ajudam a começar a usar várias ferramentas de desenvolvimento Java populares, incluindo NetBeans™ e Eclipse™. Também postamos vídeos NetBeans e Eclipse em `www.deitel.com/books/jhtp8/` para ajudá-lo a começar a usar esses IDEs populares.

Para preparar-se para compilar o programa, abra uma janela de comando e vá para o diretório onde o programa está armazenado. Muitos sistemas operacionais utilizam o comando **cd** para mudar de diretório. Por exemplo,

```
cd c:\examples\ch02\fig02_01
```

muda para o diretório `fig02_01` no Windows. O comando:

```
cd ~/examples/ch02/fig02_01
```

muda para o diretório `fig02_01` no UNIX/Linux/Max OS X.

Para compilar o programa, digite:

```
javac Welcome1.java
```

Se o programa não contiver nenhum erro de sintaxe, o comando anterior cria um novo arquivo chamado `Welcome1.class` (conhecido como o **arquivo de classe** para `Welcome1`) que contém os bytecodes Java independentes de plataforma que representam nosso aplicativo. Quando utilizamos o comando `java` para executar o aplicativo em uma dada plataforma, esses bytecodes serão traduzidos pela JVM em instruções que são entendidas pelo sistema operacional subjacente.

Dica de prevenção de erro 2.4

Ao tentar compilar um programa, se receber uma mensagem como "`bad command or filename`," "`javac: command not found`" or "`'javac' is not recognized as an internal or external command, operable program or batch file`", sua instalação do software Java não foi completada corretamente. No JDK, isso é um sinal de que a variável de ambiente `PATH` do sistema não foi configurada corretamente. Revise cuidadosamente as instruções de instalação na Seção "Antes de você começar" deste livro. Em alguns sistemas, depois de corrigir o `PATH`, é necessário reinicializar o computador ou abrir uma nova janela de comando para efetuar as configurações.

Dica de prevenção de erro 2.5

Toda mensagem de erro de sintaxe contém o nome do arquivo e número da linha em que o erro ocorreu. Por exemplo, `Welcome1.java:6` indica que um erro ocorreu no arquivo `Welcome1.java` na linha 6. O restante da mensagem de erro fornece as informações sobre o erro de sintaxe.

Dica de prevenção de erro 2.6

A mensagem de erro do compilador "`class Welcome1 is public, should be declared in a file named Welcome1.java`" indica que o nome de arquivo não corresponde exatamente ao nome da classe `public` no arquivo ou que você digitou o nome de classe incorretamente ao compilar a classe.

A Figura 2.2 mostra o programa da Figura 2.1 executando em uma janela Prompt de Comando do Microsoft® Windows Vista® . Para executar o programa, digite `java Welcome1`. Isso carrega a JVM, que carrega o arquivo "`.class`" para a classe `Welcome1`. Observe que a extensão do nome de arquivo "`.class`" é omitida do comando precedente; caso contrário a JVM não executará o programa. A JVM chama o método `main`. Em seguida, a instrução na linha 9 de `main` exibe `"Welcome to Java Programming!"` [*Nota:* muitos ambientes mostram prompts de comando com fundos pretos e texto na cor branca. Ajustamos essas configurações no nosso ambiente para tornar nossas capturas de tela mais legíveis.]

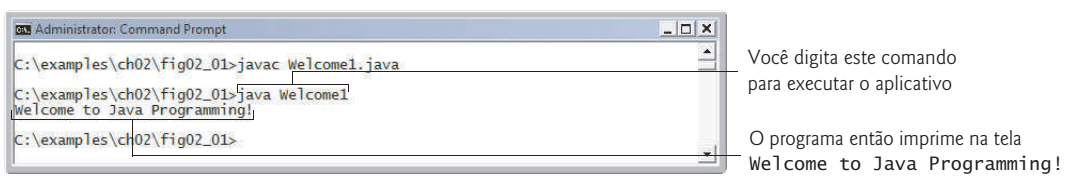


Figura 2.2 | Executando `Welcome1` a partir do Prompt de Comando.

Dica de prevenção de erro 2.7

Ao tentar executar um programa Java, se receber uma mensagem como "`Exception in thread "main" java.lang.NoClassDefFoundError: Welcome1`", sua variável de ambiente `CLASSPATH` não foi configurada corretamente. Revise cuidadosamente as instruções de instalação na Seção "Antes de você começar" deste livro. Em alguns sistemas, talvez seja necessário reinicializar seu computador ou abrir uma nova janela de comando depois de configurar a `CLASSPATH`.

2.3 Modificando nosso primeiro programa Java

Nesta seção, modificamos o exemplo na Figura 2.1 para imprimir texto em uma linha utilizando várias instruções e imprimir texto em várias linhas utilizando uma única instrução.

Exibindo uma linha de texto com múltiplas instruções

`Welcome to Java Programming!` pode ser exibido de várias maneiras. A classe `Welcome2`, mostrada na Figura 2.3, utiliza duas instruções para produzir a mesma saída mostrada na Figura 2.1. A partir desse ponto em diante, destacamos os novos e principais recursos em cada listagem de código, como mostrado nas linhas 9–10 desse programa.

O programa é parecido com Figura 2.1, portanto, aqui discutiremos somente as alterações. A linha 2

```
// Imprimindo uma linha de texto com múltiplas instruções.
```

é um comentário de fim de linha declarando o propósito desse programa. A linha 4 inicia a declaração da classe `Welcome2`. As linhas 9–10 do método `main`:

```
System.out.print( "Welcome to " );
System.out.println( "Java Programming!" );
```

exibem uma linha de texto na janela de comando. A primeira instrução utiliza o método `print` de `System.out` para exibir uma string. Diferente de `println`, depois de exibir seu argumento, `print` não posiciona o cursor de saída no início da próxima linha na janela de comando — o próximo caractere que o programa exibe aparecerá imediatamente depois do último caractere que `print` exibe. Portanto, a linha 10 posiciona o primeiro caractere no seu argumento (a letra "`J`") imediatamente depois do último caractere que a linha 9 exibe (o caractere de espaço em branco antes da aspa dupla de fechamento da string). Cada instrução `print` ou `println` retoma a exibição dos caracteres a partir de onde a última instrução `print` ou `println` parou de exibir os caracteres.

```
1  // Figura 2.3: Welcome2.java
2  // Imprimindo uma linha de texto com múltiplas instruções.
3
4  public class Welcome2
5  {
6     // método principal inicia a execução do aplicativo Java
7     public static void main( String[] args )
8     {
9        System.out.print( "Welcome to " );
10       System.out.println( "Java Programming!" );
11    } // fim do método main
12 } // fim da classe Welcome2
```

```
Welcome to Java Programming!
```

Figura 2.3 | Imprimindo uma linha de texto com múltiplas instruções.

Exibindo múltiplas linhas de texto com uma única instrução

Uma única instrução pode exibir múltiplas linhas utilizando **caracteres de nova linha**, os quais indicam aos métodos `print` e `println` de `System.out` quando posicionar o cursor de saída no começo da próxima linha na janela de comando. Como ocorre com linhas em branco, caracteres de espaço em branco e caracteres de tabulação, os caracteres de nova linha são caracteres de espaço em branco. A Figura 2.4 exibe quatro linhas de texto, utilizando caracteres de nova linha para determinar quando iniciar cada nova linha. A maior parte do programa é idêntica àquele das Figura 2.1 e Figura 2.3, portanto, aqui discutiremos somente as alterações.

```
1  // Figura 2.4: Welcome3.java
2  // Imprimindo múltiplas linhas de texto com uma única instrução.
3
```

```
4  public class Welcome3
5  {
6     // método principal inicia a execução do aplicativo Java
7     public static void main( String[] args )
8     {
9        System.out.println( "Welcome\n to \n Java\n Programming!" );
10    } // fim do método main
11 } // fim da classe Welcome3
```

```
Welcome
to
Java
Programming!
```

Figura 2.4 | Imprimindo múltiplas linhas de texto com uma única instrução.

A linha 2

```
// Imprimindo múltiplas linhas de texto com uma única instrução.
```

é um comentário que declara o objetivo do programa. A linha 4 inicia a declaração da classe `Welcome3`.

A linha 9

```
System.out.println( "Welcome\nto\nJava\nProgramming!" );
```

exibe quatro linhas separadas de texto na janela de comando. Normalmente, os caracteres em uma string são exibidos exatamente como aparecem entre as aspas duplas. Observe, porém, que os dois caracteres `\` e `n` (repetidos três vezes na instrução) não aparecem na tela. A **barra invertida** (\) é chamada **caractere de escape**. Ela indica para os métodos `print` e `println` de `System.out` que a saída de um "caractere especial" deve ser gerada. Quando aparece uma barra invertida em uma string de caracteres, o Java combina o próximo caractere com as barras invertidas para formar uma **sequência de escape.** A sequência de escape `\n` representa o caractere de nova linha. Quando um caractere de nova linha aparece em uma string sendo enviada para saída com `System.out`, o caractere de nova linha faz o cursor de saída na tela mover-se para o começo da próxima linha na janela de comando. A Figura 2.5 lista várias sequências de escape comuns e descreve como elas afetam a exibição de caracteres na janela de comando. Para obter a lista completa de sequências de escape, visite `java.sun.com/docs/books/jls/third_edition/html/lexical.html#3.10.6`.

Sequência de escape	Descrição
`\n`	Nova linha. Posiciona o cursor de tela no início da próxima linha.
`\t`	Tabulação horizontal. Move o cursor de tela para a próxima parada de tabulação.
`\r`	Retorno de carro. Posiciona o cursor da tela no início da linha atual — não avança para a próxima linha. Qualquer saída de caracteres depois do retorno de carro sobrescreve a saída de caracteres anteriormente gerados na linha atual.
`\\`	Barras invertidas. Utilizada para imprimir um caractere de barra invertida.

`\"`	Aspas duplas. Utilizada para imprimir um caractere de aspas duplas. Por exemplo,
	`System.out.println( "\"in quotes\"" );`
	exibe
	`"in quotes"`

Figura 2.5 | Algumas sequências de escape comuns.

2.4 Exibindo texto com `printf`

O método **`System.out.printf`** (`f` significa "formatted") exibe os dados formatados. A Figura 2.6 utiliza esse método para gerar a saída das strings `"Welcome to"` e `"Java Programming!"`.

As linhas 9–10:

```
System.out.printf( "%s\n%s\n",
   "Welcome to", "Java Programming!" );
```

chamam o método `System.out.printf` para exibir a saída do programa. A chamada de método especifica três argumentos. Quando um método exige múltiplos argumentos, estes são colocados em uma **lista separada por vírgulas**.

```
1  // Figura 2.6: Welcome4.java
2  // Exibindo múltiplas linhas com o método System.out.printf.
3
4  public class Welcome4
5  {
6     // método principal inicia a execução do aplicativo Java
7     public static void main( String[] args )
8     {
9        System.out.printf( "%s\n%s\n",
10          "Welcome to", "Java Programming!" );
11    } // fim do método main
12 } // fim da classe Welcome4
```

```
Welcome to
Java Programming!
```

Figura 2.6 | Exibindo múltiplas linhas com o método `System.out.printf`.

Boa prática de programação 2.8

Coloque um espaço depois de cada vírgula (,) em uma lista de argumentos para tornar os programas mais legíveis.

As linhas de 9–10 representam somente uma instrução. O Java permite que instruções grandes sejam divididas em muitas linhas. Recuamos a linha 10 para indicar que é uma continuação da linha 9. Observe que você não pode dividir uma instrução no meio de um identificador ou no meio de uma string.

Erro comum de programação 2.6

Dividir uma instrução no meio de um identificador ou de uma string é um erro de sintaxe.

O primeiro argumento do método `printf` é uma **string de formato** que pode consistir em **texto fixo** e **especificadores de formato**. A saída do texto fixo é gerada por `printf` exatamente como seria gerada por `print` ou `println`. Cada especificador de formato é um marcador de lugar para um valor e especifica o tipo da saída de dados. Especificadores de formato também podem incluir informações opcionais de formatação.

Especificadores de formato iniciam com um sinal de porcentagem (%) e são seguidos por um caractere que representa o tipo de dados. Por exemplo, o especificador de formato **`%s`** é um marcador de lugar para uma string. A string de formato na linha 9 especifica que `printf` deve gerar a saída de duas strings, cada uma seguida por um caractere de nova linha. Na primeira posição do especificador de formato, `printf` substitui o valor do primeiro argumento depois da string de formato. Em cada posição subsequente, `printf` substitui o valor do próximo argumento na lista de argumentos. Portanto, esse exemplo coloca `"Welcome to"` no lugar do primeiro `%s` e `"Java Programming!"` no lugar do segundo `%s`. A saída mostra que duas linhas de texto são exibidas.

Introduzimos vários recursos de formatação uma vez que eles são necessários nos nossos exemplos. O apêndice G apresenta os detalhes da formatação da saída com `printf`.

2.5 Outro aplicativo: somando inteiros

Nosso próximo aplicativo lê (ou insere) dois **inteiros** (números integrais, como –22, 7, 0 e 1024) digitados por um usuário no teclado, calcula a soma dos valores e exibe o resultado. Esse programa deve manter um registro dos números fornecidos pelo usuário para o cálculo mais tarde no programa. Os programas lembram-se dos números e outros dados na memória do computador e acessam esses dados por meio de elementos de programa chamados **variáveis**. O programa da Figura 2.7 demonstra esses conceitos. Na saída de exemplo, usamos o texto em negrito para identificar a entrada do usuário (isto é, **`45`** e **`72`**).

```
1  // Figura 2.7: Addition.java
2  // Programa de adição que exibe a soma de dois números.
3  import java.util.Scanner;  // programa utiliza a classe Scanner
4
5  public class Addition
```

```
6   {
7      // método principal inicia a execução do aplicativo Java
8      public static void main( String[] args )
9      {
10        // cria um Scanner para obter entrada da janela de comando
11        Scanner input = new Scanner( System.in );
12
13        int number1; // primeiro número a adicionar
14        int number2; // segundo número a adicionar
15        int sum; // soma de number1 e number2
16
17        System.out.print( "Enter first integer: " ); // prompt
18        number1 = input.nextInt(); // lê primeiro o número fornecido pelo usuário
19
20        System.out.print( "Enter second integer: " ); // prompt
21        number2 = input.nextInt(); // lê o segundo número fornecido pelo usuário
22
23        sum = number1 + number2; // soma os números, depois armazena o total em sum
24
25        System.out.printf( "Sum is %d\n", sum ); // exibe a soma
26     } // fim do método main
27  } // fim da classe Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Figura 2.7 | O programa de adição que exibe a soma de dois números.

As linhas 1–2

```
// Figura 2.7: Addition.java
// Programa de adição que exibe a soma de dois números.
```

declaram o número da figura, o nome do arquivo e o propósito do programa. A linha 3

```
import java.util.Scanner; // programa utiliza a classe Scanner
```

é uma **declaração `import`** que ajuda o compilador a localizar uma classe utilizada nesse programa. Um dos pontos fortes do Java é seu rico conjunto de classes predefinidas que você pode reutilizar em vez de "reinventar a roda". Essas classes são agrupadas em **pacotes** — chamados de grupos de classes relacionadas — e, coletivamente, são chamadas de **biblioteca de classes Java**, ou **Java Application Programming Interface** (**Java API**). Utilize declarações `import` para identificar as classes predefinidas utilizadas em um programa Java. A declaração `import` na linha 3 indica que esse exemplo usa a classe `Scanner` predefinida do Java (discutida a seguir) do pacote **`java.util`**.

Erro comum de programação 2.7

Todas as declarações `import` devem aparecer antes da primeira declaração da classe no arquivo. Colocar uma declaração `import` dentro do corpo de uma declaração de classe ou depois de uma declaração de classe é um erro de sintaxe.

Dica de prevenção de erro 2.8

Em geral, esquecer-se de incluir uma declaração `import` para uma classe utilizada no seu programa resulta em um erro de compilação contendo uma mensagem como "`cannot find symbol`". Quando isso ocorre, verifique se você forneceu as declarações `import` adequadas e que os nomes nas declarações `import` estão escritos corretamente, incluindo a utilização adequada de letras maiúsculas e minúsculas.

A linha 5

```
public class Addition
```

inicia a declaração da classe `Addition`. O nome de arquivo para essa classe `public` deve ser `Addition.java`. Lembre-se de que o corpo de cada declaração de classe inicia com uma chave esquerda de abertura (linha 6) e termina com uma chave direita de fechamento (linha 27).

O aplicativo inicia a execução com o método `main` (linhas 8–26). A chave esquerda (linha 9) marca o início do corpo de `main` e a correspondente direita (linha 26) marca o final do corpo de `main`. Observe que o método `main` está recuado um nível no corpo da classe `Addition` e que o código no corpo de `main` está recuado um outro nível para legibilidade.

A linha 11

```
Scanner input = new Scanner( System.in );
```

é uma **instrução de declaração de variável** que especifica o nome (`input`) e o tipo (`Scanner`) de uma variável que é usada nesse programa. Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para uso posterior em um programa. Todas as variáveis Java devem ser declaradas com um **nome** e um **tipo** antes que elas possam ser utilizadas. O nome de uma variável permite que o programa acesse o valor da variável na memória. O nome de uma variável pode ser qualquer identificador válido. (Os requisitos de atribuição de nomes de identificadores são fornecidos na Seção 2.2.) O tipo de uma variável especifica o tipo de informações armazenado nessa posição na memória. Como ocorre com outras instruções, as instruções de declaração terminam com um ponto-e-vírgula (`;`).

A declaração na linha 11 especifica que a variável nomeada `input` seja do tipo `Scanner`. Um **`Scanner`** permite a um programa ler os dados (por exemplo, números) para utilização em um programa. Os dados podem ser provenientes de várias origens, como os digitados pelo usuário ou um arquivo do disco. Antes de utilizar um `Scanner`, você deve criá-lo e especificar a origem dos dados.

O sinal de igual (`=`) na linha 11 indica que a variável `Scanner input` deve ser **inicializada** (isto é, preparada para utilização no programa) na sua declaração com o resultado da expressão `new Scanner(System.in)` à direita do sinal de igual. Essa expressão utiliza a palavra-chave **`new`** para criar um objeto `Scanner` que lê caracteres digitados pelo usuário no teclado. O **objeto de entrada padrão**, **`System.in`**, permite que aplicativos leiam bytes de informações digitadas pelo usuário. O objeto `Scanner` traduz esses bytes em tipos (como `int`s) que podem ser utilizados em um programa.

As instruções de declaração de variável nas linhas 13–15:

```
int number1; // primeiro número a adicionar
int number2; // segundo número a adicionar
int sum; // soma de number1 e number2
```

declaram que as variáveis `number1`, `number2` e `sum` armazenam dados do tipo **`int`** — elas podem armazenar valores inteiros (números inteiros como `72`, `-1127` e `0`). Essas variáveis ainda não são inicializadas. O intervalo de valores para um `int` é –2.147.483.648 a +2.147.483.647. [*Nota:* valores `int` reais podem não conter vírgulas.] A seguir, discutiremos os tipos **`float`** e **`double`**, para armazenar números reais, e o tipo **`char`**, para armazenar dados de caracteres. Os números reais contêm pontos decimais, como `3.4`, `0.0` e `-11.19`. Variáveis do tipo `char` representam caracteres individuais, como uma letra maiúscula (por exemplo, `A`), um dígito (por exemplo, `7`), um caractere especial (por exemplo, `*` ou `%`) ou uma sequência de escape (por exemplo, o caractere de nova linha, `\n`). Os tipos `int`, `float`, `double` e `char` são chamados de **tipos primitivos**. Os nomes dos tipos primitivos são palavras-chave e devem aparecer em letras minúsculas. O Apêndice D resume as características dos oito tipos primitivos (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`).

Diversas variáveis do mesmo tipo podem ser declaradas em uma declaração com os nomes de variável separados por vírgulas (isto é, uma lista separada por vírgulas de nomes de variáveis). Por exemplo, as linhas 13–15 também podem ser escritas como uma única instrução:

```
int number1, // primeiro número a adicionar
    number2, // segundo número a adicionar
    sum;     // soma de number1 e number2
```

Boa prática de programação 2.9

Declare cada variável em uma linha separada. Esse formato permite que um comentário descritivo seja facilmente inserido ao lado de cada declaração.

Boa prática de programação 2.10

Escolher nomes de variáveis significativos ajuda um programa ser **autodocumentado** *(isto é, pode-se entender o programa simplesmente lendo-o em vez de ler manuais ou visualizar um número excessivo de comentários).*

Boa prática de programação 2.11

Por convenção, identificadores de nomes de variáveis iniciam com uma letra minúscula e cada palavra no nome depois da primeira palavra inicia com uma letra maiúscula. Por exemplo, o identificador de nome da variável `firstNumber` inicia a sua segunda palavra, `Number`, com uma letra `N` maiúscula.

A linha 17

```
System.out.print( "Enter first integer: " ); // prompt
```

utiliza `System.out.print` para exibir a mensagem `"Enter first integer:"`. Essa mensagem é chamada **prompt** porque direciona o usuário para uma ação específica. Utilizamos o método `print` aqui em vez de `println` para que a entrada do usuário apareça na mesma linha que o prompt. Lembre-se de que na Seção 2.2 esses identificadores que iniciam com letras maiúsculas representam em geral nomes de classe. Portanto, `System` é uma classe. A classe `System` faz parte do pacote **`java.lang`**. Observe que a classe `System` não é importada com uma declaração `import` no começo do programa.

Observação de engenharia de software 2.1

Por padrão, o pacote `java.lang` é importado em cada programa Java; portanto, `java.lang` é o único na Java API que não requer uma declaração `import`.

A linha 18

```
number1 = input.nextInt(); // lê o primeiro número fornecido pelo usuário
```

utiliza o método `nextInt` do valor de `input` do objeto `Scanner` para obter um inteiro digitado pelo usuário. Nesse momento o programa espera que o usuário digite o número e pressione a tecla *Enter* para submeter o número para o programa.

Nosso programa assume que o usuário insere um valor válido de inteiro. Se não for válido, um erro de lógica em tempo de execução ocorrerá e o programa terminará. O Capítulo 11 discute como tornar seus programas mais robustos permitindo que tratem esses erros. Isso também é conhecido como tornar seu programa tolerante a falhas.

Na linha 18, o resultado da chamada ao método `nextInt` (um valor `int`) é colocado na variável `number1` utilizando o **operador de atribuição**, `=`. A instrução exibe "`number1` gets the value of `input.nextInt()`." O operador `=` é chamado de **operador binário**, porque tem dois **operandos** — `number1` e o resultado da chamada de método `input.nextInt()`. Essa instrução inteira é chamada instrução de atribuição porque atribui um valor a uma variável. Tudo à direita do operador de atribuição, `=`, sempre é avaliado antes de a atribuição ser realizada.

Boa prática de programação 2.12

Colocar espaços em qualquer um dos lados de um operador binário faz com que eles se destaquem e torna o programa mais legível.

A linha 20

```
System.out.print( "Enter second integer: " ); // prompt
```

pede para o usuário inserir o segundo inteiro.

A linha 21

```
number2 = input.nextInt(); // lê o segundo número fornecido pelo usuário
```

lê o segundo inteiro e o atribui à variável `number2`.

A linha 23

```
sum = number1 + number2; // soma os números e, então, armazena o total em sum
```

é uma instrução de atribuição que calcula a soma das variáveis `number1` e `number2` e, então, atribui o resultado à variável `sum` utilizando operador de atribuição, `=`. A instrução é lida como "`sum` obtém o valor de `number1 + number2`". Em geral, os cálculos são realizados em instruções de atribuição. Ao encontrar a operação de adição, o programa usa os valores armazenados nas variáveis `number1` e `number2` para fazer o cálculo. Na instrução anterior, o operador de adição é um operador binário — seus dois operandos são as variáveis `number1` e `number2`. As partes das instruções que contêm cálculos são chamadas de **expressões**. De fato, uma expressão é qualquer parte de uma instrução que tem um valor associado com ela. Por exemplo, o valor da expressão `number1 + number2` é a soma dos números. Da mesma forma, o valor da expressão `input.nextInt()` é um inteiro digitado pelo usuário.

Depois que o cálculo foi realizado, a linha 25

```
System.out.printf( "Sum is %d\n", sum ); // exibe a soma
```

utiliza o método `System.out.printf` para exibir a `sum`. O especificador de formato **`%d`** é um marcador de lugar para um valor `int` (nesse caso o valor de `sum`) — a letra `d` significa "inteiro decimal". Observe que além do especificador de formato `%d`, todos os caracteres restantes na string de formato são texto fixo. Portanto, o método `printf` exibe "`Sum is `", seguido pelo valor de `sum` (na posição do especificador de formato `%d`) e por uma nova linha.

Observe que os cálculos também podem ser realizados dentro de instruções `printf`. Poderíamos ter combinado as instruções nas linhas 23 e 25 na instrução:

```
System.out.printf( "Sum is %d\n", ( number1 + number2 ) );
```

Os parênteses em torno da expressão `number1 + number2` não são necessários — são incluídos para enfatizar que o valor da saída da expressão inteira é gerado na posição do especificador de formato `%d`.

Documentação da Java API

Para cada nova classe da Java API que usamos, indicamos o pacote em que ela está localizada. Essas informações ajudam a localizar descrições de cada pacote e classe na documentação da Java API. Uma versão baseada na Web dessa documentação pode ser obtida em:

```
java.sun.com/javase/6/docs/api/
```

Você pode fazer download dessa documentação em:

```
java.sun.com/javase/downloads
```

O Apêndice E mostra como utilizar essa documentação.

2.6 Conceitos de memória

Os nomes de variável como `number1`, `number2` e `sum` realmente correspondem às posições na memória do computador. Cada variável tem um **nome,** um **tipo**, um **tamanho** (em bytes) e um **valor**.

No programa de adição da Figura 2.7, quando a instrução seguinte (linha 18) executa:

```
number1 = input.nextInt(); // lê o primeiro número fornecido pelo usuário
```

o número digitado pelo usuário é colocado em uma localização de memória correspondente ao nome `number1`. Suponha que o usuário insira `45`. O computador coloca esse valor do tipo inteiro na localização `number1` (Figura 2.8), substituindo o valor anterior (se houver algum) nessa localização. O valor anterior é perdido.

Figura 2.8 | Posição da memória mostrando o nome e valor da variável `number1`.

Quando a instrução (linha 21),

```
number2 = input.nextInt(); // lê o segundo número fornecido pelo usuário
```

é executada, suponha que o usuário insira `72`. O computador coloca esse valor do tipo inteiro na localização `number2`. A memória agora aparece como mostrado na Figura 2.9.

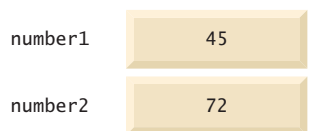


Figura 2.9 | As posições de memória depois de armazenar os valores para `number1` e `number2`.

Depois que o programa da Figura 2.7 obtém os valores para `number1` e `number2`, ele soma os valores e coloca o total na variável `sum`. A instrução (linha 23),

```
sum = number1 + number2; // soma os números, depois armazena o total em sum
```

realiza a soma e então substitui qualquer valor anterior de `sum`. Depois que a variável `sum` foi calculada, a memória aparece conforme mostrado na Figura 2.10. Observe que os valores de `number1` e `number2` aparecem exatamente como apareceram antes de serem utilizados no cálculo de `sum`. Esses valores foram utilizados, mas não destruídos, enquanto o computador realizou o cálculo. Portanto, quando um valor é lido de uma posição da memória, o processo é do tipo não destrutivo.

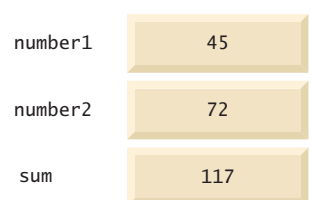


Figura 2.10 | As posições da memória depois de armazenar a soma de `number1` e `number2`.

2.7 Aritmética

A maioria dos programas realiza cálculos aritméticos. Os **operadores aritméticos** são resumidos na Figura 2.11. Note o uso de vários símbolos especiais não utilizados na álgebra. O **asterisco** (`*`) indica **multiplicação**, e o sinal de porcentagem (`%`) é o **operador de resto** (módulo), que discutiremos a seguir. Os operadores aritméticos na Figura 2.11 são operadores binários porque cada um deles opera em dois operandos. Por exemplo, a expressão `f + 7` contém o operador binário `+` e os dois operandos `f` e `7`.

Operação Java	Operador	Expressão algébrica	Expressão Java
Adição	`+`	$f + 7$	`f + 7`
Subtração	`-`	$p - c$	`p - c`
Multiplicação	`*`	bm	`b * m`
Divisão	`/`	x/y ou $\frac{x}{y}$ ou $x \div y$	`x / y`
Resto	`%`	$r \bmod s$	`r % s`

Figura 2.11 | Operadores aritméticos.

A **divisão de inteiros** produz um quociente do tipo inteiro; por exemplo, a expressão `7 / 4` é avaliada como `1` e a expressão `17 / 5` é avaliada como `3`. Qualquer parte fracionária na divisão de inteiros é simplesmente descartada (isto é, truncada) — nenhum arredondamento ocorre. O Java fornece o operador de módulo, `%`, que fornece o resto depois da divisão. A expressão `x % y` produz o restante depois que `x` é dividido por `y`. Portanto, `7 % 4` produz `3` e `17 % 5` produz `2`. Esse operador é mais comumente utilizado com operandos inteiros, mas também pode ser utilizado com outros tipos de aritmética. Nos exercícios deste capítulo e nos capítulos posteriores, vamos examinar vários aplicativos interessantes do operador módulo, como determinar se um número é um múltiplo de outro.

Expressões aritméticas na forma de linha reta

As expressões aritméticas em Java devem ser escritas na **forma de linha reta** para facilitar a inserção de programas no computador. Portanto, as expressões como "`a` dividido por `b`" devem ser escritas como `a / b`, de modo que todas as constantes, variáveis e operadores apareçam em uma linha reta. A seguinte notação algébrica geralmente não é aceitável para compiladores:

$$\frac{a}{b}$$

Parênteses para agrupar subexpressões

Os parênteses são utilizados para agrupar termos em expressões Java da mesma maneira como em expressões algébricas. Por exemplo, para multiplicar `a` vezes a quantidade `b + c` escrevemos:

```
a * ( b + c )
```

Se uma expressão contiver **parênteses aninhados**, como:

```
( ( a + b ) * c )
```

a expressão no conjunto de parênteses mais interno (`a + b` nesse caso) é avaliada primeiro.

Regras de precedência de operadores

O Java aplica os operadores em expressões aritméticas em uma sequência precisa determinada pelas seguintes **regras de precedência de operadores**, que são geralmente as mesmas seguidas em álgebra (Figura 2.12):

1. Operações de multiplicação, divisão e módulo são aplicadas primeiro. Se uma expressão contiver várias dessas operações, elas serão aplicadas da esquerda para a direita. Os operadores de multiplicação, divisão e módulo têm o mesmo nível de precedência.
2. As operações de adição e subtração são aplicadas em seguida. Se uma expressão contiver várias dessas operações, os operadores serão aplicados da esquerda para a direita. Os operadores de adição e subtração têm o mesmo nível de precedência.

Essas regras permitem ao Java aplicar operadores na ordem correta.[1] Quando dizemos que os operadores são aplicados da esquerda para a direita, estamos nos referindo à sua **associatividade**. Alguns operadores associam da direita para a esquerda. A Figura 2.12 resume essas regras de precedência de operador. Um gráfico completo de precedência está incluído no Apêndice A.

[1] Usamos exemplos simples para explicar a ordem da avaliação das expressões. Questões sutis que ocorrem em expressões mais complexas serão vistas mais adiante no livro. Para mais informações sobre a ordem de avaliação, veja o Capítulo 15 da *Java™ Language Specification* (`java.sun.com/docs/books/jls/`).

Operador	Operação	Ordem de avaliação (precedência)
*	Multiplicação	Avaliado primeiro. Se houver vários operadores desse tipo, eles são avaliados da esquerda para a direita.
/	Divisão	
%	Resto	
+	Adição	Avaliado em seguida. Se houver vários operadores desse tipo, eles são avaliados da esquerda para a direita.
–	Subtração	
=	Atribuição	Avaliado por último.

Figura 2.12 | Precedência de operadores aritméticos.

Exemplo de expressões algébricas e Java

Agora vamos considerar várias expressões à luz das regras de precedência de operador. Cada exemplo lista uma expressão algébrica e seu equivalente Java. O seguinte é um exemplo de uma média aritmética de cinco termos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

Java:
```
m = ( a + b + c + d + e ) / 5;
```

Os parênteses são exigidos porque a divisão tem precedência mais alta que a adição. A soma total `(a + b + c + d + e)` será dividida por `5`. Se os parênteses são omitidos erroneamente, obtemos `a + b + c + d + e / 5`, que é avaliado como

$$a + b + c + d + \frac{e}{5}$$

Eis um exemplo da equação de uma linha reta:

Álgebra:
```
y = mx + b
```
Java:
```
y = m * x + b;
```

Nenhum parêntese é requerido. O operador de multiplicação é aplicada primeiro porque a multiplicação tem uma precedência mais alta que a adição. A atribuição ocorre por último porque ela tem uma precedência mais baixa que a multiplicação ou adição.

O seguinte exemplo contém operações de resto (%), multiplicação, divisão, adição e subtração:

Álgebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
```
6 1 2 4 3 5

Os números dentro de círculos sob a instrução indicam a ordem em que o Java aplica os operadores. As operações de multiplicação (*), resto (%) e divisão (/) são avaliadas primeiro na ordem da esquerda para a direita (isto é, elas associam-se da esquerda para a direita), porque têm precedência mais alta que adição (+) e subtração (–). As operações + e – são avaliadas a seguir. Essas operações também são aplicadas da esquerda para a direita. A operação (=) de atribuição é avaliada por último.

Avaliação de um polinômio de segundo grau

Para entender melhor as regras de precedência de operadores, considere a avaliação de uma expressão de atribuição que inclui um polinômio de segundo grau $y = ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
```
6 1 2 4 3 5

Nessa instrução, as operações de multiplicação são avaliadas primeiro na ordem da esquerda para a direita (isto é, elas são associadas da esquerda para a direita), porque têm uma precedência mais alta que a adição. As operações de adição são avaliadas em seguida e são aplicadas da esquerda para a direita. Não há nenhum operador aritmético para exponenciação em Java, portanto x^2 é representado como `x * x`. A Seção 5.4 demonstra uma alternativa para realizar a exponenciação. Suponha que `a`, `b`, `c` e `x` no polinômio de segundo grau anterior seja inicializado (valores dados) como a seguir: `a = 2`, `b = 3`, `c = 7` e `x = 5`. A Figura 2.13 ilustra a ordem em que os operadores são aplicados.

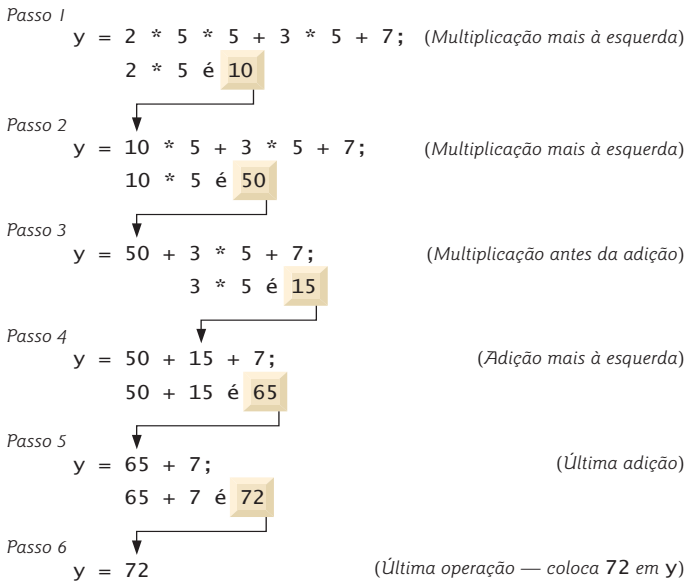


Figura 2.13 | Ordem em que um polinômio de segundo grau é avaliado.

Como na álgebra é aceitável colocar **parênteses redundantes** (parênteses desnecessários) em uma expressão para tornar a expressão mais clara. Por exemplo, a instrução precedente poderia ser posta entre parênteses como mostrado a seguir:

```
y = ( a * x * x ) + ( b * x ) + c;
```

Boa prática de programação 2.13

Utilizar parênteses redundantes em expressões aritméticas complexas pode torná-las mais fáceis de ler.

2.8 Tomada de decisão: operadores de igualdade e operadores relacionais

Uma **condição** é uma expressão que pode ser `true` ou `false`. Esta seção apresenta a **instrução de seleção `if`** do Java que permite a um programa tomar uma **decisão** com base no valor de uma condição. Por exemplo, a condição "nota é maior ou igual a 60" determina se um aluno passou em um teste. Se a condição em uma estrutura `if` for verdadeira, o corpo da estrutura `if` é executada. Se a condição for falsa, o corpo não é executado. Veremos um exemplo brevemente.

As condições nas instruções `if` podem ser formadas utilizando os **operadores de igualdade** (`==` e `!=`) e **operadores relacionais** (`>`, `<`, `>=` e `<=`) resumidos na Figura 2.14. Ambos os operadores de igualdade têm o mesmo nível de precedência, que é mais baixo do que o dos operadores relacionais. Os operadores de igualdade são associados da esquerda para a direita. Todos os operadores relacionais têm o mesmo nível de precedência e também são associados da esquerda para a direita.

Operador de igualdade ou relacional algébrico padrão	Operador de igualdade ou relacional Java	Exemplo de condição em Java	Significado da condição em Java
Operadores de igualdade			
=	==	x == y	x é igual a y
≠	!=	x != y	x é diferente de y
Operadores relacionais			
		y	x é maior que y
<	<	x < y	x é menor que y
≥	>=	x >= y	x é maior que ou igual a y
≤	<=	x <= y	x é menor que ou igual a y

Figura 2.14 | Operadores de igualdade e operadores relacionais.

A Figura 2.15 utiliza seis instruções `if` para comparar duas entradas de inteiros pelo usuário. Se a condição em alguma dessas instruções `if` for verdadeira, a instrução de saída associada com essa instrução `if` é executada; caso contrário, a instrução é pulada. O programa utiliza um `Scanner` para inserir os dois inteiros do usuário e armazena-os nas variáveis `number1` e `number2`. O programa então compara os números e exibe os resultados das comparações que são verdadeiras. Mostramos três saídas de exemplo.

```
1  // Figura 2.15: Comparison.java
2  // Compara inteiros utilizando instruções if, operadores
3  // relacionais e operadores de igualdade.
4  import java.util.Scanner; // programa utiliza a classe Scanner
5
6  public class Comparison
7  {
8     // método principal inicia a execução do aplicativo Java
9     public static void main( String[] args )
10    {
11       // cria Scanner para obter entrada da janela de comando
12       Scanner input = new Scanner( System.in );
13
14       int number1; // primeiro número a comparar
15       int number2; // segundo número a comparar
16
17       System.out.print( "Enter first integer: " ); // prompt
18       number1 = input.nextInt(); // lê o primeiro número fornecido pelo usuário
19
20       System.out.print( "Enter second integer: " ); // prompt
21       number2 = input.nextInt(); // lê o segundo número fornecido pelo usuário
22
23       if ( number1 == number2 )
24          System.out.printf( "%d == %d\n", number1, number2 );
25
26       if ( number1 != number2 )
27          System.out.printf( "%d != %d\n", number1, number2 );
28
29       if ( number1 < number2 )
30          System.out.printf( "%d < %d\n", number1, number2 );
31
32       if ( number1 > number2 )
33          System.out.printf( "%d > %d\n", number1, number2 );
34
35       if ( number1 <= number2 )
36          System.out.printf( "%d <= %d\n", number1, number2 );
37
38       if ( number1 >= number2 )
39          System.out.printf( "%d >= %d\n", number1, number2 );
40    } // fim do método main
41 } // fim da classe Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 1000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Figura 2.15 | Compare números inteiros usando instruções `if`, operadores relacionais e operadores de igualdade (Parte 2 de 2).

A declaração da classe `Comparison` inicia na linha 6:

```
public class Comparison
```

O método `main` da classe (linhas 9–40) inicia a execução do programa. A linha 12

```
Scanner input = new Scanner( System.in );
```

declara a variável `Scanner input` e lhe atribui um `Scanner` que insere dados a partir da entrada padrão (isto é, o teclado).

As linhas 14–15

```
int number1; // primeiro número a comparar
int number2; // segundo número a comparar
```

declaram as variáveis `int` utilizadas para armazenar a entrada dos valores digitados pelo usuário.

As linhas 17–18

```
System.out.print( "Enter first integer: " ); // prompt
number1 = input.nextInt(); // lê o primeiro número fornecido pelo usuário
```

solicitam que o usuário digite o primeiro inteiro e insira o valor, respectivamente. O valor de entrada é armazenado na variável `number1`.

As linhas 20–21

```
System.out.print( "Enter second integer: " ); // prompt
number2 = input.nextInt(); // lê o segundo número fornecido pelo usuário
```

solicitam que o usuário digite o segundo inteiro e insira o valor, respectivamente. O valor de entrada é armazenado na variável `number2`.

As linhas 23–24

```
if ( number1 == number2 )
   System.out.printf( "%d == %d\n", number1, number2 );
```

comparam os valores de `number1` e `number2` para determinar se eles são iguais. Uma estrutura `if` sempre inicia com a palavra-chave `if`, seguida por uma condição entre parênteses. Uma instrução `if` espera uma instrução em seu corpo, mas pode conter múltiplas instruções se elas estiverem entre parênteses (`{}`). O recuo da instrução no corpo mostrado aqui não é exigido, mas melhora a legibilidade do programa enfatizando que a instrução na linha 24 faz parte da estrutura `if` que inicia na linha 23. A linha 24 executa somente se os números armazenados nas variáveis `number1` e `number2` forem iguais (isto é, se a condição for verdadeira). As instruções `if` nas linhas 26–27, 29–30, 32–33, 35–36 e 38–39 comparam `number1` e `number2` com os operadores `!=`, `<`, `>`, `<=` e `>=`, respectivamente. Se a condição em uma das instruções `if` for verdadeira, a instrução no corpo correspondente é executada.

Erro comum de programação 2.8

Esquecer o parêntese esquerdo e/ou direito para a condição em uma estrutura `if` é um erro de sintaxe — os parênteses são requeridos.

Erro comum de programação 2.9

Confundir o operador de igualdade, ==, com o operador de atribuição, =, pode causar um erro de lógica ou um erro de sintaxe. O operador de igualdade deve ser lido como "igual a", e o operador de atribuição deve ser lido como "obtém" ou "obtém o valor de". Para evitar confusão, algumas pessoas leem o operador de igualdade como "duplo igual" ou "igual igual".

Boa prática de programação 2.14

Colocar apenas uma instrução por linha em um programa aprimora a legibilidade do programa.

Observe que não há ponto-e-vírgula (;) no final da primeira linha de cada instrução `if`. Esse ponto-e-vírgula resultaria em um erro de lógica em tempo de execução. Por exemplo,

```
if ( number1 == number2 ); // erro de lógica
   System.out.printf( "%d == %d\n", number1, number2 );
```

na realidade seria interpretado pelo Java como:

```
if ( number1 == number2 )
   ; // estrutura vazia

System.out.printf( "%d == %d\n", number1, number2 );
```

onde o ponto-e-vírgula na linha sozinho — chamado **instrução vazia** — é a instrução a executar se a condição na instrução `if` for verdadeira. Quando a instrução vazia executa, nenhuma tarefa é realizada no programa. O programa então continua com a instrução de saída, que sempre é executada, independentemente de a condição ser verdadeira ou falsa, porque a instrução de saída não faz parte da estrutura `if`.

Erro comum de programação 2.10

Colocar um ponto-e-vírgula imediatamente depois do parêntese direito da condição em uma instrução `if` é normalmente um erro de lógica.

Note o uso do espaço em branco na Figura 2.15. Lembre-se de que, em geral, o espaço em branco é ignorado pelo compilador. Então, as instruções podem ser divididas em várias linhas e podem ser espaçadas de acordo com as suas preferências sem afetar o significado de um programa. Não é correto dividir identificadores e strings. Idealmente, as instruções devem ser mantidas pequenas, mas isso nem sempre é possível.

Boa prática de programação 2.15

Uma instrução longa pode se estender por várias linhas. Se uma única instrução deve ser dividida em várias linhas, escolha dividi-la em pontos que fazem sentido, como depois de uma vírgula em uma lista separada por vírgulas ou depois de um operador em uma expressão longa. Se uma instrução for dividida em duas ou mais linhas, recue todas as linhas subsequentes até o fim da instrução.

A Figura 2.16 mostra os operadores discutidos até agora em ordem decrescente de precedência. Todos esses operadores, com a exceção do operador de atribuição, =, associam da esquerda para a direita. A adição associa da esquerda para a direita, então uma expressão como `x + y + z` é avaliada como se tivesse sido escrita `(x + y) + z`. O operador de atribuição, = , associa da direita para a esquerda, assim uma expressão como `x = y = 0` é avaliada como se tivesse sido escrita `x = (y = 0)`, que, primeiro, atribui o valor `0` à variável `y` e, então, atribui o resultado dessa atribuição, `0`, a `x`.

Operadores	Associatividade	Tipo
`*` `/` `%`	da esquerda para a direita	multiplicativo
`+` `-`	da esquerda para a direita	aditivo
`=`	da esquerda para a direita	relacional
`==` `!=`	da esquerda para a direita	igualdade
`=`	da direita para a esquerda	atribuição

Figura 2.16 | Precedência e associatividade dos operadores discutidos.

Boa prática de programação 2.16

Consulte o gráfico de operador de precedência (Apêndice A) ao escrever expressões contendo muitos operadores. Confirme se as operações na expressão são realizados na ordem em que você espera. Se você não tiver certeza sobre a ordem de avaliação em uma expressão complexa, utilize parênteses para forçar a ordem, exatamente como faria em expressões algébricas.

2.9 Conclusão

Você aprendeu muitos recursos importantes do Java neste capítulo, incluindo exibir dados na tela em uma Prompt de Command, inserir dados pelo teclado, realizar cálculos e tomar decisões. Os aplicativos apresentados aqui foram concebidos como uma introdução aos conceitos básicos de programação.

Como verá no Capítulo 3, em geral aplicativos Java contêm apenas algumas linhas de código no método `main` — essas instruções normalmente criam os objetos que realizam o trabalho do aplicativo. No Capítulo 3, você aprenderá a implementar suas próprias classes e a utilizar objetos dessas classes nos aplicativos.

Resumo

Seção 2.2 Nosso primeiro programa Java: imprimindo uma linha de texto

- Um aplicativo Java é um programa de computador que é executado quando você utiliza o comando `java` para carregar a JVM.
- Os comentários documentam programas e aprimoram sua legibilidade. Eles são ignorados pelo compilador.
- Um comentário que começa com `//` é chamado de comentário de fim de linha — ele termina no fim da linha em que aparece.
- Comentários tradicionais podem se estender por várias linhas e são delimitados por `/*` e `*/`.
- Os comentários da Javadoc, delimitados por `/**` e `*/`, permitem que você incorpore a documentação do programa no código. O programa utilitário `javadoc` gera páginas em HTML com base nesses comentários.
- Um erro de sintaxe (também chamado erro de compilador, erro em tempo de compilação ou erro de compilação) ocorre quando o compilador encontra um código que viola as regras da linguagem do Java. Isso é semelhante a um erro de gramática em uma língua natural.
- Linhas em branco, caracteres de espaço em branco e caracteres de tabulação são conhecidos como espaço em branco. O espaço em branco torna os programas mais fáceis de ler e não é ignorado pelo compilador.
- Todo programa que você cria consiste em pelo menos uma classe definida pelo programador.
- As palavras-chave são reservadas para uso pelo Java e sempre são escritas com todas as letras minúsculas.
- A palavra-chave `class` introduz uma declaração de classe e é imediatamente seguida pelo nome da classe.
- Por convenção, todos os nomes de classes em Java iniciam com uma letra maiúscula e apresentam a letra inicial de cada palavra que eles incluem em maiúscula (por exemplo, `SampleClassName`).
- O nome de uma classe Java é um identificador — uma série de caracteres que consiste em letras, dígitos, sublinhados (`_`) e sinais de cifrão (`$`) que não iniciem com um dígito e não contenham espaços.
- O Java diferencia entre letras maiúsculas e minúsculas — isto é, letras maiúsculas e minúsculas são diferentes.
- O corpo de cada declaração de classe é delimitado por chaves, `{` e `}`.
- Uma declaração da classe `public` deve ser salva em um arquivo com o mesmo nome da classe seguido pela extensão de nome de arquivo "`.java`".
- O método `main` é o ponto de partida de cada aplicativo Java e deve iniciar com:

  ```
  public static void main( String[] args )
  ```

  Caso contrário, a JVM não executará o aplicativo.
- Os métodos realizam tarefas e informações de retorno quando eles completam suas tarefas. A palavra-chave `void` indica que um método realizará uma tarefa, mas não retornará nenhuma informação.
- As instruções fazem o computador realizar ações.
- Uma string entre aspas duplas é às vezes chamada de string de caracteres ou string literal.
- O objeto de saída padrão (`System.out`) exibe caracteres na janela de comando.
- O método `System.out.println` exibe seu argumento na janela de comando seguido por um caractere de nova linha para posicionar o cursor de saída no começo da próxima linha.
- Você compila um programa com o comando `javac`. Se o programa não contiver nenhum erro de sintaxe, um arquivo de classe contendo os bytecodes Java que representam o aplicativo é criado. Esses bytecodes são interpretados pela JVM quando executamos o programa.
- Para executar um aplicativo, digite `java` seguido pelo nome da classe que contém `main`.

Seção 2.3 Modificando nosso primeiro programa Java

- `System.out.print` exibe seu argumento e posiciona o cursor de saída imediatamente após o último caractere exibido.
- Uma barra invertida (`\`) em uma string é um caractere de escape. O Java combina o próximo caractere com as barras invertidas para formar uma sequência de escape. A sequência de escape `\n` representa o caractere de nova linha.

Seção 2.4 Exibindo texto com `printf`

- O método `System.out.printf` (`f` significa "formatado") exibe os dados formatados.
- O primeiro argumento do método `printf` é uma string de formato que pode consistir em texto fixo e especificadores de formato. Cada especificador de formato é um marcador de lugar para um valor e especifica o tipo da saída de dados.
- Especificadores de formato iniciam com um sinal de porcentagem (`%`) e são seguidos por um caractere que representa o tipo de dados. Os especificadores de formato `%s` é um marcador de lugar para uma string.
- Cada especificador de formato é substituído pelo valor do argumento correspondente que aparece depois da string de formato.

Seção 2.5 Outro aplicativo: somando inteiros

- Uma declaração `import` ajuda o compilador a localizar uma classe utilizada em um programa.
- O rico conjunto do Java de classes predefinidas é agrupado em pacotes — chamados de grupos de classes.
- Coletivamente, os pacotes do Java são chamados de biblioteca de classes Java ou Java Application Programming Interface (Java API).
- Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para utilização posterior em um programa. Todas as variáveis devem ser declaradas com um nome e um tipo antes que elas possam ser utilizadas.
- O nome de uma variável permite que o programa acesse o valor da variável na memória.
- Um `Scanner` (pacote `java.util`) permite a um programa ler os dados para uso em um programa. Antes de um `Scanner` poder ser utilizado, o programa deve criá-lo e especificar a origem dos dados.
- Variáveis devem ser inicializadas para prepará-las para uso em um programa.
- A expressão `new Scanner (System.in)` cria um `Scanner` que lê o objeto de entrada padrão (`System.in`) — normalmente o teclado.
- O tipo de dados `int` é utilizado para declarar variáveis que conterão valores de inteiro. O intervalo de valores para um `int` é –2.147.483.648 a +2.147.483.647.
- Os tipos `float` e `double` especificam números reais com pontos decimais, como `3.4` e `–11.19`.
- Variáveis do tipo `char` representam caracteres individuais, como uma letra maiúscula (por exemplo, `A`), um dígito (por exemplo, `7`), um caractere especial (por exemplo, `*` ou `%`) ou uma sequência de escape (por exemplo, nova linha, `\n`).
- Tipos como `int`, `float`, `double` e `char` são tipos primitivos. Os nomes dos tipos primitivos são palavras-chave; portanto, todos devem aparecer em letras minúsculas.
- Um prompt direciona o usuário a tomar uma ação específica.
- O método `Scanner nextInt` obtém um inteiro para uso em um programa.
- O operador de atribuição, `=`, permite ao programa atribuir um valor a uma variável. O operador = é chamado operador binário porque tem dois operandos.
- Partes das instruções que contêm valores são chamadas expressões.
- O especificador de formato `%d` é um marcador de lugar para um valor `int`.

Seção 2.6 Conceitos de memória

- Os nomes de variável correspondem a posições na memória do computador. Cada variável tem um nome, um tipo, um tamanho e um valor.
- Um valor que é colocado em uma posição de memória substitui o valor anterior da posição, que é perdido.

Seção 2.7 Aritmética

- Os operadores aritméticos são `+` (adição), `–` (subtração), `*` (multiplicação), `/` (divisão) e `%` (resto).
- A divisão de inteiros produz um quociente inteiro.
- O operador de módulo, `%`, fornece o resto depois da divisão.
- As expressões aritméticas devem ser escritas em sequência direta.
- Se uma expressão contiver parênteses aninhados, o conjunto mais interno dentro dos parênteses é avaliado primeiro.
- O Java aplica os operadores a expressões aritméticas em uma sequência precisa determinada pelas regras de precedência de operadores.
- Quando dizemos que operadores são aplicados da esquerda para a direita, estamos nos referindo a sua associatividade. Alguns operadores associam da direita para a esquerda.
- Os parênteses redundantes em uma expressão podem tornar uma expressão mais clara.

Seção 2.8 Tomada de decisão: operadores de igualdade e operadores relacionais

- A instrução `if` toma uma decisão baseada no valor de uma condição (verdadeiro ou falso).
- As condições em instruções if podem ser formadas utilizando-se os operadores de igualdade (`==` e `!=`) e relacionais (`>`, `<`, `>=` e `<=`).
- Uma instrução `if` começa com palavra-chave `if`, seguida por uma condição entre parênteses e espera uma instrução no seu corpo.
- A instrução vazia é uma instrução que não realiza nenhuma tarefa.

Terminologia

–, subtração, 41
`/* */`, comentário tradicional, 29
`/`, divisão de inteiros, 41
`%d`, especificador de formato, 39
`%s`, especificador de formato, 36
`<`, operador ("maior que"), 43
`//`, comentário de fim de linha, 29
adição, 41
API (application programming interface), 37
aplicativo, 29
argumento para um método, 32
arquivo de classe, 33
associatividade de operadores, 41
asterisco (`*`), 41
autodocumentado, 38
barras invertidas (`\\`), 35
caractere de escape, 35
`cd`, para mudar de diretório, 33
`char`, tipo primitivo, 38
`class`, palavra-chave, 30
`.class`, arquivo, 33
classe definida pelo programador, 30
código autodocumentado, 38
comentário tradicional (`/* */`), 29
condição, 43
corpo da definição de método, 32
corpo de uma declaração de classe, 37
corpo de uma declaração de método, 32
decisão, 43
declaração de classe, 30
distinção de letras maiúsculas e minúsculas, 30
divisão, 41
divisão de inteiro, 41
documentar programa, 29
`double`, tipo primitivo, 38
erro de compilação, 30
erro de compilador, 30
erro de sintaxe, 30
erro em tempo de compilação, 30
especificador de formato, 36
expressão, 39
`false`, palavra-chave, 43
fim de linha (de uma única linha), comentário, `//`, 29
forma de linha reta (expressões aritméticas em Java), 41
identificador, 30
`if`, instrução de seleção única, 43
`import`, declaração, 37
imprimir uma linha de texto, 32
inicializar uma variável em uma declaração, 38
instrução, 32
instrução de declaração de variável, 38
instrução vazia (um ponto-e-vírgula, `;`), 46
`int`, tipo primitivo, 38
inteiro, 36
janela terminal, 32
`java`, comando, 29
`java.lang`, pacote, 38
Java Application Programming Interface (Java API), 37
`java.util`, pacote, 37
Java, biblioteca de classe, 37
`javadoc`, programa utilitário, 30
linha de comando, 32
lista separada por vírgulas, 36
método, 31
multiplicação, *, operador, 41
`new`, palavra-chave, 38
nome de classe, 30
nome de uma variável, 40
nova linha, caractere (`\l`), 34
objeto de entrada padrão (`System.in`), 38
objeto de saída padrão (`System.out`), 32
operador, 39
operador binário, 39
operador de adição (+), 42
operador de atribuição, =, 39
operador de divisão (`/`), 42
operador de multiplicação (`*`), 42
operador de resto, `%`, 41
operador de subtração (–), 42
operadores aritméticos, 41
operadores de igualdade, 43
operadores relacionais, 43
operando, 39
pacote, 37
palavra-chave, 30
palavra reservada, 30
parênteses, 31
parênteses aninhados, 41
ponto-e-vírgula (`;`), 32
precedência, 41
prompt, 38
`public`, palavra-chave, 31
regras de precedência de operador, 41
resto, 41
`Scanner`, classe, 38
sequência de escape, 35
shell, 32
string, 32
string de caractere, 32
string de formato, 36
subtração, 41
`System.out`, (objeto de saída padrão), 32
`System.out.printf`, método, 35
`System.in`, 38
tamanho de uma variável, 40
texto fixo em uma string de formato, 36
tipo, 38
tipo de uma variável, 40
tipo primitivo, 38
`true`, 43
valor de uma variável, 40
variáveis, 36
`void`, palavra-chave, 31

Exercícios de autorrevisão

2.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um(a) _____ no corpo de cada método e um(a) _____ termina o corpo de cada método.

b) A instrução _____ é utilizada para tomar decisões.

c) _____ em um comentário de fim de linha.

d) _____, _____ e _____ são chamados espaço em branco.

e) _____ são reservados para uso pelo Java.

f) Aplicativos Java iniciam a execução no método _____.

g) Os métodos _____, _____ e _____ exibem informações em uma janela de comando.

2.2 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Os comentários fazem com que o computador imprima o texto depois das `//` na tela quando o programa executa.

b) Todas as variáveis devem ser atribuídas a um tipo quando são declaradas.

c) O Java considera que as variáveis `number` e `NuMbEr` são idênticas.

d) O operador de resto (`%`) pode ser utilizado apenas com operandos inteiros.

e) Os operadores aritméticos `*`, `/`, `%`, `+` e `-` têm, todos, o mesmo nível de precedência.

2.3 Escreva instruções para realizar cada uma das tarefas a seguir:

a) Declare que as variáveis `c`, `thisIsAVariable`, `q76354` e `number` serão do tipo `int`.

b) Solicite que o usuário insira um inteiro.

c) Insira um inteiro e atribua o resultado à variável `int value`. Suponha que variável `Scanner input` possa ser utilizada para ler um valor digitado pelo usuário.

d) Imprima `"This is a Java program"` em uma linha na janela de comando. Utilize o método de uso `System.out.println`.

e) Imprima `"This is a Java program"` em duas linhas na janela de comando. A primeira linha deve terminar com `Java`. Utilize o método `System.out.println`.

f) Imprima `"This is a Java program"` em duas linhas na janela de comando. A primeira linha deve terminar com `Java`. Utilize o método `System.out.printf` e dois especificadores de formato `%s`.

g) Se a variável `number` não for igual a `7`, exiba `"The variable number is not equal to 7"`.

2.4 Identifique e corrija os erros em cada uma das seguintes instruções:

a)
```
if ( c < 7 );
   System.out.println( "c is less than 7" );
```

b)
```
if ( c => 7 )
   System.out.println( "c is equal to or greater than 7" );
```

2.5 Escreva declarações, instruções ou comentários que realizam cada uma das tarefas a seguir:

a) Declare que um programa calculará o produto de três inteiros.

b) Crie um `Scanner` chamado `input` que lê valores a partir da entrada padrão.

c) Declare as variáveis `x`, `y`, `z` e `result` como tipo `int`.

d) Solicite que o usuário insira o primeiro inteiro.

e) Leia o primeiro inteiro digitado pelo usuário e armazene-o na variável `x`.

f) Solicite que o usuário insira o segundo inteiro.

g) Leia o segundo inteiro digitado pelo usuário e armazene-o na variável `y`.

h) Solicite que o usuário insira o terceiro inteiro.

i) Leia o terceiro inteiro digitado pelo usuário e armazene-o na variável `z`.

j) Compute o produto dos três inteiros contidos nas variáveis `x`, `y` e `z` e atribua o resultado à variável `result`.

k) Exiba a mensagem `"Product is"` seguida pelo valor da variável `result`.

2.6 Usando as instruções que você escreveu no Exercício 2.5, escreva um programa completo que calcula e imprime o produto de três inteiros.

Respostas dos exercícios de autorrevisão

2.1 a) Chave esquerda ({), chave direita (}). b) `if`. c) `//`. d) Caracteres de espaço em branco, de nova linha e tabulações. e) Palavras-chave. f) `main`. g) `System.out.print`, `System.out.println` e `System.out.printf`.

2.2 a) Falso. Os comentários não causam nenhuma ação quando o programa executa. Eles são usados para documentar programas e melhorar sua legibilidade.

b) Verdadeiro.

c) Falso. Java diferencia letras maiúsculas de minúsculas, então essas variáveis são distintas.

d) Falso. O operador de resto também pode ser utilizado com operandos não inteiros em Java.

e) Falso. Os operadores `*`, `/` e `%` estão no mesmo nível de precedência e os operadores `+` e `–` estão em um nível mais baixo de precedência.

2.3 a)
```
int c, thisIsAVariable, q76354, number;
```
ou
```
int c;
int thisIsAVariable;
int q76354;
int number;
```

b)
```
System.out.print( "Enter an integer: " );
```

c)
```
value = input.nextInt();
```

d)
```
System.out.println( "This is a Java program" );
```

e)
```
System.out.println( "This is a Java\nprogram" );
```

f)
```
System.out.printf( "%s\n%s\n", "This is a Java", "program" );
```

g)
```
if ( number != 7 )
   System.out.println( "The variable number is not equal to 7" );
```

2.4 a) Erro: O ponto-e-vírgula depois do parêntese direito da condição `( c < 7 )` no `if`. Correção: remova o ponto-e-vírgula depois do parêntese direito. [*Nota:* como resultado, a instrução de saída executará independentemente de a condição em `if` ser verdadeira].

b) Erro: `=>`O operador relacional => é incorreto. Correção: altere => para >=.

2.5 a) `// Calcula o produto de três inteiros`

b)
```
Scanner input = new Scanner( System.in );
```

c)
```
int x, y, z, result;
```
ou
```
int x;
int y;
int z;
int result;
```

d)
```
System.out.print( "Enter first integer: " );
```

e)
```
x = input.nextInt();
```

f)
```
System.out.print( "Enter second integer: " );
```

g)
```
y = input.nextInt();
```

h)
```
System.out.print( "Enter third integer: " );
```

i)
```
z = input.nextInt();
```

j)
```
result = x * y * z;
```

k)
```
System.out.printf( "Product is %d\n", result );
```

2.6 A solução para o Exercício de autorrevisão 2.6 é a seguinte:

```
1  // Ex. 2.6: Product.Java
2  // Calcula o produto de três inteiros.
3  import java.util.Scanner; // programa utiliza Scanner
4
5  public class Product
6  {
7     public static void main( String[] args )
8     {
9        // cria Scanner para obter entrada a partir da janela de comando
10       Scanner input = new Scanner( System.in );
11
12       int x; // primeiro número inserido pelo usuário
13       int y; // segundo número inserido pelo usuário
14       int z; // terceiro número inserido pelo usuário
15       int result; // produto dos números
16
17       System.out.print( "Enter first integer: " ); // solicita entrada
18       x = input.nextInt(); // lê o primeiro inteiro
19
20       System.out.print( "Enter second integer: " ); // solicita entrada
21       y = input.nextInt(); // lê o segundo inteiro
22
23       System.out.print( "Enter third integer: " ); // solicita entrada
24       z = input.nextInt(); // lê o terceiro inteiro
25
26       result = x * y * z; // calcula o produto dos números
27
28       System.out.printf( "Product is %d\n", result );
29    } // fim do método main
30 } // fim da classe Product
```

```
Enter first integer: 10
Enter second integer: 20
Enter third integer: 30
Product is 6000
```

Exercícios

2.7 Preencha as lacunas em cada uma das seguintes afirmações:

a) _____ são utilizados para documentar um programa e aprimorar sua legibilidade.

b) Uma decisão pode ser tomada em um programa Java com um(a) _____.

c) Os cálculos normalmente são realizados pelas instruções _____.

d) Os operadores aritméticos com a mesma precedência da multiplicação são _____ e _____.

e) Quando parênteses em uma expressão aritmética estão aninhados, o conjunto de parênteses _____ é avaliado primeiro.

f) Uma posição na memória do computador que pode conter valores diferentes várias vezes ao longo da execução de um programa é chamada _____.

2.8 Escreva instruções Java que fazem cada uma das seguintes tarefas:

a) Exiba a mensagem `"Enter an integer: "`, deixando o cursor na mesma linha.

b) Atribui o produto de variáveis `b` e `c` para a variável `a`.

c) Declara que um programa realiza um cálculo de folha de pagamento de exemplo (isto é, usa texto que ajuda a documentar um programa).

2.9 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Operadores Java são avaliados da esquerda para a direta.

b) Os seguintes são todos nomes de variável válidos: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his_$account_total`, `a`, `b$`, `c`, `z` e `z2`.

c) Uma expressão aritmética Java válida sem parênteses é avaliada da esquerda para a direta.

d) Os seguintes são todos nomes de variável inválidos: `3g`, `87`, `67h2`, `h22` e `2h`.

2.10 Supondo que `x = 2` e `y = 3`, o que cada um destas instruções exibe?

a) `System.out.printf( "x = %d\n", x );`

b) `System.out.printf( "Value of %d + %d is %d\n", x, x, ( x + x ) );`

c) `System.out.printf( "x =" );`

d) `System.out.printf( "%d = %d\n", ( x + y ), ( y + x ) );`

2.11 Quais instruções Java a seguir contêm variáveis cujos valores são modificados?

a) `p = i + j + k + 7;`

b) `System.out.println( "variables whose values are modified" );`

c) `System.out.println( "a = 5" );`

d) `value = input.nextInt();`

2.12 Dado que $y = ax^3 + 7$, quais das seguintes são instruções Java corretas para essa equação?

a) `y = a * x * x * x + 7;`

b) `y = a * x * x * ( x + 7 );`

c) `y = ( a * x ) * x * ( x + 7 );`

d) `y = ( a * x ) * x * x + 7;`

e) `y = a * ( x * x * x ) + 7;`

f) `y = a * x * ( x * x + 7 );`

2.13 Declare a ordem de avaliação dos operadores em cada uma das seguintes instruções Java e mostre o valor de `x` após cada instrução ser realizada:

a) `x = 7 + 3 * 6 / 2 - 1;`

b) `x = 2 % 2 + 2 * 2 - 2 / 2;`

c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

2.14 Escreva um aplicativo que exibe os números 1 a 4 na mesma linha, com cada par de números adjacentes separados por um espaço. Escreva o programa utilizando as técnicas a seguir:

a) Utilize uma instrução `System.out.println`.

b) Utilize quatro instruções `System.out.print`.

c) Utilize uma instrução `System.out.printf`.

2.15 (***Aritmética***) Escreva um aplicativo que solicita ao usuário inserir dois inteiros, obtém do usuário esses números e imprime sua soma, produto, diferença e quociente (divisão). Utilize as técnicas mostradas na Figura 2.7.

2.16 (***Comparando inteiros***) Escreva um aplicativo que solicita ao usuário inserir dois inteiros, obtém do usuário esses números e exibe o número maior seguido pelas palavras `"is larger"`. Se os números forem iguais, imprime a mensagem `"These numbers are equal"`. Utilize as técnicas mostradas na Figura 2.15.

2.17 (***Aritmética, menor e maior***) Escreva um aplicativo que insere três inteiros digitados pelo usuário e exibe a soma, média, produto e os números menores e maiores. Utilize as técnicas mostradas na Figura 2.15. [*Nota:* o cálculo da média neste exercício deve resultar em uma representação de inteiro da média. Assim, se a soma dos valores for 7, a média deverá ser 2, não 2,3333....]

2.18 (***Exibindo formas com asteriscos***) Escreva um aplicativo que exibe uma caixa, uma oval, uma seta e um losango utilizando asteriscos (*), como segue:

```
*********     ***       *         *
*       *   *     *    ***       * *
*       *  *       *  *****     *   *
*       *  *       *    *      *     *
*       *  *       *    *     *       *
*       *  *       *    *      *     *
*       *  *       *    *       *   *
*       *   *     *     *        * *
*********     ***       *         *
```

2.19 O que o seguinte código imprime?

```
System.out.println( "*\n**\n***\n****\n*****" );
```

2.20 O que o seguinte código imprime?

```
System.out.println( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.println( "****" );
System.out.println( "**" );
```

2.21 O que o seguinte código imprime?

```
System.out.print( "*" );
System.out.print( "***" );
System.out.print( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.22 O que o seguinte código imprime?

```
System.out.print( "*" );
System.out.println( "***" );
System.out.println( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

2.23 O que o seguinte código imprime?

```
System.out.printf( "%s\n%s\n%s\n", "*", "***", "*****" );
```

2.24 (***Maiores e menores inteiros***) Escreva um aplicativo que lê cinco inteiros, determina e imprime o maior e o menor inteiro no grupo. Utilize somente as técnicas de programação que você aprendeu neste capítulo.

2.25 (***Ímpar ou par***) Escreva um aplicativo que lê um inteiro, determina e imprime se ele é ímpar ou par. [*Dica:* utilize o operador de módulo. Um número par é um múltiplo de 2. Qualquer múltiplo de 2 deixa um resto 0 quando dividido por 2.]

2.26 (***Múltiplos***) Escreva um aplicativo que lê dois inteiros, determina se o primeiro é um múltiplo do segundo e imprime o resultado. [*Dica:* utilize o operador de módulo].

2.27 (***Padrão de tabuleiro de damas de asteriscos***) Escreva um aplicativo que exibe um padrão de tabuleiro de damas, conforme mostrado a seguir:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 (***Diâmetro, circunferência e área de um círculo***) Eis uma prévia do que veremos mais adiante. Neste capítulo, você aprendeu sobre inteiros e o tipo `int`. O Java também pode representar números de pontos flutuantes que contêm pontos de fração decimal, como 3.14159. Escreva um aplicativo que lê a entrada a partir do usuário do raio de um círculo como um inteiro e imprime o diâmetro do círculo, circunferência e área utilizando o valor do ponto flutuante 3.14159 para π. Utilize as técnicas mostradas na Figura 2.7 [*Nota:* você também pode utilizar a constante `Math.PI` predefinida para o valor de π. Essa constante é mais precisa que o valor 3,14159. A classe `Math` é definida no pacote `java.lang`. As classes nesse pacote são importadas automaticamente, portanto, você não precisa `importar` a classe `Math` para utilizá-la.] Utilize as seguintes fórmulas (*r* é o raio):

diameter = 2*r*
circumference = 2π*r*
area = πr^2

Não armazene os resultados de cada cálculo em uma variável. Em vez disso, especifique cada cálculo como o valor de saída em uma instrução `System.out.printf`. Observe que os valores produzidos pela circunferência e os cálculos da área são números de ponto flutuante. A saída desses

valores pode ser gerada com o especificador de formato `%f` em uma instrução `System.out.printf`. Você aprenderá mais sobre números de pontos flutuantes no Capítulo 3.

2.29 **(*O valor inteiro de um caractere*)** Eis outra prévia do que virá adiante. Neste capítulo, você aprendeu sobre inteiros e o tipo `int`. O Java também pode representar letras maiúsculas, minúsculas e uma variedade considerável de símbolos especiais. Cada caractere tem uma representação correspondente de inteiro. O conjunto de caracteres que um computador utiliza junto com as correspondentes representações na forma de inteiro desses caracteres é chamado conjunto de caracteres desse computador. Você pode indicar um valor de caractere em um programa simplesmente incluindo esse caractere entre aspas simples, como em `'A'`.

Você pode determinar o equivalente em inteiro de um caractere precedendo esse caractere (com `int`), como em:

```
(int) 'A'
```

Essa forma é chamada operador de coerção. (Você aprenderá sobre os operadores de coerção no Capítulo 4.) A instrução a seguir gera saída de um caractere e seu equivalente de inteiro:

```
System.out.printf(
   "The character %c has the value %d\n", 'A', ( (int) 'A' ) );
```

Quando a instrução precedente executa, ela exibe o caractere `A` e o valor `65` (do conjunto de caracteres Unicode®) como parte da string. Observe que o especificador de formato `%c` é um marcador de lugar para um caractere (nesse caso, o caractere `'A'`).

Utilizando instruções semelhantes àquela mostrada anteriormente neste exercício, escreva um aplicativo que exibe os equivalentes inteiros de algumas letras maiúsculas, letras minúsculas, dígitos e símbolos especiais. Exiba os equivalentes inteiros do seguinte: `A B C a b c 0 1 2 $ * + /` e o caractere em branco.

2.30 **(*Separando os dígitos em um inteiro*)** Escreva um aplicativo que insere um número consistindo em cinco dígitos do usuário, separa o número em seus dígitos individuais e imprime os dígitos separados uns dos outros por três espaços cada. Por exemplo, se o usuário digitar o número `42339`, o programa deve imprimir:

```
4   2   3   3   9
```

Suponha que o usuário insira o número correto de dígitos. O que acontece quando você executa o programa e digita um número com mais de cinco dígitos? O que acontece quando você executa o programa e digita um número com menos de cinco dígitos? [*Dica*: é possível fazer esse exercício com as técnicas que você aprendeu neste capítulo. Você precisará utilizar tanto as operações de divisão como as de resto para "selecionar" cada dígito.]

2.31 **(*Tabela de quadrados e cubos*)** Utilizando apenas as técnicas de programação que aprendeu neste capítulo, escreva um aplicativo que calcule os quadrados e cubos dos números de 0 a 10 e imprime os valores resultantes no formato de tabela como a seguir: [*Nota:* esse programa não requer nenhuma entrada do usuário.]

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

2.32 **(*Valores negativos, positivos e zero*)** Escreva um programa que insere cinco números e determina e imprime quantos números negativos, quantos números positivos e quantos zeros foram inseridos.

Fazendo a diferença

2.33 ***(Calculadora de índice de massa corporal)*** Introduzimos a calculadora de índice de massa corporal (IMC) no Exercício 1.12. As fórmulas para calcular o IMC são

$$IMC = \frac{pesoEmQuilogramas \times 703}{alturaEmMetros \times alturaEmMetros}$$

ou

$$IMC = \frac{pesoEmQuilogramas}{alturaEmMetros \times alturaEmMetros}$$

Crie um aplicativo de calculadora IMC que lê o peso do usuário em libras e a altura em polegadas (ou, se preferir, o peso em quilogramas e a altura em metros) e, então, calcula e exibe o índice de massa corporal do usuário. Além disso, o aplicativo deve exibir as seguintes informações do Department of Health and Human Services/National Institutes of Health, portanto o usuário pode avaliar o seu IMC:

```
BMI VALUES
Underweight: less than 18.5
Normal:      between 18.5 and 24.9
Overweight:  between 25 and 29.9
Obese:       30 or greater
```

[*Nota:* neste capítulo, você aprendeu a utilizar o tipo `int` para representar números inteiros. Os cálculos de IMC quando feitos com valores `int` produzirão ambos os resultados de número inteiro. No Capítulo 3, você aprenderá a utilizar o tipo `double` para representar números com pontos decimais. Quando os cálculos de IMC são realizados com `double`s, eles produzirão ambos os números com pontos decimais — esses são chamados de números de "ponto flutuante".]

2.34 **(*Calculadora de crescimento demográfico mundial*)** Utilize a Web para determinar a população mundial atual e a taxa de crescimento demográfica mundial anual. Escreva um aplicativo que introduza esses valores e, então, que exiba a população mundial estimada depois um, dois, três, quatro e cinco anos.

2.35 **(*Calculadora de economia da faixa solidária*)** Pesquise vários sites Web de faixa solidária (ou "faixa 2") . Crie um aplicativo que calcule o custo diário de dirigir, para que possa estimar quanto dinheiro pode ser economizado com o uso da faixa solidária, que também tem outras vantagens, como reduzir emissões de carbono e congestionamento de tráfego. O aplicativo deve introduzir as seguintes informações e exibir o custo do usuário por dia de dirigir para o trabalho:

a) Milhas totais dirigidas por dia.

b) Preço por galão de gasolina.

c) Milhas médias por galão.

d) Taxas de estacionamento por dia.

e) Pedágio por dia.

3

Você verá uma coisa nova.
Duas coisas. E eu as chamo
Coisa Um e Coisa Dois.
— **Dr. Theodor Seuss Geisel**

Nada pode ter valor sem ser um objeto útil.
— **Karl Marx**

Seus servidores públicos prestam-lhe bons serviços.
— **Adlai E. Stevenson**

Saber como responder àquele que fala,
para responder àquele que envia uma mensagem.
— **Amenemope**

Introdução a classes e objetos

Objetivos

Neste capítulo, você aprenderá:

- O que são classes, objetos, métodos e variáveis de instância.
- Como declarar uma classe e utilizá-la para criar um objeto.
- Como declarar métodos em uma classe para implementar os comportamentos da classe.
- Como declarar variáveis de instância em uma classe para implementar os atributos da classe.
- Como chamar os métodos de um objeto para fazer esses métodos realizarem suas tarefas.
- As diferenças entre variáveis de instância de uma classe e variáveis locais de um método.
- Como utilizar um construtor para assegurar que os dados de um objeto sejam inicializados quando o objeto for criado.
- As diferenças entre tipos por referência primitivos.

Sumário

3.1 Introdução
3.2 Classes, objetos, métodos e variáveis de instância
3.3 Declarando uma classe com um método e instanciando um objeto de uma classe
3.4 Declarando um método com um parâmetro
3.5 Variáveis de instância, métodos *set* e *get*
3.6 Tipos primitivos *versus* tipos por referência
3.7 Inicializando objetos com construtores
3.8 Números de ponto flutuante e tipo `double`
3.9 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando caixas de diálogo
3.10 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

3.1 Introdução

Introduzimos a terminologia e os conceitos básicos da programação orientada a objetos na Seção 1.16. Em geral, os aplicativos que você desenvolve neste livro consistirão em duas ou mais classes, cada uma contendo um ou mais métodos. Se você se tornar parte de uma equipe de desenvolvimento na indústria, você poderá trabalhar em aplicativos com centenas, ou até milhares, de classes. Neste capítulo, apresentamos um framework [estrutura, arcabouço] simples para organizar aplicativos orientados a objetos em Java.

Primeiro, motivamos a noção de classes com um exemplo do mundo real. Em seguida, apresentamos cinco aplicativos funcionais completos para demonstrar a criação e utilização de suas próprias classes. Os quatro primeiros exemplos iniciam nosso estudo de caso sobre desenvolvimento de uma classe livro de notas que instrutores podem utilizar para manter as notas de teste do aluno. Esse estudo de caso é aprimorado nos capítulos 4, 5 e 7. O último exemplo no capítulo introduz números de ponto flutuante — isto é, números que contêm pontos decimais — no contexto de uma classe de conta bancária que mantém o saldo de um cliente.

3.2 Classes, objetos, métodos e variáveis de instância

Iniciemos com uma analogia simples para ajudar a entender classes e seu conteúdo. Suponha que você queira guiar um carro e fazê-lo andar mais rápido pisando no pedal acelerador. O que deve acontecer antes que você possa fazer isso? Bem, antes de poder dirigir um carro, alguém tem de projetá-lo. Em geral, um carro inicia com os desenhos de engenharia, semelhantes às plantas utilizadas para projetar uma casa. Esses desenhos de engenharia incluem o projeto de um pedal acelerador para aumentar a velocidade do carro. O pedal "oculta" do motorista os complexos mecanismos que realmente fazem o carro ir mais rápido, assim como o pedal de freio "oculta" os mecanismos que diminuem a velocidade do carro e a direção "oculta" os mecanismos que mudam a direção do carro. Isso permite que as pessoas com pouco ou nenhum conhecimento de como os motores funcionam dirijam um carro facilmente.

Infelizmente, você não pode determinar o projeto de um carro. Antes de poder guiar um carro, ele deve ser construído a partir dos desenhos de engenharia que o descrevem. Um carro pronto tem um pedal de acelerador real para fazer o carro andar mais rápido, mas até isso não é suficiente — o carro não acelerará por conta própria, então o motorista deve pressionar o pedal do acelerador.

Agora vamos utilizar nosso exemplo do carro para introduzir os conceitos-chave de programação desta seção. Para realizar uma tarefa em um programa é necessário um método. O método descreve os mecanismos que realmente realizam suas tarefas. O método oculta de seu usuário as tarefas complexas que ele realiza, assim como o pedal acelerador de um carro oculta do motorista os complexos mecanismos que fazem o carro andar mais rápido. Em Java, primeiro criamos uma unidade de programa chamada classe para abrigar um método, assim como os desenhos de engenharia do carro abrigam o projeto de um pedal acelerador. Em uma classe, você fornece um ou mais métodos que são projetados para realizar as tarefas da classe. Por exemplo, uma classe que representa uma conta bancária poderia conter um método para fazer depósitos de dinheiro em uma conta, outro para fazer saques e um terceiro para perguntar qual é o saldo atual.

Assim como você não pode dirigir um desenho de engenharia de um carro, você não pode "dirigir" uma classe. Assim como alguém tem de construir um carro a partir de seus desenhos de engenharia antes de poder realmente guiar o carro, você deve construir um objeto de uma classe antes de fazer um programa realizar as tarefas que a classe descreve como fazer. Essa é uma razão de o Java ser conhecido como uma linguagem de programação orientada a objetos.

Ao dirigir um carro, o ato de pressionar o acelerador envia uma mensagem para o carro realizar uma tarefa — isto é, fazer o carro andar mais rápido. De maneira semelhante, você **envia mensagens para um objeto** — cada mensagem é implementada como uma **chamada de método** que instrui um método do objeto a realizar sua tarefa.

Até aqui, utilizamos a analogia do carro para introduzir classes, objetos e métodos. Um carro, além de suas capacidades, também tem muitos atributos, como cor, o número de portas, a quantidade de gasolina no tanque, a velocidade atual e o total de quilômetros percorridos (isto é, a leitura do hodômetro). Como as capacidades do carro, esses atributos são representados como parte do projeto de um carro em seus diagramas de engenharia. Quando você dirige um carro, esses atributos estão sempre associados com o carro. Cada carro mantém seus próprios atributos. Por exemplo, cada carro sabe a quantidade de gasolina que há no seu tanque, mas não sabe quanto há no tanque de outros carros. Um objeto tem atributos que são carregados com o objeto quando ele é utilizado em um programa. Esses atributos são especificados

como parte da classe do objeto. Por exemplo, um objeto conta bancária tem um atributo 'saldo' que representa a quantidade de dinheiro na conta. Cada objeto 'conta bancária' sabe o saldo da conta que ele representa, mas não sabe os saldos de outras contas no banco. Os atributos são especificados pelas **variáveis de instância** da classe.

Resumo dos exemplos deste capítulo

O restante deste capítulo apresenta exemplos que demonstram os conceitos que introduzimos no contexto da analogia do carro. Os quatro primeiros exemplos constroem incrementalmente uma classe `GradeBook` para demonstrar esses conceitos:

1. O primeiro exemplo apresenta uma classe `GradeBook` com um método que simplesmente exibe uma mensagem de boas-vindas quando é chamado. Então mostramos como criar um objeto dessa classe e chamar o método para ele exibir a mensagem de boas-vindas.
2. O segundo exemplo aprimora o primeiro para permitir ao método receber o nome de um curso como argumento e exibir o nome como parte da mensagem de boas-vindas.
3. O terceiro exemplo mostra como armazenar o nome do curso em um objeto `GradeBook`. Para essa versão da classe, nós também mostramos como utilizar métodos para configurar o nome do curso e obter seu nome.
4. O quarto exemplo demonstra como os dados em um objeto `GradeBook` podem ser inicializados quando o objeto é criado.

O último exemplo apresenta uma classe `Account` que reforça os conceitos apresentados nos quatro primeiros exemplos e introduz números de ponto flutuante. A classe `Account` representa uma conta bancária e mantém seu saldo como um número de ponto flutuante. A classe contém dois métodos — um que credita um depósito à conta, aumentando assim o saldo, e outro número que recupera o saldo. O construtor da classe permite que o saldo de cada objeto `Account` seja inicializado quando o objeto é criado. Criamos dois objetos `Account` e fizemos depósitos em cada um para mostrar que cada objeto mantém seu próprio saldo. O exemplo também demonstra como inserir e exibir números de ponto flutuante.

3.3 Declarando uma classe com um método e instanciando um objeto de uma classe

Nas seções 2.5 e 2.8, você criou um objeto da classe existente `Scanner` e, então, utilizou esse objeto para ler dados a partir do teclado. Nesta seção, você criará uma nova classe e irá usá-la para criar um objeto. Iniciamos com um exemplo que consiste em classes `GradeBook` (Figura 3.1) e `GradeBookTest` (Figura 3.2). A classe `GradeBook` (declarada no arquivo `GradeBook.java`) será utilizada para exibir uma mensagem na tela (Figura 3.2) que dá boas-vindas ao instrutor do aplicativo livro de notas. A classe `GradeBookTest` (declarada no arquivo `GradeBookTest.java`) é uma classe de aplicativos na qual o método `main` utilizará um objeto da classe `GradeBook`. Cada declaração de classe que inicia com a palavra-chave `public` deve ser armazenada em um arquivo que tenha o mesmo nome da classe e terminar com a extensão de nome de arquivo `.java`. Dessa forma, as classes `GradeBook` e `GradeBookTest` devem ser declaradas em arquivos separados, porque cada classe é declarada `public`. Como discutiremos, iniciando no Capítulo 8, há outros modos de declarar classes.

Erro comum de programação 3.1

Declarar mais de uma classe `public` no mesmo arquivo é um erro de compilação.

A classe GradeBook

A declaração de classe `GradeBook` (Figura 3.1) contém um método `displayMessage` (linhas 7–10) que exibe uma mensagem na tela. Lembre-se de que uma classe é como uma planta arquitetônica — precisaremos fazer um objeto dessa classe e chamar seu método para executar a linha 9 e exibir a mensagem.

```
1  // Figura 3.1: GradeBook.java
2  // Declaração de classe com um método.
3
4  public class GradeBook
5  {
6     // exibe uma mensagem de boas-vindas para o usuário GradeBook
7     public void displayMessage()
8     {
9        System.out.println( "Welcome to the Grade Book!" );
10    } // fim do método displayMessage
11 } // fim da classe GradeBook
```

Figura 3.1 | Declaração de classe com um método.

A declaração de classe inicia na linha 4. A palavra-chave `public` é um **modificador de acesso**. Por enquanto, simplesmente declaramos toda classe `public`. Cada declaração de classe contém a palavra-chave `class` seguida imediatamente do nome da classe. O corpo de cada classe está entre as chaves esquerda e direita (`{` e `}`), como nas linhas 5–11 da classe `GradeBook`.

No Capítulo 2, cada classe que declaramos tinha um método chamado `main`. A classe `GradeBook` também tem um método — `displayMessage` (linhas 7–10). Lembre-se de que `main` é um método especial que será sempre chamado automaticamente pela Java Virtual Machine quando você executar um aplicativo. A maioria dos métodos não é chamada automaticamente. Como logo verá, você deve chamar o método `displayMessage` para instruí-lo explicitamente a realizar sua tarefa.

A declaração do método começa com palavra-chave `public` para indicar que o método está "disponível para o público" — ele pode ser chamado a partir de métodos de outras classes. Em seguida, vem o **tipo de retorno** do método, que especifica o tipo de dados que o método retorna depois de realizar sua tarefa. O tipo de retorno `void` indica que esse método realizará uma tarefa mas *não* retornará (isto é, devolverá) nenhuma informação para seu **método chamador** ao completar sua tarefa. Você já usou métodos que retornam informações — por exemplo, no Capítulo 2 você usou o método `Scanner` `nextInt` para inserir um inteiro digitado pelo usuário no teclado. Quando `nextInt` lê um valor, ele retorna esse valor para uso no programa.

O nome do método, `displayMessage`, segue o tipo de retorno. Por convenção, os nomes de método iniciam com a primeira letra minúscula e as palavras subsequentes do nome iniciam com uma maiúscula. Os parênteses depois do nome do método indicam que isso é um método. Os parênteses vazios, como na linha 7, indicam que esse método não requer informações adicionais para realizar sua tarefa. A linha 7 é comumente referida como o **cabeçalho de método**. O corpo de cada método é delimitado pelas chaves esquerda e direita (`{` e `}`), como nas linhas 8–10.

O corpo de um método contém uma ou mais instruções que realizam a tarefa do método. Nesse caso, o método contém uma instrução (linha 9) que exibe a mensagem `"Welcome to the GradeBook!"` seguida por caractere de nova linha na janela de comando. Depois que essa instrução executar, o método terá concluído sua tarefa.

Em seguida, gostaríamos de utilizar a classe `GradeBook` em um aplicativo. Como você aprendeu no Capítulo 2, o método `main` começa a execução de cada aplicativo. Uma classe que contém o método `main` inicia a execução de um aplicativo Java. A classe `GradeBook` não é um aplicativo porque não contém `main`. Portanto, se tentar executar `GradeBook` digitando `java GradeBook` na janela de comando, você obterá uma mensagem de erro como essa:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Isso não era um problema no Capítulo 2, porque cada classe que você declarava tinha um método `main`. Para corrigir esse problema, devemos declarar uma classe separada que contenha um método `main` ou colocar um método `main` na classe `GradeBook`. Para ajudá-lo a se preparar para os programas maiores encontrados mais adiante neste livro e na indústria, utilizamos uma classe separada (`GradeBookTest` nesse exemplo) que contém o método `main` para testar cada classe nova criada neste capítulo. Alguns programadores tratam essa classe como uma classe *driver*.

A classe `GradeBookTest`

A declaração de classe `GradeBookTest` (Figura 3.2) contém o método `main` que controlará a execução do nosso aplicativo. A declaração de classe `GradeBookTest` inicia na linha 4 e termina na linha 15. A classe contém somente um método `GradeBookTest`, que é típico de muitas classes que iniciam a execução de um aplicativo.

```
1  // Figura 3.2: GradeBookTest.java
2  // Criando um objeto GradeBook e chamando seu método displayMessage.
3
4  public class GradeBookTest
5  {
6     // método main inicia a execução de programa
7     public static void main( String[] args )
8     {
9        // cria um objeto GradeBook e o atribui a myGradeBook
10       GradeBook myGradeBook = new GradeBook();
11
12       // chama método displayMessage de myGradeBook
13       myGradeBook.displayMessage();
14    } // fim de main
15 } // fim da classe GradeBookTest
```

```
Welcome to the Grade Book!
```

Figura 3.2 | Criando um objeto `GradeBook` e chamando seu método `displayMessage`.

As linhas 7–14 declaram o método `main`. Uma parte-chave para permitir à JVM localizar e chamar o método `main` para iniciar a execução do aplicativo é a palavra-chave `static` (linha 7), que indica que `main` é um método `static`. Um método `static` é especial, porque pode ser chamado sem primeiro criar um objeto da classe em que o método é declarado. Explicamos completamente os métodos `static` no Capítulo 6, "Métodos: uma visão mais aprofundada".

Nesse aplicativo, gostaríamos de chamar o método `displayMessage` da classe `GradeBook` para exibir a mensagem de boas-vindas na janela de comando. Em geral, você não pode chamar um método que pertence a outra classe até criar um objeto dessa classe, como mostrado na linha 10. Iniciamos com a declaração da variável `myGradeBook`. Observe que o tipo da variável é `GradeBook` — a classe que declaramos na Figura 3.1. Cada nova classe que você cria torna-se um novo tipo que pode ser utilizado para declarar variáveis e criar objetos. Você pode declarar novos tipos de classe conforme necessário; essa é uma razão pela qual o Java é conhecido como uma **linguagem extensível**.

A variável `myGradeBook` é inicializada com o resultado da **expressão de criação de instância de classe** `new GradeBook()`. A palavra-chave **new** cria um novo objeto da classe especificada à direita da palavra-chave (isto é, `GradeBook`). Os parênteses à direita de `GradeBook` são necessários. Como você aprenderá na Seção 3.7, esses parênteses em combinação com um nome de classe representam uma chamada para um **construtor**, que é semelhante a um método, mas é utilizado na hora em que um objeto é criado para inicializar os dados do objeto. Nessa seção você verá que os dados podem ser colocados entre parênteses para especificar os valores iniciais para os dados do objeto. Por enquanto, simplesmente não incluímos nada entre os parênteses.

Assim como podemos usar o objeto `System.out` para chamar os métodos `print`, `printf` e `println`, também podemos usar o objeto `myGradeBook` para chamar o método `displayMessage`. A linha 13 chama o método `displayMessage` (as linhas 7–10 da Figura 3.1) usando `myGradeBook` seguido por um **ponto separador** (`.`), o nome do método `displayMessage` e um conjunto de parênteses vazio. Essa chamada faz com que o método `displayMessage` realize sua tarefa. Essa chamada de método difere daquelas do Capítulo 2 que exibiam informações em uma janela de comando — todas aquelas chamadas de método forneciam argumentos que especificavam os dados a serem exibidos. No começo da linha 13, `myGradeBook.` indica que `main` deve utilizar o objeto `myGradeBook` que foi criado na linha 10. A linha 7 da Figura 3.1 indica que o método `displayMessage` tem uma lista de parâmetros vazia — isto é, `displayMessage` não requer informações adicionais para realizar sua tarefa. Por isso, a chamada de método (linha 13 da Figura 3.2) especifica um conjunto de parênteses vazio depois do nome do método para indicar que nenhum argumento está sendo passado ao método `displayMessage`. Quando o método `displayMessage` completa sua tarefa, o método `main` continua a executar na linha 14. Esse é o fim do método `main`, portanto, o programa termina.

Observe que qualquer classe pode conter um método `main`. O JVM invoca o método `main` somente na classe utilizada para executar o aplicativo. Se um aplicativo tiver múltiplas classes que contêm `main`, então aquele que é invocado é aquele na classe nomeada no comando `java`.

Compilando um aplicativo com múltiplas classes

Você deve compilar as classes na Figura 3.1 e Figura 3.2 antes de poder executar o aplicativo. Primeiro, mude para o diretório que contém os arquivos de código-fonte do aplicativo. Em seguida, digite o comando:

```
javac GradeBook.java GradeBookTest.java
```

para compilar ambas as classes de uma só vez. Se o diretório que contém o aplicativo incluir somente os arquivos desse aplicativo, você pode compilar todas as classes no diretório com o comando:

```
javac *.java
```

O asterisco (`*`) em `*.java` indica que todos os arquivos no diretório atual que têm a extensão `.java` devem ser compilados.

Diagrama de classes UML para a classe `GradeBook`

A Figura 3.3 apresenta um **diagrama de classes UML** para a classe `GradeBook` da Figura 3.1. Lembre-se da Seção 1.16 que a UML é uma linguagem gráfica utilizada pelos programadores para representar sistemas orientados a objetos de uma maneira padronizada. Na UML, cada classe é modelada em um diagrama de classe como um retângulo com três **compartimentos**. O superior contém o nome da classe centralizado horizontalmente no tipo negrito. O compartimento do meio contém os atributos da classe, que correspondem às variáveis de instância (discutidas na Seção 3.5) em Java. Na Figura 3.3, o compartimento do meio está vazio, porque essa classe `GradeBook` não tem nenhum atributo. O compartimento inferior contém as **operações** da classe, que correspondem aos métodos em Java. A UML modela operações listando o nome da operação precedido por um modificador de acesso (nesse caso +) e seguido por um conjunto de parênteses. A classe `GradeBook` tem um método, `displayMessage`, então o compartimento inferior da Figura 3.3 lista uma operação com esse nome. O método `displayMessage` não requer informações adicionais para executar suas tarefas, portanto os parênteses depois do nome do método no diagrama de classe estão vazios, da mesma maneira como estavam na declaração do método na linha 7 da Figura 3.1. O sinal de adição (+) na frente do nome da operação indica que `displayMessage` é uma operação `public` na UML (isto é, um método `public` no Java). Utilizaremos com frequência os diagramas de classe UML para resumir os atributos e as operações de uma classe.

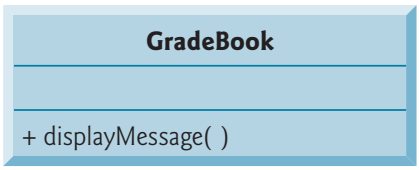


Figura 3.3 | Diagrama de classe UML indicando que a classe `GradeBook` tem uma operação `public displayMessage`.

3.4 Declarando um método com um parâmetro

Em nossa analogia do carro da Seção 3.2, discutimos o fato de que pressionar o acelerador de um carro envia uma mensagem para ele realizar uma tarefa — fazer o carro andar mais rápido. Mas quanto o carro deve acelerar? Como você sabe, quanto mais pressionar o pedal, mais rápido o carro irá acelerar. Então a mensagem para o carro na verdade inclui a tarefa a ser realizada e informações adicionais que ajudam o carro a executar essa tarefa. Essas informações adicionais são conhecidas como **parâmetro** — o valor do parâmetro ajuda o carro a determinar com que rapidez acelerar. De maneira semelhante, um método pode exigir um ou mais parâmetros que representam informações adicionais necessárias para realizar a tarefa. Os parâmetros são definidos em uma **lista de parâmetros** separados por vírgula, que está localizada nos parênteses depois do nome do método. Todo parâmetro deve especificar um tipo e um identificador. A lista de parâmetros pode conter qualquer número de parâmetros, inclusive nenhum parâmetro. Os parênteses vazios que se seguem ao nome de método (como na Figura 3.1, linha 7) indicam que um método não requer nenhum parâmetro.

Uma chamada de método fornece valores — chamados argumentos — para cada um dos parâmetros do método. Por exemplo, o método `System.out.println` exige um argumento que especifica que dados enviar para a saída em uma janela de comando. De maneira semelhante, para fazer um depósito em uma conta bancária, um método `deposit` especifica um parâmetro que representa a quantidade de depósito. Quando o método `deposit` é chamado, um valor de argumento que representa a quantidade de depósito é atribuído ao parâmetro do método. O método então faz um depósito dessa quantia.

Nosso próximo exemplo declara a classe `GradeBook` (Figura 3.4) com um método `displayMessage` que exibe o nome do curso como parte da mensagem de boas-vindas. (Ver a execução de exemplo na Figura 3.5.) O novo método `displayMessage` requer um parâmetro que representa o nome do curso a ser enviado para a saída.

```
1  // Figura 3.4: GradeBook.java
2  // Declaração de classe com um método que tem um parâmetro.
3
4  public class GradeBook
5  {
6     // exibe uma mensagem de boas-vindas para o usuário GradeBook
7     public void displayMessage( String courseName )
8     {
9        System.out.printf( "Welcome to the GradeBook for\n%s!\n",
10          courseName );
11    } // fim do método displayMessage
12 } // fim da classe GradeBook
```

Figura 3.4 | Declaração de classe com um método que tem um parâmetro.

```
1  // Figura 3.5: GradeBookTest.java
2  // Cria objeto GradeBook e passa uma String para
3  // seu método displayMessage.
4  import java.util.Scanner; // programa utiliza Scanner
5
6  public class GradeBookTest
7  {
8     // método main inicia a execução de programa
9     public static void main( String[] args )
10    {
11       // cria Scanner para obter entrada a partir da janela de comando
12       Scanner input = new Scanner( System.in );
13
```

```
14          // cria um objeto GradeBook e o atribui a myGradeBook
15          GradeBook myGradeBook = new GradeBook();
16
17          // prompt para entrada do nome do curso
18          System.out.println( "Please enter the course name:" );
19          String nameOfCourse = input.nextLine(); // lê uma linha de texto
20          System.out.println(); // gera saída de uma linha em branco
21
22          // chama método displayMessage de myGradeBook
23          // e passa nameOfCourse como um argumento
24          myGradeBook.displayMessage( nameOfCourse );
25       } // fim de main
26    } // fim da classe GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the GradeBook for
CS101 Introduction to Java Programming!
```

Figura 3.5 | Criando um objeto `GradeBook` e passando uma `String` ao seu método `displayMessage`.

Antes de discutir os novos recursos da classe `GradeBook`, vejamos como a nova classe é utilizada a partir do método `main` da classe `GradeBookTest` (Figura 3.5). A linha 12 cria um `Scanner` chamado `input` para ler o nome do curso fornecido pelo usuário. A linha 15 cria um objeto da classe `GradeBook` e a atribui à variável `myGradeBook`. A linha 18 exibe um prompt que pede para o usuário inserir o nome de um curso. A linha 19 mostra o nome do usuário e o atribui à variável `nameOfCourse`, utilizando método `Scanner` **`nextLine`** para realizar a entrada. O usuário digita o nome do curso e pressiona *Enter* para enviar o nome do curso ao programa. Observe que pressionar *Enter* insere um caractere de nova linha no final dos caracteres digitados pelo usuário. O método `nextLine` lê os caracteres digitados pelo usuário até o caractere de nova linha ser encontrado, depois retorna uma `String` que contém o caractere até, mas não incluindo, o caractere de nova linha. O caractere de nova linha é descartado. A classe `Scanner` também fornece um método semelhante — **`next`** — que lê palavras específicas. Quando o usuário pressiona *Enter* depois de digitar a entrada, o método `next` lê os caracteres até que um caractere de espaço em branco (como um espaço, tabulação ou nova linha) seja encontrado e retorna uma `String` contendo os caracteres até, mas sem incluir, o caractere de espaço em branco (que é descartado). Nenhuma informação depois do primeiro caractere de espaço em branco é perdida — essas informações podem ser lidas por outras instruções que chamam os métodos de `Scanner` posteriormente no programa. A linha 20 gera a saída de uma linha em branco.

A linha 24 chama o método `displayMessage` de `myGradeBooks`. A variável `nameOfCourse` entre parênteses é o argumento que é passado ao método `displayMessage` para o método poder realizar sua tarefa. O valor da variável `nameOfCourse` em `main` torna-se o valor do *parâmetro* `courseName` do método `displayMessage` na linha 7 da Figura 3.4. Quando você executa esse aplicativo, observe que o método `displayMessage` gera a saída do nome que você digita como parte da mensagem de boas-vindas (Figura 3.5).

Mais sobre argumentos e parâmetros

Na Figura 3.4, a lista de parâmetros de `displayMessage` (linha 7) declara um parâmetro indicando que o método requer uma `String` para executar sua tarefa. Quando o método é chamado, o valor de argumento na chamada é atribuído ao parâmetro correspondente (nesse caso, `courseName`) no cabeçalho do método. Em seguida, o corpo do método utiliza o valor do parâmetro `courseName`. As linhas 9–10 da Figura 3.4 exibem o valor do parâmetro `courseName`, utilizando o especificador de formato `%s` na string de formato do `printf`. Observe que o nome da variável de parâmetro (`courseName` na Figura 3.4, linha 7) pode ser o mesmo nome da variável de argumento ou um nome diferente (`courseName` na Figura 3.5, linha 24).

O número de argumentos em uma chamada de método deve corresponder ao número de parâmetros na lista de parâmetros da declaração do método. Além disso, os tipos de argumento na chamada do método devem ser "consistentes com" os tipos dos parâmetros correspondentes na declaração do método. (Como você aprenderá nos capítulos subsequentes, nem sempre é requerido que um tipo do argumento e o tipo de seu parâmetro correspondente sejam idênticos.) Em nosso exemplo, a chamada de método passa um argumento do tipo `String` (`nameOfCourse` é declarada como uma `String` na linha 19 da Figura 3.5) e a declaração de método especifica um parâmetro de tipo `String` (`courseName` é declarado como uma `String` na linha 7 da Figura 3.4). Portanto, nesse exemplo, o tipo do argumento na chamada de método corresponde exatamente ao tipo do parâmetro no cabeçalho do método.

Erro comum de programação 3.2

Ocorrerá um erro de compilação se o número de argumentos em uma chamada de método não corresponder ao número de parâmetros na declaração do método.

Erro comum de programação 3.3

Ocorrerá um erro de compilação se o tipo de qualquer argumento em uma chamada de método não for consistente com o tipo do parâmetro correspondente na declaração do método.

Diagrama da classe UML atualizado para a classe GradeBook

O diagrama de classes UML da Figura 3.6 modela a classe `GradeBook` da Figura 3.4. Como a Figura 3.1, essa classe `GradeBook` contém `displayMessage`, uma operação `public`. Mas essa versão de `displayMessage` tem um parâmetro. A UML modela um parâmetro de um modo pouco diferente do Java listando o nome de parâmetro, seguido por dois-pontos e pelo tipo de parâmetro nos parênteses que seguem o nome da operação. A UML tem seus próprios tipos de dados semelhantes àqueles do Java (mas, como você verá, todos os tipos de dados UML têm os mesmos nomes dos tipos de Java correspondentes). O tipo UML `String` corresponde ao tipo do Java `String`. `displayMessage` do método `GradeBook` (Figura 3.4) tem um parâmetro `String` chamado `courseName`, portanto a Figura 3.6 lista `courseName: String` entre os parênteses depois de `displayMessage`.

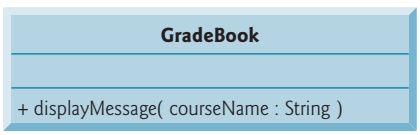


Figura 3.6 | Diagrama de classe UML indicando que a classe `GradeBook` tem uma operação `displayMessage` com um parâmetro `courseName` de tipo UML `String`.

Notas sobre declarações `import`

Note a declaração `import` na Figura 3.5 (linha 4). Essa declaração indica ao compilador que o programa utiliza a classe `Scanner`. Por que precisamos importar a classe `Scanner`, mas não as classes `System`, `String` ou `GradeBook`? As classes `System` e `String` estão no pacote `java.lang`, que é implicitamente importado para cada programa Java, assim todos os programas podem utilizar as classes desse pacote sem importá-las explicitamente. A maioria das outras classes que você utilizará nos programas Java precisa ser importada explicitamente.

Há um relacionamento especial entre as classes que são compiladas no mesmo diretório no disco, como as classes `GradeBook` e `GradeBookTest`. Por padrão, essas classes são consideradas pertencentes ao mesmo pacote — conhecido como o **pacote padrão**. As classes do mesmo pacote são importadas implicitamente para os arquivos de código-fonte de outras classes do mesmo pacote. Assim, uma declaração `import` não é necessária quando uma classe em um pacote utiliza outra no mesmo pacote — por exemplo, quando a classe `GradeBookTest` utiliza a classe `GradeBook`.

A declaração `import` na linha 4 não é requerida se sempre referenciarmos a classe `Scanner` como `java.util.Scanner`, o que inclui o nome inteiro do pacote e o nome de classe. Isto é conhecido como **nome de classe completamente qualificado** da classe. Por exemplo, a linha 12 poderia ser escrita como:

```
java.util.Scanner input = new java.util.Scanner( System.in );
```

Observação de engenharia de software 3.1

O compilador Java não exigirá as declarações `import` em um arquivo de código-fonte Java se o nome de classe completamente qualificado for especificado toda vez que um nome de classe for utilizado no código-fonte. A maioria dos programadores Java prefere utilizar declarações `import`.

3.5 Variáveis de instância, métodos *set* e *get*

No Capítulo 2, declaramos todas as variáveis de um aplicativo no método `main` do aplicativo. As variáveis declaradas no corpo de um método particular são conhecidas como **variáveis locais** e só podem ser utilizadas nesse método. Quando esse método terminar, os valores de suas variáveis locais são perdidos. A partir da Seção 3.2, lembre-se de que um objeto tem atributos que são carregados com o objeto quando ele é utilizado em um programa. Esses atributos existem antes de um método ser chamado em um objeto e depois de o método completar a execução.

Uma classe normalmente consiste em um ou mais métodos que manipulam os atributos que pertencem a um objeto particular da classe. Os atributos são representados como variáveis em uma declaração de classe. Essas variáveis são chamadas **campos** e estão declaradas dentro de uma declaração de classe, mas fora do corpo das declarações de método da classe. Quando cada objeto de uma classe mantém sua própria cópia de um atributo, o campo que representa o atributo também é conhecido como uma variável de instância — cada objeto (instância) da classe tem uma instância separada da variável na memória. O exemplo nesta seção demonstra uma classe `GradeBook` que contém uma variável de instância `courseName` para representar o nome do curso de um objeto `GradeBook` particular.

Classe GradeBook com uma variável de instância, um método set *e um método* get

Em nosso próximo aplicativo (figuras 3.7 e 3.8), a classe `GradeBook` (Figura 3.7) mantém o nome do curso como uma variável de instância para que ele possa ser utilizado ou modificado a qualquer hora durante a execução de um aplicativo. A classe contém três métodos — `setCourseName`, `getCourseName` e `displayMessage`. O método `setCourseName` armazena o nome de um curso de um `GradeBook`. O método `getCourseName` obtém o nome do curso de uma `GradeBook`. O método `displayMessage`, que agora não especifica nenhum parâmetro, ainda exibe uma mensagem de boas-vindas que inclui o nome do curso; como você verá, o método agora obtém o nome do curso chamando outro método na mesma classe — `getCourseName`.

```
1  // Figura 3.7: GradeBook.java
2  // classe GradeBook que contém uma variável de instância
3  // courseName e métodos para configurar e obter seu valor.
4
5  public class GradeBook
6  {
7     private String courseName; // nome do curso para esse GradeBook
8
9     // método para configurar o nome do curso
10    public void setCourseName( String name )
11    {
12       courseName = name; // armazena o nome do curso
13    } // fim do método setCourseName
14
15    // método para recuperar o nome do curso
16    public String getCourseName()
17    {
18       return courseName;
19    } // fim do método getCourseName
20
21    // exibe uma mensagem de boas-vindas para o usuário GradeBook
22    public void displayMessage()
23    {
24       // chama getCourseName para obter o nome do
25       // o curso que essa GradeBook representa
26       System.out.printf( "Welcome to the GradeBook for\n%s!\n",
27          getCourseName() );
28    } // fim do método displayMessage
29 } // fim da classe GradeBook
```

Figura 3.7 | A classe `GradeBook` que contém uma variável de instância `courseName` e métodos para configurar e obter seu valor.

Um instrutor típico dá mais de um curso, cada um com seu próprio nome. A linha 7 declara que `courseName` é uma variável de tipo `String`. Como a variável é declarada no corpo da classe, mas fora dos corpos dos métodos da classe (linhas 10–13, 16–19 e 22–28), a linha 7 é uma declaração para uma variável de instância. Toda instância (isto é, objeto) de classe `GradeBook` contém uma cópia de cada variável de instância. Por exemplo, se houver dois objetos `GradeBook`, cada objeto terá uma cópia própria de `courseName` (uma por objeto). Um benefício de tornar `courseName` uma variável de instância é que todos os métodos da classe (nesse caso, `GradeBook`) podem manipular qualquer variável de instância que aparece na classe (nesse caso, `courseName`).

Modificadores de acesso `public` *e* `private`

A maioria das declarações de variável de instância é precedida pela palavra-chave `private` (como na linha 7). Como **`public`**, a palavra-chave **`private`** é um modificador de acesso. As variáveis ou métodos declarados com o modificador de acesso `private` só são acessíveis a métodos da classe em que são declarados. Dessa forma, a variável `courseName` só pode ser utilizada nos métodos `setCourseName`, `getCourseName` e `displayMessage` da (cada objeto da) classe `GradeBook`.

A declaração de variáveis de instância com o modificador de acesso `private` é conhecida como **ocultamento de dados** ou ocultamento de informações. Quando um programa cria (instancia) um objeto de classe `GradeBook`, a variável `courseName` é encapsulada (ocultada) no objeto e pode ser acessada apenas por métodos da classe do objeto. Isso evita que `courseName` seja modificado acidentalmente por uma classe em outra parte do programa. Na classe `GradeBook`, os métodos `setCourseName` e `getCourseName` manipulam a variável de instância `courseName`.

Observação de engenharia de software 3.2

Anteceda cada campo e declaração de método com um modificador de acesso. Geralmente, as variáveis de instância devem ser declaradas `private` *e os métodos* `public`. *(Veremos que é apropriado declarar certos métodos* `private`, *se eles forem acessados apenas por outros métodos da classe.)*

Boa prática de programação 3.1

Preferimos listar os campos de uma classe primeiro, de modo que, à medida que você leia o código, também veja os nomes e tipos das variáveis antes que sejam utilizados nos métodos da classe. Você pode listar os campos da classe em qualquer lugar na classe fora de suas declarações de método, mas sua dispersão tende a resultar em um código de difícil leitura.

Boa prática de programação 3.2

Coloque uma linha em branco entre as declarações de método para separar os métodos e aprimorar a legibilidade do programa.

Métodos `setCourseName` *e* `getCourseName`

O método `setCourseName` (linhas 10–13) não retorna quaisquer dados quando ele completa sua tarefa, portanto, seu tipo de retorno é `void`. O método recebe um parâmetro — `name` — que representa o nome do curso que será passado para o método como um argumento. A linha 12 atribui `name` à variável de instância `courseName`.

O método `getCourseName` (linhas 16–19) devolve um objeto `courseName` específico de `GradeBook`. O método tem uma lista vazia de parâmetros, então não exige informações adicionais para realizar sua tarefa. O método especifica que ele retorna uma `String` — esse é o tipo de retorno do método. Quando um método que especifica um tipo de retorno diferente de `void` for chamado e completar sua tarefa, o método retornará um resultado para seu método chamador. Por exemplo, ao utilizar um caixa eletrônico (Automated Teller Machine — ATM) e solicitar o saldo da sua conta, você espera o ATM devolver um valor que representa seu saldo. De maneira semelhante, quando uma instrução chama o método `getCourseName` em um objeto `GradeBook`, a instrução espera receber o nome do curso do `GradeBook` (nesse caso, um `String`, como especificado no tipo de retorno da declaração de método).

A instrução **`return`** na linha 18 passa o valor da variável de instância `courseName` de volta à instrução que chama o método `getCourseName`. Considere, a linha 27 do método `displayMessage`, que chama o método `getCourseName`. Quando o valor é retornado, a instrução nas linhas 26–27 utiliza esse valor para gerar o nome do curso. De modo semelhante, se você tiver um método `square` que retorna o quadrado de seu argumento, você esperaria que a instrução:

```
int result = square( 2 );
```

retornasse `4` a partir do método `square` e atribuísse `4` à variável `result`. Se tiver um método `maximum` que retorna o maior de três argumentos do tipo inteiro, você esperaria que a instrução:

```
int biggest = maximum( 27, 114, 51 );
```

retornasse `114` a partir do método `maximum` e atribuísse 114 à variável `biggest`.

Observe que todas as instruções nas linhas 12 e 18 utilizam `courseName`, embora ele não tenha sido declarado em nenhum dos métodos. Podemos utilizar `courseName` em métodos `GradeBook` porque `courseName` é uma variável de instância da classe. Observe também que a ordem em que os métodos são declarados em uma classe não determina quando eles são chamados em tempo de execução. Então o método `getCourseName` poderia ser declarado antes de método `setCourseName`.

O método `displayMessage`

O método `displayMessage` (linhas 22–28) não retorna quaisquer dados quando ele completa sua tarefa, portanto, seu tipo de retorno é `void`. O método não recebe parâmetros, então, a lista de parâmetros está vazia. As linhas 26–27 geram saída de uma mensagem de boas-vindas que inclui o valor da variável de instância `courseName`, que é retornado pela chamada ao método `getCourseName` na linha 27. Note que um método de uma classe (`displayMessage` nesse caso) pode chamar outro método da mesma classe utilizando apenas o nome do método (nesse caso, `getCourseName`). Mais uma vez, precisamos criar um objeto de classe `GradeBook` e chamar seus métodos antes que a mensagem de boas-vindas possa ser exibida.

A classe `GradeBookTest` *que demonstra a classe* `GradeBook`

A classe `GradeBookTest` (Figura 3.8) cria um objeto da classe `GradeBook` e demonstra seus métodos. A linha 14 cria um objeto `GradeBook` e o atribui à variável local `myGradeBook` de tipo `GradeBook`. As linhas 17–18 exibem o nome do curso inicial que chama o método `getCourseName` do objeto. Observe que a primeira linha da saída mostra o nome "`null`". Diferentemente das variáveis locais, que não são automaticamente inicializadas, todo campo tem um **valor inicial padrão** — um valor fornecido pelo Java quando você não especifica o valor inicial do campo. Portanto, não é exigido que os campos sejam explicitamente inicializados antes de serem utilizados em um programa — a menos que eles devam ser inicializados para valores diferentes de seus valores padrão. O valor padrão de um campo do tipo `String` (como `courseName` nesse exemplo) é `null`, sobre o qual discutiremos mais na Seção 3.6.

```
1  // Figura 3.8: GradeBookTest.java
2  // Criando e manipulando um objeto GradeBook.
3  import java.util.Scanner; // programa utiliza Scanner
4
5  public class GradeBookTest
6  {
7     // método main inicia a execução de programa
8     public static void main( String[] args )
9     {
10       // cria Scanner para obter entrada a partir da janela de comando
11       Scanner input = new Scanner( System.in );
12
13       // cria um objeto GradeBook e o atribui a myGradeBook
14       GradeBook myGradeBook = new GradeBook();
15
16       // exibe valor inicial de courseName
17       System.out.printf( "Initial course name is: %s\n\n",
18          myGradeBook.getCourseName() );
19
20       // solicita e lê o nome do curso
21       System.out.println( "Please enter the course name:" );
22       String theName = input.nextLine(); // lê uma linha de texto
23       myGradeBook.setCourseName( theName ); // configura o nome do curso
24       System.out.println(); // gera saída de uma linha em branco
25       // exibe mensagem de boas-vindas depois de especificar o
26       // nome do curso
27       myGradeBook.displayMessage();
28    } // fim de main
29 } // fim da classe GradeBookTest
```

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the GradeBook for
CS101 Introduction to Java Programming!
```

Figura 3.8 | Criando e manipulando um objeto `GradeBook`.

A linha 21 exibe um prompt que pede para usuário inserir o nome de um curso. A variável `String` `theName` local (declarada na linha 22) é inicializada com o nome do curso inserido pelo usuário, que é retornado pela chamada ao método `nextLine` do objeto `Scanner` `input`. A linha 23 chama o método `setCourseName` do objeto `myGradeBook` e armazena `theName` como o argumento do método. Quando o método é chamado, o valor do argumento é atribuído ao parâmetro `name` (linha 10, Figura 3.7) do método `setCourseName` (linhas 10–13, Figura 3.7). Então o valor do parâmetro é atribuído à variável de instância `courseName` (linha 12, Figura 3.7). A linha 24 (Figura 3.8) pula uma linha na saída e então a linha 27 chama o método `displayMessage` do objeto `myGradeBook` para exibir a mensagem de boas-vindas contendo o nome do curso.

Métodos set *e* get

Os campos `private` de uma classe só podem ser manipulados pelos métodos de classe. Portanto, um **cliente de um objeto** — isto é, qualquer classe que chama os métodos do objeto — chama métodos `public` da classe para manipular os campos `private` de um objeto da classe. É por isso que as instruções no método `main` (Figura 3.8) chamam os métodos `setCourseName`, `getCourseName` e `displayMessage` em um objeto `GradeBook`. As classes costumam fornecer métodos `public` para permitir a clientes ***configurar*** (***set***, isto é, atribuir valores a) ou ***obter*** (***get***, isto é, obter os valores de) variáveis de instância `private`. Os nomes desses métodos não precisam começar com *set* ou *get*, mas essa convenção de atribuição de nomes é recomendada e requerida para componentes de software Java especiais, denominados JavaBeans, que podem simplificar a programação em muitos ambientes de desenvolvimento integrado Java (Integrated Development Environments — IDEs). O método que configura (*set*) a variável de instância `courseName` nesse exemplo chama-se `setCourseName` e o método que obtém (*get*) o valor de `courseName` chama-se `getCourseName`.

Diagrama de classe UML do GradeBook *com uma variável de instância e os métodos* set *e* get

A Figura 3.9 contém um diagrama de classes UML atualizado para a versão da classe `GradeBook` na Figura 3.7. Esse diagrama modela a variável de instância `courseName` da classe `GradeBook` como um atributo no compartimento no meio da classe. A UML representa as

variáveis de instância como atributos listando o nome do atributo, seguido por um caractere de dois-pontos e o tipo de atributo. O tipo UML do atributo `courseName` é `String`. A variável de instância `courseName` é `private` em Java, portanto, o diagrama de classe lista um modificador de acesso com o sinal de subtração (–) na frente do nome do atributo correspondente. A classe `GradeBook` contém três métodos `public`, então o diagrama de classe lista três operações no terceiro compartimento. Lembre-se de que o sinal de adição (+) antes de cada nome de operação indica que a operação é `public`. A operação `setCourseName` tem um parâmetro `String` chamado `name`. A UML indica o tipo de retorno de uma operação colocando um dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação. O método `getCourseName` da classe `GradeBook` (Figura 3.7) tem um tipo de retorno `String` em Java, então o diagrama de classe mostra um tipo de retorno `String` na UML. Observe que as operações `setCourseName` e `displayMessage` não retornam valores (isto é, elas retornam `void` em Java), portanto, o diagrama de classe UML não especifica um tipo de retorno depois dos parênteses dessas operações.

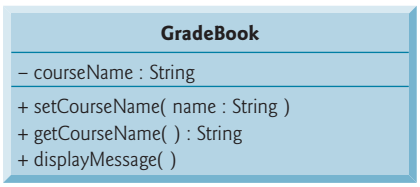


Figura 3.9 | O diagrama de classes UML que indica que a classe `GradeBook` tem um atributo `courseName` privado do tipo UML `String` e três operações públicas — `setCourseName` (com um parâmetro `name` do tipo UML `String`), `getCourseName` (que retorna o tipo UML `String`) e `displayMessage`.

3.6 Tipos primitivos *versus* tipos por referência

Os tipos do Java são divididos em tipos primitivos e **tipos por referência**. Os tipos primitivos são `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`. Todos os tipos não primitivos são tipos por referência, portanto, as classes, que especificam os tipos de objeto, são tipos por referência.

Uma variável de tipo primitivo pode armazenar exatamente um valor de seu tipo declarado por vez. Por exemplo, uma variável `int` pode armazenar um número inteiro (como 7) por vez. Quando outro valor for atribuído a essa variável, seu valor inicial será substituído. As variáveis de instância de tipo primitivo são inicializadas por padrão — as variáveis dos tipos `byte`, `char`, `short`, `int`, `long`, `float` e `double` são inicializadas como 0, e as variáveis do tipo `boolean` são inicializadas como `false`. Você pode especificar seu próprio valor inicial para uma variável do tipo primitivo atribuindo à variável um valor na sua declaração. Lembre-se de que as variáveis locais *não* são inicializadas por padrão.

Dica de prevenção de erro 3.1
Uma tentativa de utilizar uma variável local não inicializada causa um erro de compilação.

Os programas utilizam as variáveis de tipos por referência (normalmente chamados **referências**) para armazenar as localizações de objetos na memória do computador. Diz-se que tal variável **referencia um objeto** no programa. Os objetos que são referenciados podem todos conter muitas variáveis de instância e métodos. A linha 14 da Figura 3.8 cria um objeto de classe `GradeBook` e a variável `myGradeBook` contém uma referência a esse objeto `GradeBook`. As variáveis de instância de tipo por referência são inicializadas por padrão com o valor **`null`** — uma palavra reservada que representa uma "referência a nada". Essa é a razão por que a primeira chamada a `getCourseName` na linha 18 da Figura 3.8 retornou `null` — o valor de `courseName` não foi configurado, assim o valor inicial padrão `null` foi retornado. A lista completa das palavras reservadas e palavras-chave está listada no Apêndice C.

Ao utilizar um objeto de outra classe, uma referência ao objeto deve **invocar** (isto é, chamar) seus métodos. No aplicativo da Figura 3.8, as instruções no método `main` utilizam a variável `myGradeBook` para enviar as mensagens para o objeto `GradeBook`. Essas mensagens são chamadas para os métodos (como `setCourseName` e `getCourseName`) que permitem ao programa interagir com o objeto `GradeBook`. Por exemplo, a instrução na linha 23 utiliza `myGradeBook` para enviar a mensagem `setCourseName` ao objeto `GradeBook`. A mensagem inclui o argumento que `setCourseName` exige para realizar sua tarefa. O objeto `GradeBook` utiliza essas informações para configurar a variável de instância `courseName`. Observe que as variáveis de tipo primitivo não referenciam objetos, então essas variáveis não podem ser utilizadas para invocar métodos.

Observação de engenharia de software 3.3
O tipo declarado de uma variável (por exemplo, `int`, `double` ou `GradeBook`) indica se a variável é de um tipo primitivo ou tipo por referência. Se o tipo de uma variável não for um dos oito tipos primitivos, então ele é um tipo por referência.

3.7 Inicializando objetos com construtores

Como mencionado na Seção 3.5, quando um objeto de classe `GradeBook` (Figura 3.7) é criado, sua variável de instância `courseName` é inicializada como `null` por padrão. E se você quiser fornecer o nome de um curso quando criar um objeto `GradeBook`? Cada classe que você declara pode fornecer um método especial chamado construtor que pode ser utilizado para inicializar um objeto de uma classe quando o objeto for criado. De fato, o Java requer uma chamada de construtor para todo objeto que é criado. A palavra-chave `new` solicita memória do sistema para armazenar um objeto e então chama o construtor da classe correspondente para inicializar o objeto. A chamada é indicada pelos parênteses depois do nome da classe. Um construtor deve ter o mesmo nome que a classe. Por exemplo, a linha 14 da Figura 3.8 utiliza primeiro `new` para criar um objeto `GradeBook`. Os parênteses vazios depois de "`new GradeBook`" indicam uma chamada ao construtor da classe sem argumentos. Por padrão, o compilador fornece um **construtor padrão** sem parâmetros em qualquer classe que não inclui explicitamente um construtor. Quando uma classe tem somente o construtor padrão, suas variáveis de instância são inicializadas de acordo com seus valores padrões.

Ao declarar uma classe, você pode fornecer seu próprio construtor a fim de especificar a inicialização personalizada para objetos de sua classe. Por exemplo, talvez você queira especificar o nome de um curso para um objeto `GradeBook` quando o objeto é criado, como em:

```
GradeBook myGradeBook =
   new GradeBook( "CS101 Introduction to Java Programming" );
```

Nesse caso, o argumento `"CS101 Introduction to Java Programming"` é passado para o construtor do objeto `GradeBook` e utilizado para inicializar `courseName`. A instrução anterior exige que a classe forneça um construtor com um parâmetro `String`. A Figura 3.10 contém uma classe `GradeBook` modificada com esse construtor.

```
1  // Figura 3.10: GradeBook.java
2  // Classe GradeBook com um construtor para inicializar o
3  // nome de um curso.
4  public class GradeBook
5  {
6     private String courseName; // nome do curso para esse GradeBook
7
8     // o construtor inicializa courseName com o argumento String
9     public GradeBook( String name )
10    {
11       courseName = name; // inicializa courseName
12    } // fim do construtor
13
14    // método para configurar o nome do curso
15    public void setCourseName( String name )
16    {
17       courseName = name; // armazena o nome do curso
18    } // fim do método setCourseName
19
20    // método para recuperar o nome do curso
21    public String getCourseName()
22    {
23       return courseName;
24    } // fim do método getCourseName
25
26    // exibe uma mensagem de boas-vindas para o usuário GradeBook
27    public void displayMessage()
28    {
29       // essa instrução chama getCourseName para obter o
30       // nome do curso que esse GradeBook representa
31       System.out.printf( "Welcome to the GradeBook for\n%s!\n",
32          getCourseName() );
33    } // fim do método displayMessage
34 } // fim da classe GradeBook
```

Figura 3.10 | A classe `GradeBook` com um construtor para inicializar o nome do curso.

As linhas 9–12 declaram o construtor de `GradeBook`. Como um método, a lista de parâmetros de um construtor especifica os dados que ele requer para realizar sua tarefa. Ao criar um novo objeto (como faremos na Figura 3.11), esses dados são colocados nos parênteses depois do nome da classe. A linha 9 indica que o construtor tem um parâmetro `String` chamado `name`. O `name` passado para o construtor é atribuído à variável de instância `courseName` na linha 11.

A Figura 3.11 inicializa os objetos `GradeBook` utilizando o construtor. As linhas 11–12 criam e inicializam um objeto `GradeBook` `gradeBook1`. O construtor de `GradeBook` é chamado com o argumento `"CS101 Introduction to Java Programming"` para inicializar o nome do curso. A expressão de criação de instância de classe nas linhas 11–12 retorna uma referência ao novo objeto, que é atribuído à variável `gradeBook1`. As linhas 13–14 repetem esse processo, dessa vez passando o argumento `"CS102 Data Structures in Java"` para inicializar o nome do curso de `gradeBook2`. As linhas 17–20 utilizam o método `getCourseName` de cada objeto para obter os nomes de curso e mostrar que eles foram inicializados quando os objetos foram criados. A saída confirma que cada `GradeBook` mantém sua própria cópia da variável de instância `courseName`.

```
1  // Figura 3.11: GradeBookTest.java
2  // construtor GradeBook utilizado para especificar o nome
3  // do curso na hora em que cada objeto GradeBook é criado.
4
5  public class GradeBookTest
6  {
7     // método main inicia a execução de programa
8     public static void main( String[] args )
9     {
10       // cria objeto GradeBook
11       GradeBook gradeBook1 = new GradeBook(
12          "CS101 Introduction to Java Programming" );
13       GradeBook gradeBook2 = new GradeBook(
14          "CS102 Data Structures in Java" );
15
16       // exibe valor inicial de courseName para cada GradeBook
17       System.out.printf( "gradeBook1 course name is: %s\n",
18          gradeBook1.getCourseName() );
19       System.out.printf( "gradeBook2 course name is: %s\n",
20          gradeBook2.getCourseName() );
21    } // fim de main
22 } // fim da classe GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Figura 3.11 | O construtor de `GradeBook` usado para especificar o nome do curso no momento em que cada objeto `GradeBook` é criado.

Uma diferença importante entre construtores e métodos é que os construtores não podem retornar valores, portanto, não podem especificar um tipo de retorno (nem mesmo `void`). Normalmente, os construtores são declarados `public`. Se uma classe não incluir um construtor, as variáveis de instância da classe são inicializadas como seus valores padrão. *Se você declarar qualquer construtor para uma classe, o compilador Java não criará um construtor padrão para essa classe.* Assim, não é mais possível criar um objeto `GradeBook` com `new GradeBook()` como fizemos nos exemplos anteriores.

Dica de prevenção de erro 3.2

A menos que a inicialização padrão de variáveis de instância de sua classe seja aceitável, forneça um construtor para assegurar que as variáveis de instância da sua classe sejam adequadamente inicializadas com valores significativos quando cada novo objeto de sua classe for criado.

Adicionando o construtor ao diagrama de classe UML da classe GradeBook

O diagrama de classe UML da Figura 3.12 modela a classe `GradeBook` da Figura 3.10, que tem um construtor que possui um parâmetro `name` de tipo `String`. Assim como operações, a UML modela construtores no terceiro compartimento de uma classe em um diagrama de classe. Para distinguir entre um construtor e operações de uma classe, a UML requer que a palavra "constructor" seja colocada entre **aspas francesas (« e »)** antes do nome do construtor. É habitual listar construtores antes de outras operações no terceiro compartimento.

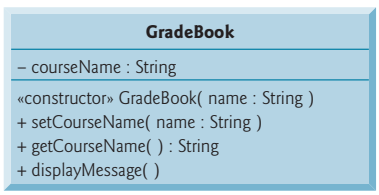


Figura 3.12 | Diagrama de classe UML que indica que a classe `GradeBook` tem um construtor com um parâmetro `name` de tipo UML `String`.

Construtores com múltiplos parâmetros

Às vezes é recomendável inicializar objetos com múltiplos dados. Por exemplo, o Exercício 3.11 solicita que você armazene o nome do curso *e* o nome do instrutor em um objeto `GradeBook`. Nesse caso, o construtor de `GradeBook` seria modificado para receber duas `String`s, como em:

```
public GradeBook( String courseName, String instructorName )
```

e você chamaria o construtor de `GradeBook` desta maneira:

```
GradeBook gradeBook = new GradeBook(
   "CS101 Introduction to Java Programming", "Sue Green" );
```

3.8 Números de ponto flutuante e tipo `double`

Agora, temporariamente deixamos de lado nosso estudo de caso `GradeBook` para declarar uma classe `Account` que mantém o saldo de uma conta bancária. A maioria dos saldos de conta não é constituída de números inteiros (por exemplo, 0, –22 e 1.024). Por essa razão, a classe `Account` representa o saldo de conta como um **número de ponto flutuante** (isto é, um número com um ponto de fração decimal, como 7,33, 0,0975 ou 1.000,12345). O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória — `float` e `double`. A principal diferença entre eles é que as variáveis `double` podem armazenar números com maior magnitude e mais detalhes (isto é, mais dígitos à direita do ponto de fração decimal — também conhecido como a **precisão** do número) do que as variáveis `float`.

Precisão de número de ponto flutuante e requisitos de memória

As variáveis do tipo **`float`** representam **números de ponto flutuante de precisão simples** e podem representar até sete dígitos significativos. As variáveis do tipo **`double`** representam **números de ponto flutuante de precisão dupla**. Essas requerem duas vezes a quantidade de memória das variáveis `float` e fornecem 15 dígitos significativos — aproximadamente o dobro da precisão de variáveis `float`. Para o intervalo de valores requerido pela maioria dos programas, as variáveis de tipo `float` devem bastar, mas você pode utilizar `double` para "trabalhar com segurança". Em alguns aplicativos, mesmo as variáveis do tipo `double` serão inadequadas. A maioria dos programadores representa números de ponto flutuante com o tipo `double`. De fato, o Java trata todos os números de ponto flutuante que você digita no código-fonte de um programa (como 7,33 e 0,0975) como valores `double` por padrão. Esses valores no código-fonte são conhecidos como **literais de ponto flutuante**. Veja o Apêndice D, Tipos primitivos, para informações sobre os intervalos de valores de `float`s e `double`s.

Embora os números de ponto flutuante não sejam sempre 100% precisos, eles têm numerosas aplicações. Por exemplo, quando falamos de uma temperatura "normal" de corpo de 98,6 ºF não precisamos ser tão precisos a ponto de envolver um grande número de dígitos. Quando lemos a temperatura de 98,6 ºF em um termômetro, ela pode, de fato, ser 98,5999473210643 ºF. Chamar simplesmente esse número de 98,6 ºF serve para a maioria dos aplicativos que medem temperaturas de corpo. Devido à natureza imprecisa dos números de ponto flutuante, o tipo `double` é preferido ao tipo `float`, porque as variáveis `double` podem representar números de ponto flutuante com mais exatidão. Por essa razão, utilizamos principalmente o tipo `double` por todo o livro. Para números de ponto flutuante precisos, o Java fornece a classe `BigDecimal` (pacote `java.math`).

Os números de ponto flutuante também surgem como resultado de divisão. Na aritmética convencional, quando dividimos 10 por 3, o resultado é 3,3333333..., com a sequência de 3 repetindo-se infinitamente. O computador aloca apenas uma quantidade fixa de espaço para armazenar tal valor, portanto evidentemente o valor de ponto flutuante armazenado somente pode ser uma aproximação.

Erro comum de programação 3.4

Utilizar números de ponto flutuante de uma maneira que assuma que eles são representados precisamente pode levar a resultados incorretos.

A classe `Account` com uma variável de instância do tipo `double`

Nosso próximo aplicativo (figuras 3.13–3.14) contém uma classe chamada `Account` (Figura 3.13) que mantém o saldo de uma conta bancária. Um banco típico atende muitas contas, cada uma com um saldo próprio, portanto a linha 7 declara uma variável de instância chamada `balance` do tipo `double`. A variável `balance` é uma variável de instância porque é declarada no corpo da classe, mas fora das declarações de método da classe (linhas 10–16, 19–22 e 25–28). Cada instância (isto é, objeto) de classe `Account` contém sua própria cópia de `balance`.

```
1  // Figura 3.13: Account.java
2  // classe Account com um construtor para validar e
3  // inicializa a variável de instância balance do tipo double.
4
5  public class Account
6  {
```

```
7       private double balance; // variável de instância que armazena o saldo
8
9       // construtor
10      public Account( double initialBalance )
11      {
12         // valida que initialBalance é maior que 0,0;
13         // se não, o saldo é inicializado como o valor padrão 0.0
14         if ( initialBalance > 0.0 )
15            balance = initialBalance;
16      } // fim do construtor Account
17
18      // credita (adiciona) uma quantia à conta
19      public void credit( double amount )
20      {
21         balance = balance + amount; // adiciona quantia ao saldo
22      } // fim do método credit
23
24      // retorna o saldo da conta
25      public double getBalance()
26      {
27         return balance; // fornece o valor de saldo ao método chamador
28      } // fim do método getBalance
29   } // fim da classe Account
```

Figura 3.13 | A classe `account` com um construtor para validar e inicializar a variável de instância `balance` do tipo `double`.

A classe tem um construtor e dois métodos. É comum que alguém que abre uma conta deposite o dinheiro imediatamente, assim o construtor (linhas 10–16) recebe um parâmetro `initialBalance` do tipo `double` que representa o saldo inicial. As linhas 14–15 asseguram que `initialBalance` seja maior do que `0.0`. Se for, o valor de `initialBalance` é atribuído à variável de instância `balance`. Caso contrário, `balance` permanece em `0.0` — seu valor inicial padrão.

O método `credit` (linhas 19–22) não retorna quaisquer dados quando ele completa sua tarefa, portanto, seu tipo de retorno é `void`. O método recebe um parâmetro chamado `amount` — um valor `double` que será adicionado ao saldo. A linha 21 adiciona `amount` ao valor atual de `balance`, então atribui o resultado ao `balance` (substituindo assim a quantia do saldo anterior).

O método `getBalance` (linhas 25–28) permite aos clientes da classe (isto é, outras classes que utilizam essa classe) obter o valor do `balance` de um objeto `Account` particular. O método especifica o tipo de retorno `double` e uma lista vazia de parâmetros.

Mais uma vez, observe que as instruções nas linhas 15, 21 e 27 utilizam a variável de instância `balance` mesmo se ela não tiver sido declarada em nenhum dos métodos. Podemos utilizar `balance` nesses métodos porque ele é uma variável de instância da classe.

A classe `AccountTest` para utilizar a classe `Account`

A classe `AccountTest` (Figura 3.14) cria dois objetos `Account` (linhas 10–11) e os inicializa com `50.00` e `-7.53`, respectivamente. As linhas 14–17 geram a saída do saldo em cada `Account` chamando o método `getBalance` da `Account`. Quando o método `getBalance` é chamado por `account1` a partir da linha 15, o valor do saldo da `account1` é retornado da linha 27 da Figura 3.13 e exibido pela instrução `System.out.printf` (Figura 3.14, linhas 14–15). De maneira semelhante, quando o método `getBalance` for chamado por `account2` da linha 17, o valor do saldo da `account2` é retornado da linha 27 da Figura 3.13 e exibido pela instrução `System.out.printf` (Figura 3.14, linhas 16–17). Observe que o saldo de `account2` é `0.00`, porque o construtor assegurou que a conta não poderia iniciar com um saldo negativo. A saída do valor é gerada por `printf` com o especificador de formato `%.2f`. O especificador de formato **`%f`** é utilizado para gerar saída de valores de tipo `float` ou `double`. O `.2` entre `%` e `f` representa o número de casas decimais (2) que devem ser enviadas para a saída à direita do ponto de fração decimal no número de ponto flutuante — também conhecido como a **precisão** do número. Qualquer saída de valor de ponto flutuante com `%.2f` será arredondada para a casa decimal dos centésimos — por exemplo, 123,457 seria arredondado para 123,46; 27,333 seria arredondado para 27,33 e 123,455 para 123,46.

```
1   // Figura 3.14: AccountTest.Java
2   // Entrada e saída de números de ponto flutuante com objetos Account.
3   import java.util.Scanner;
4
5   public class AccountTest
6   {
7      // método main inicia a execução do aplicativo Java
8      public static void main( String[] args )
9      {
10        Account account1 = new Account( 50.00 ); // cria o objeto Account
11        Account account2 = new Account( -7.53 ); // cria o objeto Account
```

```
12
13          // exibe o saldo inicial de cada objeto
14          System.out.printf( "account1 balance: $%.2f \n",
15             account1.getBalance() );
16          System.out.printf( "account2 balance: $%.2f \n\n",
17             account2.getBalance() );
18
19          // cria Scanner para obter entrada a partir da janela de comando
20          Scanner input = new Scanner( System.in );
21          double depositAmount; // quantia de depósito lida a partir do usuário
22
23          System.out.print( "Enter deposit amount for account1: " ); // prompt
24          depositAmount = input.nextDouble(); // entrada do usuário
25          System.out.printf( "\nadding %.2f to account1 balance\n\n",
26             depositAmount );
27          account1.credit( depositAmount ); // adiciona o saldo de account1
28
29          // exibe os saldos
30          System.out.printf( "account1 balance: $%.2f \n",
31             account1.getBalance() );
32          System.out.printf( "account2 balance: $%.2f \n\n",
33             account2.getBalance() );
34          // prompt
35          System.out.print( "Enter deposit amount for account2: " );
36          depositAmount = input.nextDouble(); // entrada do usuário
37          System.out.printf( "\nadding %.2f to account2 balance\n\n",
38             depositAmount );
39          account2.credit( depositAmount ); // adiciona ao saldo
40                                            // de account2
41          // exibe os saldos
42          System.out.printf( "account1 balance: $%.2f \n",
43             account1.getBalance() );
44          System.out.printf( "account2 balance: $%.2f \n",
45             account2.getBalance() );
46       } // fim de main
47    } // fim da classe AccountTest
```

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Figura 3.14 | Entrada e saída de números de ponto flutuante com objetos `Account`.

A linha 21 declara a variável local `depositAmount` para armazenar cada quantia de depósito inserida pelo usuário. Ao contrário da variável de instância `balance` na classe `Account`, a variável local `depositAmount` em `main` não é inicializada como 0.0 por padrão. Entretanto, essa variável não precisa ser inicializada aqui porque seu valor será determinado pela entrada do usuário.

A linha 23 pede ao usuário para inserir uma quantia de depósito para `account1`. A linha 24 obtém a entrada do usuário chamando o método **`nextDouble`** do objeto `input` de `Scanner`, que retorna um valor `double` inserido pelo usuário. As linhas 25–26 exibem o quanto foi depositado. A linha 27 chama o método `credit` do objeto `account1` e armazena `depositAmount` como o argumento do método. Quando o método é chamado, o valor do argumento é atribuído ao parâmetro `amount` (linha 19 da Figura 3.13) do método `credit` (linhas 19–22 da Figura 3.13), então o método `credit` adiciona esse valor ao `balance` (linha 21 da Figura 3.13). As linhas 30–33 (Figura 3.14) geram a saída dos saldos de ambas as `Accounts` para mostrar novamente que apenas o saldo de `account1` mudou.

A linha 35 pede ao usuário para inserir uma quantia de depósito para `account2`. A linha 36 obtém a entrada do usuário chamando o método `nextDouble` do objeto `Scanner input`. As linhas 37–38 exibem o quanto foi depositado. A linha 39 chama o método `credit`

do objeto `account2` e armazena `depositAmount` como o argumento do método; em seguida, o método `credit` adiciona esse valor ao saldo. Por fim, as linhas 42–45 geram a saída dos saldos de ambas as `Accounts` para mostrar novamente que somente o saldo de `account2` mudou.

Diagrama de classes UML para a classe `Account`

O diagrama de classes UML na Figura 3.15 modela a classe `Account` da Figura 3.13. O diagrama modela o atributo `private balance` com o tipo UML `Double` para corresponder à variável de instância `balance` da classe do tipo Java `double`. O diagrama modela o construtor da classe `Account` com um parâmetro `initialBalance` do tipo UML `Double` no terceiro compartimento da classe. Os dois métodos `public` da classe também são modelados como operações no terceiro compartimento. O diagrama modela a operação `credit` com um parâmetro `amount` do tipo UML `Double` (porque o método correspondente tem um parâmetro `amount` do tipo Java `double`) e a operação `getBalance` com um tipo de retorno de `Double` (porque o método Java correspondente retorna um valor `double`).

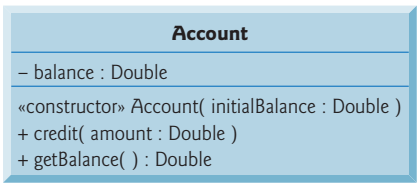


Figura 3.15 | O diagrama de classes UML indicando que a classe `Account` tem um atributo `private balance` do tipo UML `Double`, um construtor (com um parâmetro do tipo UML `Double`) e duas operações `public` — `credit` (com um parâmetro `amount` do tipo UML `Double`) e `getBalance` (retorna o tipo UML `Double`).

3.9 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando caixas de diálogo

Esse estudo de caso opcional é projetado para aqueles que querem começar a aprender as poderosas capacidades do Java para criar interfaces gráficas com o usuário (Graphical User Interfaces — GUIs) e as imagens gráficas iniciais do livro, antes das principais discussões desses tópicos no Capítulo 14, "Componentes GUI: Parte 1", Capítulo 15, "Imagens gráficas e Java2D", e Capítulo 25, "Componentes GUI: Parte 2".

O estudo de caso sobre GUI e imagens gráficas aparece em 10 seções curtas (ver Figura 3.16). Cada seção introduz vários novos conceitos e fornece exemplos de capturas de tela que mostram interações e resultados de exemplo. Nas poucas primeiras seções, você criará seus primeiros aplicativos gráficos. Nas próximas seções, você utilizará conceitos da programação orientada a objetos para criar um aplicativo que desenha várias formas. Ao introduzirmos as GUIs formalmente no Capítulo 14, usamos o mouse para escolher exatamente que formas desenhar e onde as desenhar. No Capítulo 15, adicionamos capacidades gráficas da Java 2 API para desenhar as formas com diferentes espessuras de linha e preenchimentos. Esperamos que esse estudo de caso seja informativo e divertido para você.

Posição	Título — Exercício(s)
Seção 3.9	Utilizando caixas de diálogo — Entrada e saída básicas com caixas de diálogo
Seção 4.14	Criando desenhos simples — exibindo e desenhando linhas na tela
Seção 5.10	Desenhando retângulos e ovais — Utilizando formas para representar dados
Seção 6.13	Cores e formas preenchidas — Desenhando um alvo e imagens gráficas aleatórias
Seção 7.15	Desenhando arcos — desenhando espirais com arcos
Seção 8.16	Utilizando objetos com elementos gráficos — armazenando formas como objetos
Seção 9.8	Exibindo texto e imagens utilizando rótulos — Fornecendo informações de status
Seção 10.8	Desenhando com polimorfismo — identificando as semelhanças entre as formas
Exercícios 14.17	Expandindo a interface — Utilizando componentes GUI e tratamento de evento
Exercícios 15.31	Adicionando Java 2D — Utilizando a API Java 2D para aprimorar desenhos

Figura 3.16 | Resumo dos estudos de caso de GUI e imagens gráficas em cada capítulo.

Exibindo texto em uma caixa de diálogo

Os programas apresentados até agora exibem a saída na janela de comando. Muitos aplicativos utilizam janelas ou **caixas de diálogo** (também chamadas **diálogos**) para exibir a saída. Navegadores Web, como o Mozilla Firefox e o Microsoft Internet Explorer exibem páginas Web em janelas separadas. Os programas de correio eletrônico permitem digitar e ler mensagens em uma janela. Tipicamente, as caixas de diálogo são janelas nas quais os programas exibem mensagens importantes aos usuários. A classe **`JOptionPane`** fornece caixas de diálogo pré-construídas que permitem aos programas exibir janelas que contêm mensagens — essas janelas são chamadas de **diálogos de mensagem**. A Figura 3.17 exibe a `String` `"Welcome \nto\nJava"` em uma caixa de diálogo de mensagem.

```
1  // Figura 3.17: Dialog1.java
2  // Imprimindo múltiplas linhas na caixa de diálogo.
3  import javax.swing.JOptionPane; // importa classe JOptionPane
4
5  public class Dialog1
6  {
7     public static void main( String[] args )
8     {
9        // exibe um diálogo com uma mensagem
10       JOptionPane.showMessageDialog( null, "Welcome\nto\nJava" );
11    } // fim de main
12 } // fim da classe Dialog1
```

Figura 3.17 | Utilizando `JOptionPane` para exibir múltiplas linhas em uma caixa de diálogo.

A linha 3 indica que o programa utiliza a classe `JOptionPane` do pacote **`javax.swing`**. Esse pacote contém muitas classes que ajudam-lhe a criar **interfaces gráficas com o usuário** (Graphical User Interfaces — GUIs) para aplicativos. **Componentes GUI** facilitam a entrada de dados pelo usuário de um programa e apresentação das saídas ao usuário. A linha 10 chama o método `JOptionPane` **`showMessageDialog`** para exibir uma caixa de diálogo que contém uma mensagem. O método requer dois argumentos. O primeiro ajuda o aplicativo Java a determinar onde posicionar a caixa de diálogo. Um diálogo é tipicamente exibido a partir de um aplicativo GUI em uma janela própria. O primeiro argumento refere-se a essa janela (conhecida como a janela pai) e faz o diálogo aparecer centralizado na janela do aplicativo. Se o primeiro argumento for `null`, a caixa de diálogo será exibida no centro da tela. O segundo argumento é a `String` a ser exibida na caixa de diálogo.

Introduzindo métodos `static`

O método `JOptionPane` `showMessageDialog` é o chamado de **método `static`**. Esses métodos muitas vezes definem tarefas frequentemente utilizadas. Por exemplo, muitos programas exibem caixas de diálogo, e o código para fazer isso sempre é o mesmo. Em vez de exigir que você "reinvente a roda" e crie o código para exibir uma caixa de diálogo, os projetistas da classe `JOptionPane` declararam um método `static` que realiza essa tarefa para você. Em geral, um método `static` é chamado utilizando-se seu nome de classe seguido por um ponto (`.`) e o nome de método, como em:

NomeDaClasse`.`*nomeDoMétodo*`(` *argumentos* `)`

Note que você não cria um objeto da classe `JOptionPane` para utilizar seu método `static` `showMessageDialog`. Discutiremos métodos `static` mais detalhadamente no Capítulo 6.

Inserindo texto em uma caixa de diálogo

A Figura 3.18 utiliza outra caixa de diálogo `JOptionPane` predefinida chamada **caixa de diálogo de entrada** que permite ao usuário inserir dados em um programa. O programa solicita o nome do usuário e responde com uma caixa de diálogo de mensagem que contém uma saudação e o nome que o usuário inseriu.

```
1  // Figura 3.18: NameDialog.Java
2  // Entrada básica com uma caixa de diálogo.
3  import javax.swing.JOptionPane;
4
5  public class NameDialog
6  {
7     public static void main( String[] args )
8     {
9        // pede para o usuário inserir seu nome
10       String name =
11          JOptionPane.showInputDialog( "What is your name?" );
12
13       // cria a mensagem
14       String message =
15          String.format( "Welcome, %s, to Java Programming!", name );
16
17       // exibe a mensagem para cumprimentar o usuário pelo nome
18       JOptionPane.showMessageDialog( null, message );
19    } // fim de main
20 } // termina NameDialog
```

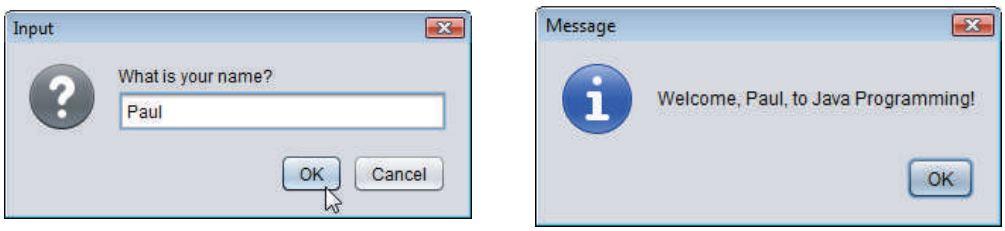


Figura 3.18 | Obtendo a entrada de usuário a partir de um diálogo.

As linhas 10–11 usam o método **`showInputDialog`** de `JOptionPane` para exibir uma caixa de diálogo de entrada que contém um prompt e um campo (conhecido como **campo de texto**) no qual o usuário pode inserir o texto. O argumento do método `showInputDialog` é o prompt que indica o nome que o usuário deve inserir. O usuário digita caracteres no campo de texto, depois clica no botão **OK** ou pressiona a tecla Enter para retornar a `String` para o programa. O método `showInputDialog` (linha 11) retorna uma `String` contendo os caracteres digitados pelo usuário. Armazenamos a `String` na variável `name` (linha 10). [*Nota:* se você pressionar o botão do diálogo **Cancel** ou pressionar a tecla *Esc*, o método retornará `null` e o programa exibirá a palavra "null" como nome.]

As linhas 14–15 usam o método `static String` **`format`** para retornar uma `String` que contém uma saudação com o nome do usuário. O método `format` funciona como o método `System.out.printf`, exceto que `format` retorna a `String` formatada em vez de exibi-la em uma janela de comando. A linha 18 exibe a saudação em uma caixa de diálogo de mensagem, assim como fizemos na Figura 3.17.

Exercício do Estudo de caso GUI e imagens gráficas

3.1 Modifique o programa de adição na Figura 2.7 para utilizar entrada e saída baseadas em caixas de diálogo com os métodos da classe `JOptionPane`. Uma vez que o método `showInputDialog` retorna uma `String`, você deve converter a `String` que o usuário insere em um `int` para utilização em cálculos. O método `static parseInt` da classe `Integer` recebe um argumento `String` que representa um número inteiro (por exemplo, o resultado de `JOptionPane.showInputDialog`) e retorna o valor como um `int`. O método `parseInt` é um método `static` da classe `Integer` (do pacote `java.lang`). Observe que, se a `String` não contiver um inteiro válido, o programa terminará com um erro.

3.10 Conclusão

Neste capítulo, você aprendeu os conceitos básicos de classes, objetos, métodos e variáveis de instância — esses conceitos serão utilizados na maioria dos aplicativos Java que você criar. Em particular, você aprendeu a declarar variáveis de instância de uma classe para manter os dados para cada objeto da classe e aprendeu a declarar métodos que operam sobre esses dados. Você aprendeu a chamar um método para instruí-lo a executar a tarefa e a passar informações para métodos como argumentos. Você aprendeu a diferença entre variável local de um método e variável de instância de uma classe e que apenas variáveis de instância são inicializadas automaticamente. Você também aprendeu a utilizar um construtor da classe para especificar os valores iniciais para as variáveis de instância de um objeto. Por todo o capítulo, você viu como a UML pode ser utilizada para criar diagramas de classe que modelam os construtores, métodos e atributos de classes. Por fim, você aprendeu a lidar com os números de ponto flutuante — como armazená-los com variáveis de tipo primitivo `double`, como inseri-los com um objeto `Scanner`, e como formatá-los com `printf` e o especificador `%f` para propósitos de exibição. No próximo capítulo iniciamos nossa introdução às instruções de controle, que especificam a ordem em que as ações de um programa são realizadas. Você as utilizará em seus métodos para especificar como eles devem realizar suas tarefas.

Resumo

Seção 3.2 Classes, objetos, métodos e variáveis de instância

- Para realizar uma tarefa em um programa é necessário um método. Dentro do método você insere os mecanismos para que ele faça suas tarefas.
- A unidade de programa que abriga um método é chamada de classe. Uma classe pode conter um ou mais métodos que são projetados para realizar as tarefas da classe.
- Um método pode realizar uma tarefa e retornar um resultado.
- Uma classe pode ser utilizada para criar uma instância da classe chamada de objeto.
- Uma chamada de método instrui um método a executar sua tarefa.
- Todo método pode especificar parâmetros que representam informações adicionais que o método exige para realizar corretamente sua tarefa. Uma chamada de método fornece argumentos aos parâmetros do método.
- Um objeto tem atributos que são carregados com o objeto quando ele é utilizado em um programa. Esses atributos são especificados como parte da classe do objeto. Os atributos são especificados em classes por campos.

Seção 3.3 Declarando uma classe com um método e instanciando um objeto de uma classe

- Toda declaração de classe que inicia com o modificador de acesso `public` deve ser armazenada em um arquivo que tem exatamente o mesmo nome que a classe e termina com a extensão de nome do arquivo `.java`.
- Cada declaração de classe contém a palavra-chave `class` seguida imediatamente do nome da classe.
- Uma declaração de método que começa com a palavra-chave `public` indica que o método pode ser chamado por outras classes declaradas fora da declaração de classe.
- A palavra-chave `void` indica que um método realizará uma tarefa, mas não retornará nenhuma informação.
- Por convenção, os nomes de método iniciam com a primeira letra minúscula e todas as palavras subsequentes no nome iniciam com a primeira letra maiúscula.
- Os parênteses vazios que seguem um nome de método indicam que o método não requer nenhum parâmetro para realizar sua tarefa.
- O corpo de todos os métodos é delimitado pelas chaves esquerda e direita (`{` e `}`).
- O corpo do método contém instruções que realizam a tarefa do método. Depois que as instruções executam, o método completou sua tarefa.
- Quando você tentar executar uma classe, o Java procura o método `main` da classe para iniciar a execução.
- Em geral, você não pode chamar um método de outra classe até criar um objeto dessa classe.
- As expressões de criação de instância de classe que iniciam com palavra-chave `new` criam novos objetos.
- Para chamar um método de um objeto, o nome da variável deve ser seguido de um separador ponto (`.`), do nome de método e de um conjunto de parênteses que contém os argumentos do método.
- Na UML, cada classe é modelada em um diagrama de classe como um retângulo com três compartimentos. Aquele na parte superior contém o nome da classe centralizado horizontalmente em negrito. O do meio contém os atributos da classe, que correspondem a campos em Java. O inferior contém as operações da classe, que correspondem a métodos e construtores em Java.
- A UML modela operações listando o nome da operação seguido por um conjunto de parênteses. Um sinal de adição (+) na frente do nome da operação indica que é uma operação `public` na UML (isto é, um método `public` em Java).

Seção 3.4 Declarando um método com um parâmetro

- Os métodos costumam exigir informações adicionais para realizar suas tarefas. Essas informações adicionais são fornecidas para os métodos via argumentos em chamadas de método.
- O método `Scanner` `nextLine` lê os caracteres até um caractere de nova linha ser encontrado, depois retorna os caracteres como um método `String`.
- O método `Scanner` `next` lê os caracteres até um caractere de espaço em branco ser encontrado, então retorna os caracteres como uma `String`.
- Um método que requer dados para realizar sua tarefa deve especificar isso em sua declaração colocando informações adicionais na lista de parâmetros do método.
- Todo parâmetro deve especificar um tipo e um identificador.
- Na hora em que um método é chamado, seus argumentos são atribuídos a seus parâmetros. Então o corpo de método utiliza as variáveis de parâmetro para acessar os valores de argumento.
- Um método especifica múltiplos parâmetros em uma lista separada por vírgula.
- O número de argumentos na chamada de método deve corresponder ao número de parâmetros na lista de parâmetros da declaração do método. Além disso, os tipos de argumento na chamada de método devem ser consistentes com os tipos dos parâmetros correspondentes na declaração do método.
- A classe `String` está no pacote `java.lang`, que é importado implicitamente em todos os arquivos de código-fonte.
- Há um relacionamento especial entre as classes que são compiladas no mesmo diretório no disco. Por padrão, essas classes são consideradas como estando no mesmo pacote. As classes do mesmo pacote são importadas implicitamente para os arquivos de código-fonte de outras classes do mesmo pacote.

- Declarações `import` não são necessárias se você sempre utilizar nomes de classe totalmente qualificados.
- A UML modela um parâmetro de uma operação listando o nome do parâmetro, seguido por um caractere de dois-pontos e o tipo de parâmetro entre os parênteses depois do nome de operação.
- A UML tem seus próprios tipos de dados semelhantes aos do Java. Nem todos os tipos de dados UML têm os mesmos nomes que os tipos Java correspondentes.
- O tipo `String` da UML corresponde ao tipo `String` do Java.

Seção 3.5 Variáveis de instância, métodos set *e* get

- As variáveis declaradas no corpo de um método são variáveis locais e só podem ser utilizadas nesse método.
- Normalmente, uma classe consiste em um ou mais métodos que manipulam os atributos (dados) que pertencem a um objeto particular da classe. Essas variáveis são chamadas campos e estão declaradas dentro de uma declaração de classe, mas fora do corpo das declarações de método da classe.
- Quando cada objeto de uma classe mantém sua própria cópia de um atributo, o campo correspondente é conhecido como uma variável de instância.
- As variáveis ou métodos declarados com o modificador de acesso `private` só são acessíveis a métodos da classe em que são declarados.
- A declaração de variáveis de instância com o modificador de acesso `private` é conhecida como ocultamento de dados.
- Um benefício dos campos é que todos os métodos da classe podem usar os campos. Outra distinção entre um campo e uma variável local é que o campo tem um valor inicial padrão fornecido pelo Java quando você não especifica o valor inicial do campo, mas uma variável local não.
- O valor padrão de um campo do tipo `String` (ou qualquer outro tipo por referência) é `null`.
- Quando um método que especifica um tipo de retorno for chamado e completar sua tarefa, o método retornará um resultado para seu método chamador.
- As classes costumam fornecer métodos `public` para permitir aos clientes da classe configurar (*set*) ou obter (*get*) variáveis de instância `private`. Os nomes desses métodos não precisam iniciar com *set* ou *get*, mas essa convenção de atribuição de nomes é recomendada e requerida para componentes de software Java especiais chamados JavaBeans.
- A UML representa variáveis de instância como um nome de atributo, seguido por um dois-pontos e o tipo.
- Os atributos privados são precedidos por um sinal de subtração (–) na UML.
- A UML indica o tipo de retorno de uma operação colocando um dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação.
- Os diagramas de classe UML não especificam tipos de retorno para operações que não retornam valores.

Seção 3.6 Tipos primitivos versus *tipos por referência*

- Tipos no Java são divididos em duas categorias — tipos primitivos e tipos por referência. Os tipos primitivos são `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` e `double`. Todos os outros são tipos por referência; portanto, classes, que especificam os tipos de objetos, são tipos por referência.
- Uma variável de tipo primitivo pode armazenar exatamente um valor de seu tipo declarado por vez.
- As variáveis de instância de tipo primitivo são inicializadas por padrão. Variáveis dos tipos `byt`, `char`, `short`, `int`, `long`, `float` e `double` são inicializadas como `0`. As variáveis de tipo `boolean` são inicializadas como `false`.
- Os programas utilizam as variáveis de tipos por referência (chamadas de referências) armazenam a localização de um objeto na memória do computador. Essas variáveis referenciam objetos no programa. O objeto que é referenciado pode conter muitas variáveis de instância e métodos.
- Os campos de tipo por referência são inicializados por padrão como o valor `null`.
- Para invocar métodos de instância de um objeto é necessária uma referência a um objeto. Uma variável de tipo primitivo não referencia um objeto e, portanto, não pode ser utilizada para invocar um método.

Seção 3.7 Inicializando objetos com construtores

- A palavra-chave `new` solicita memória do sistema para armazenar um objeto e então chama o construtor da classe correspondente para inicializar o objeto.
- Um construtor pode ser utilizado para inicializar o objeto de uma classe quando o objeto é criado.
- Os construtores podem especificar parâmetros mas não podem especificar tipos de retorno.
- Se uma classe não definir construtores, o compilador fornecerá um construtor padrão sem parâmetros, e as variáveis de instância da classe serão inicializadas com seus valores padrão.
- Assim como as operações, a UML modela construtores no terceiro compartimento de um diagrama de classe. Para distinguir entre um construtor e operações de uma classe, a UML coloca a palavra "constructor" entre aspas francesas (« e ») antes do nome do construtor.

Seção 3.8 Números de ponto flutuante e tipo `double`

- Número de ponto flutuante é um número com um ponto de fração decimal, como 7,33, 0,0975 ou 1.000,12345. O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória — `float` e `double`. A principal diferença entre esses tipos é que as variáveis `double` podem armazenar números com maior magnitude e maior detalhe (conhecida como precisão do número) do que as variáveis `float`.
- As variáveis de tipo `float` representam números de ponto flutuante de precisão simples e têm sete dígitos significativos. As variáveis de tipo `double` representam números de ponto flutuante de dupla precisão. Essas requerem duas vezes a quantidade de memória das variáveis `float` e fornecem 15 dígitos significativos — aproximadamente o dobro da precisão de variáveis `float`.

- Por padrão, esses literais de ponto flutuante são do tipo `double`.
- O método `scanner` `nextDouble` retorna um valor `double`.
- O especificador de formato `%f` é utilizado para gerar saída de valores de tipo `float` ou `double`. O especificador de formato `%.2f` especifica que dois dígitos da precisão devem ser gerados à direita do ponto decimal no número de ponto flutuante.
- O valor padrão de um campo de tipo `double` é `0.0`, e o valor padrão de um campo de tipo `int` é `0`.

Terminologia

`%f`, especificador de formato, 71
`.`, ponto separador, 60
aspas francesas (« e ») na UML, 69
cabeçalho de método, 59
caixa de diálogo, 74
campo, 63
campo de texto, 75
chamada de método, 57
cliente de um objeto, 66
compartimento, 60
construtor, 60
construtor padrão, 68
declarar um método de uma classe, 58
diagrama de classes na UML, 60
diálogo, 74
diálogo de entrada, 74
diálogo de mensagem, 74
`double`, tipo primitivo, 70
enviar uma mensagem para um objeto, 57
expressão de criação de instância de classe, 60
`float`, tipo primitivo, 70
get, método, 66
instanciar um objeto de uma classe, 58
interface gráfica com usuário (*Graphical User Interface* — GUI), 74
invocar um método, 67
`javax.swing`, pacote, 74
linguagem extensível, 60
lista de parâmetros, 61
literal de ponto flutuante, 70
método chamador, 59
modificador de acesso, 59
`new`, palavra-chave, 60
`next`, método da classe `Scanner`, 62
`nextDouble`, método da classe `Scanner`, 72
`nextLine`, método da classe `Scanner`, 62
nome de classe completamente qualificado, 63
`null`, palavra-chave, 67
número de ponto flutuante de precisão dupla, 70
número de ponto flutuante de precisão simples, 70
obter (*get*) um valor, 66
ocultamento de dados, 64
operação, 60
pacote padrão, 63
parâmetro, 61
ponto flutuante, número, 70
ponto separador (`.`), 60
precisão, 71
precisão de um número de ponto flutuante formatado, 71
`private`, palavra-chave, 64
`public`, palavra-chave, 64
referência, 67
referenciar um objeto, 67
`return`, palavra-chave, 65
set, método, 66
`showInputDialog`, método da classe `JOptionPane`, 75
`showMessageDialog`, método da classe `JOptionPane`, 74
tipo por referência, 67
valor inicial padrão, 65
valor padrão, 65
variável de instância, 58
variável local, 63

Exercícios de autorrevisão

3.1 Preencha as lacunas em cada um dos seguintes:

a) Uma casa está para uma planta arquitetônica assim como um(a) _____ está para uma classe.

b) Toda declaração de classe que inicia com a palavra-chave _____ deve ser armazenada em um arquivo que tem exatamente o mesmo nome que a classe e terminar com a extensão de nome do arquivo `.java`.

c) A palavra-chave _____ em uma declaração de classe é imediatamente seguida pelo nome da classe.

d) A palavra-chave _____ solicita memória do sistema para armazenar um objeto e então chama o construtor da classe correspondente para inicializar o objeto.

e) Todo parâmetro deve especificar um(a) _____ e um(a) _____.

f) Por padrão, as classes que são compiladas no mesmo diretório são consideradas como estando no mesmo pacote, conhecido como _____.

g) Quando cada objeto de uma classe mantém sua própria cópia de um atributo, o campo que representa o atributo também é conhecido como um(a) _____.

h) O Java fornece dois tipos primitivos para armazenar números de ponto flutuante na memória: _____ e _____.

i) As variáveis de tipo `double` representam números de ponto flutuante _____.

j) O método `scanner` _____ retorna um valor `double`.

k) A palavra-chave `public` é um _____ de acesso.

l) O tipo de retorno _____ indica que um método não retornará um valor.

m) O método `Scanner` _____ lê os caracteres de até um caractere de nova linha ser encontrado e então retorna esses caracteres como uma `String`.

n) A classe `String` está no pacote _____.

o) Um(a) _____ não é requerido(a) se você sempre referenciar uma classe por meio do seu nome de classe completamente qualificado.

p) Um(a) _____ é um número com um ponto de fração decimal, como 7,33, 0,0975 ou 1.000,12345.

q) As variáveis de tipo `float` representam números de ponto flutuante _____.

r) O especificador de formato _____ é utilizado para gerar saída de valores de tipo `float` ou `double`.

s) Os tipos no Java são divididos em duas categorias — tipos _____ e tipos _____.

3.2 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Por convenção, os nomes de método iniciam com a primeira letra maiúscula e todas as palavras subsequentes do nome iniciam com a primeira letra maiúscula.

b) Uma declaração `import` não é necessária quando uma classe em um pacote utiliza outra no mesmo pacote.

c) Parênteses vazios que se seguem a um nome de método em uma declaração de método indicam que o método não requer nenhum parâmetro para realizar sua tarefa.

d) As variáveis ou métodos declarados com o modificador de acesso `private` só são acessíveis a métodos da classe em que são declarados.

e) Uma variável de tipo primitivo pode ser utilizada para invocar um método.

f) As variáveis declaradas no corpo de um método particular são conhecidas como variáveis de instância e podem ser utilizadas em todos os métodos da classe.

g) O corpo de todos os métodos é delimitado pelas chaves esquerda e direita (`{` e `}`).

h) As variáveis locais de tipo primitivo são inicializadas por padrão.

i) As variáveis de instância de tipo por referência são inicializadas por padrão para o valor `null`.

j) Qualquer classe que contém `public static void main (String [] args)` pode ser utilizada para executar um aplicativo.

k) O número de argumentos na chamada de método deve corresponder ao número de parâmetros na lista de parâmetros da declaração do método.

l) Os valores de ponto flutuante que aparecem no código-fonte são conhecidos como literais de ponto flutuante e são tipos `float` por padrão.

3.3 Qual diferença entre uma variável local e um campo?

3.4 Explique o propósito de um parâmetro de método. Qual a diferença entre um parâmetro e um argumento?

Respostas dos exercícios de autorrevisão

3.1 a) objeto. b) `public`. c) `class`. d) `new`. e) tipo, nome. f) pacote padrão. g) variável de instância. h) `float`, `double`. i) precisão dupla. j) `nextDouble`. k) modificador. l) `void`. m) `nextLine`. n) `java.lang`. o) declaração `import`. p) número de ponto flutuante. q) precisão única. r) `%f`. s) primitivo, referência.

3.2 a) Falso. Por convenção, os nomes de método iniciam com a primeira letra minúscula e todas as palavras subsequentes no nome iniciam com a primeira letra maiúscula. b) Verdadeiro. c) Verdadeiro. d) Verdadeiro. e) Falso. Uma variável de tipo primitivo não pode ser utilizada para invocar um método — uma referência a um objeto é necessária para que os métodos do objeto possam ser invocados. f) Falso. Essas variáveis são chamadas de variáveis locais e podem ser utilizadas somente no método em que são declaradas. g) Verdadeiro. h) Falso. As variáveis de instância de tipo primitivo são inicializadas por padrão. Deve-se atribuir um valor explicitamente a cada variável local . i) Verdadeiro. j) Verdadeiro. k) Verdadeiro. l) Falso. Esses literais são de tipo `double` por padrão.

3.3 Uma variável local é declarada no corpo de um método e só pode ser utilizada do ponto em que ela é declarada até o fim da declaração do método. Um campo é declarado em uma classe, mas não no corpo de qualquer um dos métodos da classe. Além disso, os campos são acessíveis a todos os métodos da classe. (Veremos uma exceção a isso no Capítulo 8.)

3.4 Um parâmetro representa informações adicionais que um método requer para realizar sua tarefa. Cada parâmetro requerido por um método é especificado na declaração do método. Um argumento é o valor real de um parâmetro de método. Quando um método é chamado, os valores de argumento são passados para os parâmetros correspondentes do método para que ele possa realizar sua tarefa.

Exercícios

3.5 Qual é o propósito da palavra-chave `new`? Explique o que acontece quando você a utiliza.

3.6 O que é um construtor padrão? Como as variáveis de instância de um objeto são inicializadas se uma classe tiver somente um construtor padrão?

3.7 Explique o propósito de uma variável de instância.

3.8 A maioria de classes precisa ser importada antes de poder ser utilizada em um aplicativo. Por que cada aplicativo pode utilizar as classes `System` e `String` sem antes importá-las?

3.9 Explique como um programa pode utilizar a classe `Scanner` sem importar a classe.

3.10 Explique por que uma classe pode fornecer um método *set* e um método *get* para uma variável de instância.

3.11 **(*Classe* `GradeBook` *modificada*)** Modifique a classe `GradeBook` (Figura 3.10) como a seguir:

a) Inclua uma variável de instância `String` que representa o nome do instrutor do curso.

b) Forneça um método *set* para alterar o nome do instrutor e um método *get* para recuperá-lo.

c) Modifique o construtor para especificar dois parâmetros — um para o nome do curso e um para o nome do instrutor.

d) Modifique o método `displayMessage` para gerar saída da mensagem de boas-vindas e do nome do curso, seguido por `"This course is presented by:"` e o nome do professor.

Utilize sua classe modificada em um aplicativo de teste que demonstra as novas capacidades da classe.

3.12 **(*Classe* `Account` *modificada*)** Modifique a classe `Account` (Figura 3.13) para fornecer um método chamado `debit` que retira dinheiro de uma `Account`. Assegure que a quantidade de débito não exceda o saldo de `Account`. Se exceder, o saldo deve ser deixado inalterado e o método deve

imprimir uma mensagem que indica "Debit amount exceeded account balance" [Quantia de débito excedeu o saldo da conta]. Modifique a classe AccountTest (Figura 3.14) para testar o método debit.

3.13 (***Classe Invoice***) Crie uma classe chamada Invoice para que uma loja de suprimentos de informática possa utilizá-la para representar uma fatura de um item vendido na loja. Uma Invoice (fatura) deve incluir quatro partes das informações como variáveis de instância — o número (tipo String), a descrição (tipo String), a quantidade comprada de um item (tipo int) e o preço por item (double). Sua classe deve ter um construtor que inicializa as quatro variáveis de instância. Forneça um método *set* e um *get* para cada variável de instância. Além disso, forneça um método chamado getInvoiceAmount que calcula a quantidade de fatura (isto é, multiplica a quantidade pelo preço por item), e depois retorna a quantidade como um valor double. Se a quantidade não for positiva, ela deve ser configurada como 0. Se o preço por item não for positivo, ele deve ser configurado como 0.0. Escreva um aplicativo de teste chamado InvoiceTest que demonstra as capacidades da classe Invoice.

3.14 (***Classe Employee***) Crie uma classe chamada Employee que inclua três variáveis de instância — um primeiro nome (tipo String), um sobrenome (tipo String) e um salário mensal (double). Forneça um construtor que inicializa as três variáveis de instância. Forneça um método *set* e um *get* para cada variável de instância. Se o salário mensal não for positivo, não configure seu valor. Escreva um aplicativo de teste chamado EmployeeTest que demonstra as capacidades da classe Employee. Crie dois objetos Employee e exiba o salário *anual* de cada objeto. Então dê a cada Employee um aumento de 10% e exiba novamente o salário anual de cada Employee.

3.15 (***Classe Date***) Crie uma classe chamada Date que inclua três variáveis de instância — mês (tipo int), dia (tipo int) e ano (tipo int). Forneça um construtor que inicializa as três variáveis de instância e suponha que os valores fornecidos estejam corretos. Forneça um método *set* e um *get* para cada variável de instância. Forneça um método displayDate que exibe o mês, o dia e o ano separados por barras normais (/). Escreva um aplicativo de teste chamado DateTest que demonstra as capacidades da classe Date.

Fazendo a diferença

3.16 (***Calculadora de frequência cardíaca-alvo***) Ao realizar exercícios físicos, você pode utilizar um monitor de frequência cardíaca para ver se sua frequência permanece dentro de um intervalo seguro, sugerido pelos seus treinadores e médicos. Segundo a American Heart Association (AHA) (www.americanheart.org/presenter.jhtml?identifier=4736), a fórmula para calcular a *frequência cardíaca máxima* por minuto é 220 menos a idade. Sua *frequência cardíaca alvo* é um intervalo entre 50–85% da frequência cardíaca máxima. [*Nota:* essas fórmulas são estimativas fornecidas pela AHA. As frequências cardíacas máximas e o alvo podem variar com base na saúde, capacidade física e sexo da pessoa. Sempre consulte um médico ou profissional de saúde qualificado antes de começar ou modificar um programa de exercícios físicos.] Crie uma classe chamada HeartRates. Os atributos da classe devem incluir o nome, sobrenome e data de nascimento da pessoa (consistindo em atributos separados para o mês, dia e ano de nascimento). Sua classe deve ter um construtor que recebe esses dados como parâmetros. Para cada atributo forneça métodos *set* e *get*. A classe também deve incluir um método que calcula e retorna a idade da pessoa (em anos), um método que calcula e retorna a frequência cardíaca máxima da pessoa e um método que calcula e retorna a frequência cardíaca-alvo da pessoa. Escreva um aplicativo Java que solicite as informações da pessoa, instancie um objeto da classe HeartRates e imprime as informações a partir desse objeto — incluindo o nome, sobrenome e data de nascimento da pessoa — calcule e imprima a idade da pessoa (em anos), intervalo de frequência cardíaca máxima e frequência cardíaca-alvo.

3.17 (***Computadorização dos registros de saúde***) Uma questão relacionada à assistência médica discutida ultimamente nos veículos de comunicação é a computadorização dos registros de saúde. Essa possibilidade está sendo abordada cautelosamente por causa de preocupações quanto à privacidade e segurança de dados sigilosos, entre outros. [Iremos discutir essas preocupações nos exercícios mais adiante.] A computadorização dos registros de saúde pode tornar mais fácil que pacientes compartilhem seus perfis e históricos de saúde entre vários profissionais de saúde. Isso pode aprimorar a qualidade da assistência médica, ajudar a evitar conflitos de medicamentos e prescrições de medicamentos errados, reduzir custos em ambulatórios e poderia salvar vidas. Nesse exercício, você projetará uma classe HealthProfile "inicial" para uma pessoa. Os atributos da classe devem incluir o nome, sobrenome, sexo, data de nascimento (consistindo em atributos separados para o mês, dia e ano de nascimento), altura (em polegadas) e peso (em libras) da pessoa. Sua classe deve ter um construtor que recebe esses dados. Para cada atributo, forneça métodos *set* e *get*. A classe também deve incluir métodos que calculem e retornem a idade do usuário em anos, intervalo de frequência cardíaca máxima e frequência cardíaca-alvo (ver Exercício 3.16) e índice de massa corporal (IMC; ver Exercício 2.33). Escreva um aplicativo Java que solicite as informações da pessoa, instancie um objeto da classe HealthProfile para essa pessoa e imprime as informações a partir desse objeto — incluindo o nome, sobrenome, sexo, data de nascimento, altura e peso da pessoa — e, então, calcule e imprima a idade da pessoa em anos, IMC, intervalo de frequência cardíaca máxima e frequência cardíaca-alvo. Ele também deve exibir o gráfico de "valores IMC" do Exercício 2.33.

Vamos todos dar um passo à frente.
— Lewis Carroll

A roda já deu uma volta completa.
— William Shakespeare

Quantas maçãs não caíram na cabeça de Newton antes de ele ter entendido a pista!
— Robert Frost

Toda a evolução que conhecemos procede do vago para o definido.
— Charles Sanders Peirce

4

Instruções de controle: Parte I

Objetivos

Neste capítulo, você aprenderá :

- A utilizar técnicas básicas de solução de problemas.
- A desenvolver algoritmos por meio do processo de refinamento passo a passo de cima para baixo utilizando pseudocódigo.
- A utilizar instruções de seleção `if` e `if...else` para escolher entre ações alternativas.
- A utilizar a instrução de repetição `while` para executar instruções em um programa repetidamente.
- A utilizar repetição controlada por contador e repetição controlada por sentinela.
- A utilizar os operadores compostos de atribuição, incremento e decremento.
- A portabilidade dos tipos de dados primitivos.

Sumário

4.1 Introdução
4.2 Algoritmos
4.3 Pseudocódigo
4.4 Estruturas de controle
4.5 A instrução de seleção única `if`
4.6 A instrução de seleção dupla `if...else`
4.7 A instrução de repetição `while`
4.8 Formulando algoritmos: repetição controlada por contador
4.9 Formulando algoritmos: repetição controlada por sentinela
4.10 Formulando algoritmos: instruções de controle aninhadas
4.11 Operadores de atribuição composta
4.12 Operadores de incremento e decremento
4.13 Tipos primitivos
4.14 (Opcional) Estudo de caso de GUI e imagens gráficas: criando desenhos simples
4.15 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

4.1 Introdução

Antes de escrever um programa para resolver um problema, você deve ter um entendimento completo do problema e uma abordagem cuidadosamente planejada para resolvê-lo. Ao escrever um programa, você também deve entender os tipos de blocos de construção que estão disponíveis e empregar técnicas de construção do programa comprovadas. Neste capítulo e no Capítulo 5, "Instruções de controle: Parte 2", discutimos essas questões na nossa apresentação da teoria e princípios da programação estruturada. Os conceitos apresentados aqui são cruciais na construção de classes e manipulação de objetos.

Neste capítulo, introduzimos as instruções `if`, `if...else` e `while` do Java, três dos blocos de construção que permitem-lhe especificar a lógica requerida para que os métodos realizem suas tarefas. Dedicamos uma parte deste capítulo (e os capítulos 5 e 7) para desenvolver em mais detalhes a classe `GradeBook` introduzida no Capítulo 3. Em particular, adicionamos um método à classe `GradeBook` que utiliza as instruções de controle para calcular a média de um conjunto de notas de alunos. Um outro exemplo demonstra maneiras adicionais de combinar instruções de controle para resolver um problema semelhante. Apresentamos os operadores de atribuição composta, operadores de incremento e decremento do Java. Por fim, discutimos a portabilidade dos tipos primitivos do Java.

4.2 Algoritmos

Qualquer problema de computação pode ser resolvido executando uma série de ações em uma ordem específica. Um procedimento para resolver um problema em termos

1. das **ações** a executar e
2. da **ordem** em que essas ações executam

chama-se **algoritmo**. O exemplo a seguir demonstra que é importante especificar corretamente a ordem em que as ações executam.

Considere o "algoritmo cresça e brilhe" seguido por um executivo para sair da cama e ir trabalhar: (1) Levantar da cama; (2) tirar o pijama; (3) tomar banho; (4) vestir-se; (5) tomar café da manhã; (6) dirigir o carro até o trabalho. Essa rotina leva o executivo para trabalhar bem preparado para tomar decisões críticas. Suponha que os mesmos passos sejam seguidos em uma ordem um pouco diferente: (1) Levantar da cama; (2) tirar o pijama; (3) vestir-se; (4) tomar banho; (5) tomar café da manhã; (6) dirigir o carro até o trabalho. Nesse caso, nosso executivo chegará ao trabalho completamente molhado. Especificar a ordem em que as instruções (ações) são executadas em um programa é chamado **controle de programa**. Este capítulo investiga o controle de programa utilizando as **instruções de controle** do Java.

4.3 Pseudocódigo

Pseudocódigo é uma linguagem informal que ajuda a desenvolver algoritmos sem a preocupação com os estritos detalhes da sintaxe da linguagem Java. O pseudocódigo que apresentamos é particularmente útil para desenvolver algoritmos que serão convertidos em partes estruturadas de programas Java. O pseudocódigo é similar à língua cotidiana — é conveniente e amigável ao usuário, mas não é uma linguagem de programação de computador real. Você verá um algoritmo escrito em pseudocódigo na Figura 4.5.

O pseudocódigo não é executado nos computadores. Mais exatamente, ele ajuda você a "estudar" um programa antes de tentar escrevê-lo em uma linguagem de programação como Java. Este capítulo fornece vários exemplos de como utilizar o pseudocódigo para desenvolver programas Java.

O estilo de pseudocódigo que apresentamos consiste puramente em caracteres, para que você possa digitar o pseudocódigo convenientemente, utilizando um programa editor de textos qualquer. Um programa de pseudocódigo cuidadosamente preparado pode ser facilmente convertido em um programa Java correspondente.

Normalmente, o pseudocódigo só descreve as instruções que representam as ações que ocorrem depois que você converte um programa no pseudocódigo em Java e depois de o programa ser executado em um computador. Essas ações poderiam incluir entrada, saída ou cálculos. Em geral, não incluímos declarações de variáveis em nosso pseudocódigo, mas alguns programadores escolhem listar variáveis e mencionar seus objetivos no início do pseudocódigo.

4.4 Estruturas de controle

Normalmente, instruções em um programa são executadas uma após a outra na ordem em que são escritas. Esse processo é chamado **execução sequencial**. Várias instruções Java, que discutiremos mais adiante, permitirão que você especifique que a próxima instrução a executar não é necessariamente a próxima na sequência. Isso é chamado de **transferência de controle**.

Durante a década de 1960, tornou-se claro que a utilização indiscriminada de transferências de controle era a raiz de muita dificuldade experimentada por grupos de desenvolvimento de software. A culpa disso é a **instrução `goto`** (utilizada na maioria das linguagens de programação atuais), que permite ao programador especificar uma transferência de controle para um entre vários destinos em um programa. O termo **programação estruturada** tornou-se quase sinônimo de "eliminação de `goto`". [*Nota*: O Java não contém uma instrução `goto`; entretanto, a palavra `goto` é reservada pelo Java e não deve ser utilizada como um identificador em programas].

A pesquisa de Bohm e Jacopini[1] demonstrou ser possível escrever programas sem nenhuma instrução `goto`. O desafio dos programadores na época era mudar seus estilos para "programação sem `goto`". Foi somente em meados da década de 1970 que os programadores começaram a levar a sério a programação estruturada. Os resultados foram impressionantes. Grupos de desenvolvimento de software informaram tempos de desenvolvimento menores, mais frequente cumprimento dos prazos de entrega dos sistemas e mais frequente conclusão dentro do orçamento dos projetos de software. A chave para esses sucessos era, para começar, que programas estruturados eram mais claros, mais fáceis de depurar e modificar e menos propensos a conterem bugs.

O trabalho de Bohm e Jacopini demonstrou que todos os programas poderiam ser escritos em termos de somente três estruturas de controle — a **estrutura de sequência**, a **estrutura de seleção** e a **estrutura de repetição**. Ao introduzirmos as implementações das estruturas de controle do Java, na terminologia da especificação da linguagem Java, nós as chamamos "instruções de controle".

Estrutura de sequência em Java

A estrutura de sequência está incorporada ao Java. A não ser que seja instruído de outra forma, o computador executa as instruções Java uma depois da outra na ordem em que elas são escritas — isto é, em sequência. O **diagrama de atividades** na Figura 4.1 ilustra uma estrutura de sequência típica em que dois cálculos são realizados na ordem. O Java lhe permite ter o número de ações que quiser em uma estrutura de sequência. Como logo veremos, uma única ação pode ser colocada em qualquer lugar; podemos colocar várias ações em sequência.

Um diagrama de atividades UML modela o **fluxo de trabalho** (também chamado **atividade**) de uma parte de um sistema de software. Esses fluxos de trabalho podem incluir uma parte de um algoritmo, como a estrutura de sequência na Figura 4.1. Os diagramas de atividade são compostos de símbolos, como **símbolos de estado de ação** (retângulos com os lados esquerdo e direito substituídos por arcos curvados para fora), **losangos** e **pequenos círculos**. Esses símbolos são conectados por **setas de transição**, que representam o fluxo da atividade — isto é, a ordem em que as ações devem ocorrer.

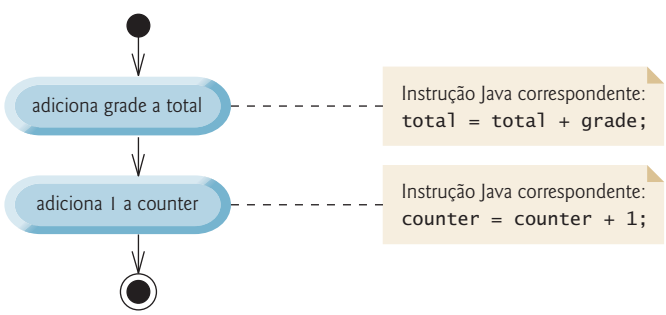


Figura 4.1 | Diagrama de atividades da estrutura de sequência.

Como ocorre com o pseudocódigo, os diagramas de atividades ajudam-lhe a desenvolver e representar algoritmos, embora muitos programadores prefiram o pseudocódigo. Os diagramas de atividades mostram claramente como operam as estruturas de controle. Utilizamos a UML neste capítulo e no Capítulo 5 para mostrar o fluxo de controle em instruções de controle. Nos capítulos 12–13, utilizamos a UML em um estudo de caso de caixa eletrônico real.

[1] BOHM, C.; JACOPINI, G. "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM*, v. 9, n. 5, maio 1966, p. 336–371.

Considere o diagrama de atividade de estrutura de sequência na Figura 4.1. Ele contém dois **estados de ação** que representam ações a ser realizadas. Cada estado da ação contém uma **expressão de ação** — por exemplo, "adicionar grade a total" ou "adicionar 1 a counter" — que especifica uma ação particular a ser realizada. Outras ações poderiam incluir cálculos ou operações de entrada e saída. As setas no diagrama de atividade representam **transições**, as quais indicam a ordem em que as ações representadas pelos estados da ação ocorrem. O programa que implementa as atividades ilustradas pelo diagrama na Figura 4.1 primeiro adiciona `grade` a `total` e então adiciona `1` a `counter`.

O **círculo sólido** na parte superior do diagrama de atividade representa o **estado inicial** — o começo do fluxo de trabalho antes do programa realizar as ações modeladas. O **círculo sólido cercado por um círculo vazio** que aparece na parte inferior do diagrama representa o **estado final** — o fim do fluxo de trabalho depois que o programa realiza suas ações.

A Figura 4.1 também inclui retângulos com os cantos superiores direitos dobrados. Estes são **notas** na UML (como comentários em Java) — observações explanatórias que descrevem o propósito dos símbolos no diagrama. A Figura 4.1 utiliza notas da UML para mostrar o código de Java associado com cada estado de ação. Uma **linha pontilhada** conecta cada nota com o elemento que ele descreve. Os diagramas de atividades normalmente não mostram o código Java que implementa a atividade. Fazemos isso aqui para ilustrar como o diagrama se relaciona ao código Java. Para obter mais informações sobre a UML, consulte o nosso estudo de caso opcional (capítulos 12–13) ou visite `www.uml.org`.

Instruções de seleção em Java

O Java contém três tipos de **instruções de seleção** (discutidos neste capítulo e no Capítulo 5). A instrução `if` realiza uma ação (seleciona) se uma condição for verdadeira ou pula a ação se a condição for falsa. A instrução `if...else` realiza uma ação se uma condição for verdadeira e realiza uma ação diferente se a condição for falsa. A instrução de seleção `switch` (Capítulo 5) realiza uma de muitas ações diferentes, dependendo do valor de uma expressão.

A instrução `if` é uma **instrução de seleção única** porque seleciona ou ignora uma única ação (ou, como veremos a seguir, um único grupo de ações). A instrução `if...else` é chamada **instrução de seleção dupla** porque seleciona entre duas ações diferentes (ou grupos de ações). A instrução `switch` é chamada de **instrução de seleção múltipla** pois seleciona entre muitas ações diferentes (ou grupos de ações).

Instruções de repetição em Java

O Java fornece três **instruções de repetição** (também chamadas **instruções de loop**) que permitem que programas executem instruções repetidamente contanto que uma condição (chamada **condição de continuação do loop**) permaneça verdadeira. As instruções de repetição são as instruções `while`, `do...while` e `for`. (O Capítulo 5 apresenta as instruções `do...while` e `for`.) As instruções `while` e `for` realizam a ação (ou grupo de ações) no seu corpo zero ou mais vezes — se a condição de continuação de loop for inicialmente falsa, a ação (ou grupo de ações) não será executada. A instrução `do...while` realiza a ação (ou grupo de ações) no seu corpo uma ou mais vezes. As palavras `if`, `else`, `switch`, `while`, `do` e `for` são palavras-chave Java. Uma lista completa das palavras-chave Java é apresentada no Apêndice C.

Resumo das instruções de controle em Java

O Java contém somente três tipos de estruturas de controle, que daqui para a frente chamaremos de instruções de controle: a instrução de sequência, instruções de seleção (três tipos) e instruções de repetição (três tipos). Cada programa é formado combinando a quantidade de instruções apropriada para o algoritmo que o programa implementa. Podemos modelar cada instrução de controle como um diagrama de atividade. Como na Figura 4.1, cada diagrama contém um estado inicial e um estado final que representa um ponto de entrada e um ponto de saída da instrução de controle, respectivamente. As **instruções de controle de entrada única/saída única** facilitam a construção de programas — basta conectarmos o ponto de saída de uma instrução ao ponto de entrada da instrução seguinte. Chamamos isso de **empilhamento de instruções de controle**. Aprenderemos que existe apenas outra maneira de conectar instruções de controle — **aninhamento de instruções de controle** — em que uma instrução de controle aparece dentro da outra. Portanto, algoritmos nos programas Java são construídos a partir de apenas três tipos de instruções de controle, combinadas apenas de duas maneiras. Isso é a essência da simplicidade.

4.5 A instrução de seleção única `if`

Os programas utilizam instruções de seleção para escolher entre cursos alternativos de ações. Por exemplo, suponha que a nota de aprovação de um exame seja 60. A instrução em pseudocódigo:

```
Se a nota do aluno for maior que ou igual a 60
Imprima "Aprovado"
```

determina se a condição "nota do aluno é maior que ou igual a 60" é verdadeira. Nesse caso "Aprovado" é impresso, e a próxima instrução de pseudocódigo é "realizada". (Lembre-se de que o pseudocódigo não é uma linguagem de programação real.) Se a condição for falsa, a instrução *Imprima* é ignorada e a próxima instrução de pseudocódigo na sequência é realizada. O recuo da segunda linha dessa instrução de seleção é opcional, mas recomendável, porque enfatiza a estrutura inerente dos programas estruturados.

A instrução de pseudocódigo *If* precedente pode ser escrita em Java como:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

Observe que o código Java tem uma correspondência precisa com o pseudocódigo. Essa é uma das propriedades do pseudocódigo que torna essa ferramenta de desenvolvimento de programas tão útil.

A Figura 4.2 ilustra a instrução de seleção única `if`. Essa figura contém o símbolo mais importante em um diagrama de atividade — o losango, ou **símbolo de decisão**, que indica que uma decisão deve ser tomada. O fluxo de trabalho continua ao longo de um caminho determinado pelas **condições de guarda** do símbolo associado, que podem ser verdadeiras ou falsas. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda (especificada entre colchetes ao lado da seta). Se uma condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta de transição aponta. Na Figura 4.2, se a nota for maior ou igual a 60, o programa imprime "Aprovado" e então se dirige para o estado final da atividade. Se a nota for menor que 60, o programa se dirige imediatamente para o estado final sem exibir uma mensagem.

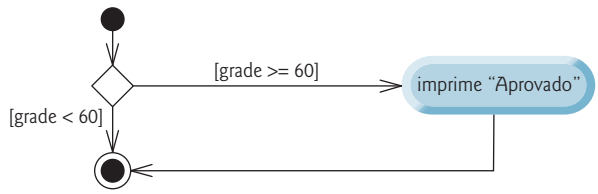


Figura 4.2 | Diagrama de atividades UML de uma instrução de seleção única `if`.

A instrução `if` é uma instrução de controle de uma única entrada e uma única saída. Veremos que os diagramas de atividades para as instruções de controle restantes também contêm estados iniciais, setas de transição, estados de ação que indicam ações a realizar, símbolos de decisão (com condições de guarda associadas) que indicam decisões a serem tomadas e estados finais.

Pense em sete bins, cada um contendo somente um tipo de instrução de controle Java. As instruções de controle estão todas vazias. Sua tarefa é montar um programa com o número necessário de cada tipo de instrução de controle que o algoritmo demanda, combiná-las somente de duas maneiras possíveis (empilhamento ou aninhamento) e, então, preencher os estados e decisões da ação com expressões e condições de guarda apropriadas para o algoritmo. Discutiremos agora as várias maneiras como as ações e decisões podem ser escritas.

4.6 A instrução de seleção dupla `if...else`

A instrução `if` de seleção única realiza uma ação indicada somente quando a condição é `true`; caso contrário, a ação é pulada. A **instrução de seleção dupla `if...else`** permite especificar uma ação a realizar quando a condição é verdadeira e uma ação diferente quando a condição é falsa. Por exemplo, este pseudocódigo:

```
Se a nota do aluno for maior que ou igual a 60
Imprima "Aprovado"
Caso contrário (Else)
Imprima "Reprovado"
```

imprime "Aprovado" se a nota do aluno for maior ou igual a 60, mas imprime "Reprovado" se for menor que 60. Em qualquer um dos casos, depois que impressão ocorre, a próxima instrução do pseudocódigo na sequência é "realizada".

A instrução *If...Else* no pseudocódigo anterior pode ser escrita em Java assim:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

Observe que o corpo do `else` também é recuado. Qualquer que seja a convenção de recuo escolhida, você deve aplicá-la consistentemente por todos os seus programas.

Boa prática de programação 4.1

Recue as duas instruções do corpo de uma instrução `if...else`.

Boa prática de programação 4.2

Se existem vários níveis de recuo, cada nível deve ser recuado pela mesma quantidade adicional de espaço.

A Figura 4.3 ilustra o fluxo de controle na instrução `if...else`. Mais uma vez, os símbolos no diagrama de atividades UML (além do estado inicial, setas de transição e estado final) representam os estados e decisões da ação.

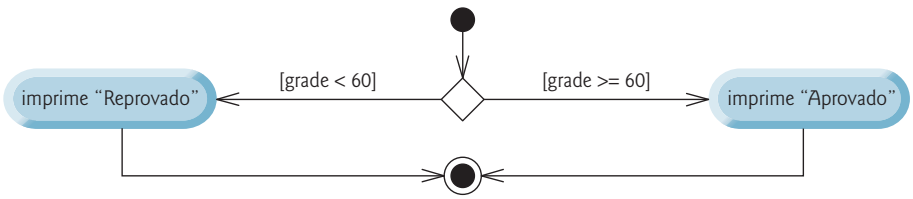


Figura 4.3 | Diagrama de atividades da UML de instrução de seleção dupla `if...else`.

Operador condicional (?:)

O Java fornece o **operador condicional (`?:`)** que pode ser utilizado no lugar de uma instrução `if...else`. Esse é o único **operador ternário** do Java (operador que recebe três operandos). Juntos, os operandos e o símbolo `?:` formam uma expressão condicional. O primeiro operando (à esquerda do `?`) é uma expressão **`boolean`** (isto é, uma condição que é avaliada como um valor `boolean` — **`true`** ou **`false`**), o segundo operando (entre o `?` e `:`) é o valor da expressão condicional se a expressão `boolean` for `true` e o terceiro operando (à direita do `:`) é o valor da expressão condicional se a expressão `boolean` for avaliada como `false`. Por exemplo, a instrução:

```
System.out.printIn ( studentGrade>=60 ? "Passed" : "Failed" );
```

imprime o valor do argumento da expressão condicional de `println`. A expressão condicional nessa instrução é avaliada para a string `"Passed"` se a expressão `boolean studentGrade >= 60` for verdadeira e para a string `"Failed"` se a expressão `boolean` for falsa. Portanto, essa instrução com o operador condicional realiza essencialmente a mesma função da instrução `if...else` mostrada anteriormente nesta seção. A precedência do operador condicional é baixa, então a expressão condicional inteira normalmente é colocada entre parênteses. Veremos que as expressões condicionais podem ser utilizadas em algumas situações nas quais as instruções `if...else` não podem.

Boa prática de programação 4.3

As expressões condicionais são mais difíceis de ler que as instruções `if...else` e devem ser utilizadas para substituir somente instruções `if...else` simples que escolhem entre dois valores.

Instruções `if...else` aninhadas

Um programa pode testar múltiplos casos colocando instruções `if...else` dentro de outras instruções `if...else` para criar **instruções `if...else` aninhadas**. Por exemplo, o pseudocódigo a seguir representa uma `if...else` aninhada que imprime `A` para notas de exame maiores que ou igual a 90, `B` para notas de 80 a 89, `C` para notas de 70 a 79, `D` para notas de 60 a 69 e `F` para todas as outras notas:

```
Se a nota do aluno for maior que ou igual a 90
    Imprima "A"
Caso contrário
    Se a nota do aluno for maior que ou igual a 80
        Imprima "B"
    Caso contrário
        Se a nota do aluno for maior que ou igual a 70
            Imprima "C"
        Caso contrário
            Se a nota do aluno for maior que ou igual a 60
                Imprima "D"
            Caso contrário
                Imprima "F"
```

Esse pseudocódigo pode ser escrito em Java como:

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

Se a variável `studentGrade` for maior ou igual a 90, as quatro primeiras condições na instrução `if...else` aninhada serão verdadeiras, mas somente a instrução na parte `if` da primeira instrução `if...else` será executada. Depois que essa instrução for executada, a parte `else` da instrução `if...else` "mais externa" é pulada. A maioria dos programadores Java prefere escrever a instrução `if...else` anterior deste modo:

```
if ( studentGrade >=  90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

As duas formas são idênticas, exceto pelo espaçamento e recuo, que o compilador ignora. A última forma é popular porque evita grande recuo de código para a direita. Tal entrada muitas vezes deixa pouco espaço em uma linha de código-fonte, forçando a divisão de linhas.

O problema do `else` oscilante

O compilador Java sempre associa um `else` à instrução `if` imediatamente anterior, a menos que instruído de outro modo pela colocação de chaves (`{` e `}`). Esse comportamento pode levar àquilo que é chamado do **problema do `else` oscilante**. Por exemplo:

```
 if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( "x is <= 5" );
```

parece indicar que, se `x` for maior do que `5`, a instrução `if` aninhada determina se `y` também é maior do que `5`. Se for assim, a string `"x and y are > 5"` é enviada para a saída. Caso contrário, parece que se `x` não for maior que `5`, a parte `else` do `if...else` imprime a string `"x is <= 5"`. Cuidado! Essa instrução `if...else` aninhada não é executada como parece. Na verdade, o compilador interpreta a instrução como:

```
 if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
   else
      System.out.println( "x is <= 5" );
```

em que o corpo da primeira `if` é uma `if...else` aninhada. A instrução `if` externa testa se `x` é maior que `5`. Se for, a execução continuará testando se `y` também é maior que `5`. Se a segunda condição for verdadeira, a string adequada — `"x and y are > 5"` — é exibida. Mas, se a segunda condição for falsa, a string `"x is <= 5"` é exibida, apesar de sabermos que `x` é maior que `5`. Igualmente ruim, se a condição da instrução `if` externa for falsa, o `if...else` interno é pulado e nada é exibido.

Para forçar a instrução `if...else` aninhada para executar como foi originalmente concebida, devemos escrevê-la como a seguir:

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
```

As chaves indicam que a segunda instrução `if` está no corpo da primeira e que a instrução `else` está associada com a *primeira* instrução `if`. Os exercícios 4.27–4.28 investigam ainda mais o problema do `else` oscilante.

Blocos

A instrução `if` normalmente espera somente uma instrução no seu corpo. Para incluir várias instruções no corpo de uma `if` (ou no corpo de um `else` de uma instrução `if...else`), inclua as instruções dentro de chaves (`{` e `}`). As instruções contidas em um par de chaves formam um **bloco**. Um bloco pode ser colocado em qualquer lugar em um programa em que uma única instrução pode ser colocada.

O exemplo a seguir inclui um bloco na parte `else` de uma instrução `if...else`:

```
if ( grade >=  60 )
   System.out.println( "Passed" );
else
{
   System.out.println( "Failed" );
   System.out.println( "You must take this course again." );
}
```

Nesse caso, se `grade` é menor que 60, o programa executa ambas as instruções no corpo do `else` e imprime:

```
Failed.
You must take this course again.
```

Observe as chaves que cercam as duas instruções na cláusula `else`. Essas chaves são importantes. Sem as chaves, a instrução:

```
System.out.println( "You must take this course again." );
```

estaria fora do corpo na parte `else` da instrução `if...else` e seria executada independentemente se a nota fosse menor que 60.

Os erros de sintaxe (por exemplo, quando não é colocada uma das chaves em um bloco do programa) são capturados pelo compilador. Um **erro de lógica** (por exemplo, quando não colocadas as duas chaves em um bloco do programa) são capturadas em tempo de execução. Um **erro de lógica fatal** faz com que um programa falhe e finalize prematuramente. Um **erro de lógica não fatal** permite a um programa continuar a executar, mas faz com que produza resultados incorretos.

Erro comum de programação 4.1

Esquecer uma ou ambas as chaves que delimitam uma instrução composta pode levar a erros de sintaxe ou de lógica.

Boa prática de programação 4.4

Sempre utilizar as chaves em uma instrução de controle `if...else` (ou outra) ajuda a evitar uma omissão acidental, especialmente ao adicionar instruções à parte `if` ou à parte `else` mais tarde. Para evitar omitir uma ou as duas chaves, digite as chaves de abertura ou fechamento de blocos antes de digitar as instruções individuais dentro das chaves.

Assim como um bloco pode ser colocado em qualquer lugar em que uma instrução individual pode ser colocada, também é possível ter uma instrução vazia. Lembre-se na Seção 2.8 de que a **instrução vazia** é representada colocando um **ponto-e-vírgula** (`;`) onde uma instrução normalmente estaria.

Erro comum de programação 4.2

Colocar um ponto-e-vírgula depois da condição em uma instrução `if` ou `if...else` resulta em um erro de lógica em instruções `if` de seleção única e um erro de sintaxe em instruções `if...else` de seleção dupla (quando a parte `if` contém uma instrução de corpo real).

4.7 A instrução de repetição `while`

Uma **instrução de repetição** (ou um **loop**) permite especificar que um programa deve repetir uma ação enquanto alguma condição permanece verdadeira. A instrução de pseudocódigo

Enquanto houver mais itens em minha lista de compras
Compre o próximo item e risque-o da minha lista

descreve a repetição que ocorre durante um passeio de compras. A condição "enquanto houver mais itens em minha lista de compras" pode ser verdadeira ou falsa. Se ela for verdadeira, então a ação "Compre o próximo item e risque-o de minha lista" é realizada. Essa ação será realizada repetidamente enquanto a condição permanecer verdadeira. A(s) instrução(ões) contida(s) na instrução de repetição *While* constitui(em) seu corpo, que pode ser uma instrução única ou um bloco. Por fim, a condição se tornaria falsa (quando o último item na

lista de compras foi comprado e riscado da lista). Nesse ponto, a repetição termina e a primeira instrução depois da instrução de repetição é executada.

Como exemplo da **instrução de repetição `while`** do Java, considere um segmento de programa projetado para encontrar a primeira potência de 3 maior que 100. Suponha que a variável `int product` tenha sido inicializada como 3. Quando a instrução `while` seguinte terminar a execução, `product` conterá o resultado:

```
while ( product <= 100 )
   product = 3 * product;
```

Quando essa instrução `while` inicia a execução, o valor da variável `product` é 3. Cada iteração da instrução `while` multiplica `product` por 3, assim `product` assume os valores 9, 27, 81 e 243 sucessivamente. Quando a variável `product` torna-se 243, a condição da instrução `while` — `product <= 100` — torna-se falsa. Isso termina a repetição, portanto o valor final de `product` é 243. Nesse ponto, a execução de programa continua com a próxima instrução depois da instrução `while`.

Erro comum de programação 4.3

Não fornecer, no corpo de uma instrução `while`, uma ação que consequentemente faz com que a condição na `while` torne-se falsa normalmente resulta em um erro de lógica chamado ***loop infinito*** *(o loop nunca termina).*

O diagrama de atividades UML na Figura 4.4 ilustra o fluxo de controle na instrução `while` anterior. Mais uma vez, os símbolos no diagrama (além do estado inicial, setas de transição, um estado final e três notas) representam um estado e uma decisão de ação. Esse diagrama também introduz o **símbolo de agregação**. A UML representa o símbolo de agregação e o símbolo de decisão como losangos. O símbolo de agregação une dois fluxos de atividade a um único. Nesse diagrama, o símbolo de agregação une as transições do estado inicial e do estado de ação, assim ambos fluem para a decisão que determina se o loop deve iniciar (ou continuar) a execução. Os símbolos de decisão e agregação podem ser separados pelo número de setas de transição "entrantes" e "saintes". Um símbolo de decisão contém uma seta de transição apontando para o losango e duas ou mais apontando a partir dele para indicar possíveis transições a partir desse ponto. Além disso, cada seta de transição apontando de um símbolo de decisão contém uma condição de guarda ao lado dela. Um símbolo de agregação tem duas ou mais setas de transição apontando para o losango e somente uma seta saindo do losango, para indicar que diversos fluxos de atividades se juntam a fim de dar continuidade à atividade. Nenhuma das setas de transição associadas com um símbolo de agregação contém uma condição de guarda.

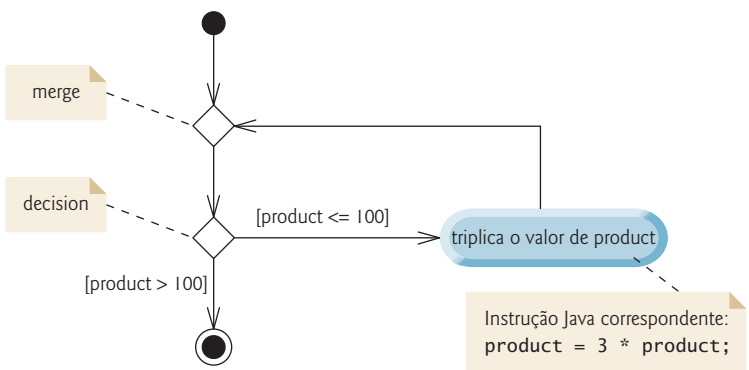


Figura 4.4 | Diagrama de atividades UML da instrução de repetição `while`.

A Figura 4.4 mostra claramente a repetição da instrução `while` discutida anteriormente nesta seção. A seta de transição que advém do estado da ação aponta de volta à agregação, a partir da qual o programa passa o fluxo de volta à decisão que é testada no começo de cada iteração do loop. O loop continua a executar até que a condição de guarda `product > 100` torne-se verdadeira. Em seguida, a instrução `while` termina (alcança seu estado final) e passa o controle para a próxima instrução na sequência do programa.

4.8 Formulando algoritmos: repetição controlada por contador

Para ilustrar como os algoritmos são desenvolvidos, modificaremos a classe `GradeBook` do Capítulo 3 a fim de resolver duas variações de um problema que calcula a média das notas dos alunos. Considere a seguinte declaração do problema:

Uma classe de dez alunos se submeteu a um questionário. As notas (inteiros no intervalo 0 a 100) desse questionário estão disponíveis. Determine a média da classe no questionário.

A média de classe é igual à soma das notas divididas pelo número de alunos. O algoritmo para resolver esse problema em um computador deve inserir cada nota, armazenar o total de todas as notas inseridas, realizar o cálculo da média e imprimir o resultado.

Algoritmo em pseudocódigo com repetição controlada por contador

Vamos usar pseudocódigo para listar as ações a executar e especificar a ordem em que elas devem ser executadas. Utilizamos **repetição controlada por contador** para inserir as notas uma por vez. Essa técnica utiliza uma variável chamada **contador** (ou **variável de controle**) para controlar o número de vezes que um conjunto de instruções será executado. A repetição controlada por contador costuma ser chamada de **repetição definida**, porque o número de repetições é conhecido antes de o loop começar a executar. Nesse exemplo, a repetição termina quando o contador excede 10. Esta seção apresenta um algoritmo de pseudocódigo totalmente desenvolvido (Figura 4.5) e uma versão da classe `GradeBook` (Figura 4.6) que implementa o algoritmo em um método Java. Apresentamos, em seguida, um aplicativo (Figura 4.7) que demonstra o algoritmo em ação. Na Seção 4.9, demonstramos como usar o pseudocódigo para desenvolver esse algoritmo a partir do zero.

Observação de engenharia de software 4.1

A experiência tem mostrado que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Uma vez que um algoritmo correto foi especificado, produzir um programa Java que execute o algoritmo é normalmente simples.

Observe que as referências no algoritmo da Figura 4.5 a um total e a um contador. Um **total** é uma variável utilizada para acumular a soma de vários valores. Um contador é uma variável utilizada para contar — nesse caso, o contador de notas indica qual das 10 notas está em vias de ser inserida pelo usuário. Variáveis utilizadas para armazenar totais normalmente são inicializadas como zero antes de serem utilizadas em um programa.

```
1   Configura o total como zero
2   Configura o contador de notas como um
3
4   Enquanto contador de notas for menor ou igual a dez
5           Solicite para o usuário inserir a próxima nota
6           Insere a próxima nota
7           Adicione a nota ao total
8           Adicione um ao contador de notas
9
10  Configure a média da classe como o total dividido por dez
11  Imprime a média da classe
```

Figura 4.5 | O algoritmo em pseudocódigo com a repetição controlada por contador para resolver o problema da média da classe.

Implementando a repetição controlada por contador na classe GradeBook

A classe `GradeBook` (Figura 4.6) contém um construtor (linhas 11–14) que atribui um valor à variável de instância `courseName` da classe (declarada na linha 8). As linhas 17–20, 23–26 e 29–34 declaram métodos `setCourseName`, `getCourseName` e `displayMessage`, respectivamente. As linhas 37–66 declaram o método `determineClassAverage`, que implementa o algoritmo de média da classe descrito pelo pseudocódigo na Figura 4.5.

```
1   // Figura 4.6: GradeBook.java
2   // Classe GradeBook que resolve o problema da média da classe
3   // utilizando repetição controlada por contador.
4   import java.util.Scanner; // programa utiliza a classe Scanner
5
6   public class GradeBook
7   {
8      private String courseName; // nome do curso que esse GradeBook representa
9
10     // o construtor inicializa courseName
11     public GradeBook( String name )
12     {
13        courseName = name; // inicializa courseName
14     } // fim do construtor
15
16     // método para configurar o nome do curso
17     public void setCourseName( String name )
```

```
18      {
19         courseName = name; // armazena o nome do curso
20      } // fim do método setCourseName
21
22      // método para recuperar o nome do curso
23      public String getCourseName()
24      {
25         return courseName;
26      } // fim do método getCourseName
27
28      // exibe uma mensagem de boas-vindas ao usuário GradeBook
29      public void displayMessage()
30      {
31         // getCourseName obtém o nome do curso
32         System.out.printf( "Welcome to the GradeBook for\n%s!\n\n",
33            getCourseName() );
34      } // fim do método displayMessage
35
36      // determina a média da classe com base em 10 notas inseridas
37      public void determineClassAverage()
38      {
39         // cria Scanner para obter entrada da janela de comando
40         Scanner input = new Scanner( System.in );
41
42         int total; // soma das notas inseridas pelo usuário
43         int gradeCounter; // número da nota a ser inserida a seguir
44         int grade; // valor da nota inserida pelo usuário
45         int average; // média das notas
46
47         // fase de inicialização
48         total = 0; // inicializa o total
49         gradeCounter = 1; // inicializa o contador de loops
50
51         // fase de processamento
52         while (gradeCounter <= 10 ) // faz o loop 10 vezes
53         {
54            System.out.print( "Enter grade: " ); // prompt
55            grade = input.nextInt(); // insere a próxima nota
56            total = total + grade; // adiciona grade a total
57            gradeCounter = gradeCounter + 1; // incrementa o contador por 1
58         } // fim do while
59
60         // fase de término
61         average = total / 10; // divisão de ints produz um int
62
63         // exibe o total e a média das notas
64         System.out.printf( "\nTotal of all 10 grades is %d\n", total );
65         System.out.printf( "Class average is %d\n", average );
66      } // fim do método determineClassAverage
67   } // fim da classe GradeBook
```

Figura 4.6 | Repetição controlada por contador: problema da média da classe.

A linha 40 declara e inicializa a variável `Scanner` `input`, utilizada para ler os valores inseridos pelo usuário. As linhas 42–45 declaram as variáveis locais `total`, `gradeCounter`, `grade` e `average` como do tipo `int`. A variável `grade` armazena a entrada de usuário.

Observe que as declarações (nas linhas 42–45) aparecem no corpo do método `determineClassAverage`. Lembre-se de que as variáveis declaradas no corpo de um método são variáveis locais e podem ser utilizadas apenas da linha de sua declaração até a chave direita de fechamento da declaração de método. A declaração de uma variável local deve aparecer antes da variável ser utilizada nesse método. Uma variável local não pode ser acessada fora do método em que é declarada.

Neste capítulo, a classe `GradeBook` apenas lê e processa um grupo de notas. O cálculo da média é realizado no método `determineClassAverage` utilizando variáveis locais — não preservamos nenhuma informação sobre as notas dos alunos em variáveis de instância de classe.

As atribuições (nas linhas 48–49) inicializam `total` como `0` e `gradeCounter` como `1`. Observe que essas inicializações ocorrem antes de as variáveis serem utilizadas nos cálculos. As variáveis `grade` e `average` (para a entrada de usuário e a média calculada, respectivamente) não precisam ser inicializadas aqui — seus valores serão atribuídos à medida que forem inseridos ou calculados mais tarde no método.

Erro comum de programação 4.4

Utilizar o valor de uma variável local antes de ela ser inicializada resulta em um erro de compilação. Todas as variáveis locais devem ser inicializadas antes de seus valores serem utilizados nas expressões.

Dica de prevenção de erro 4.1

Inicialize cada contador e total, em sua declaração ou em uma instrução de atribuição. Normalmente, os totais são inicializados como 0. Os contadores normalmente são inicializados como 0 ou 1, dependendo de como eles são utilizados (mostraremos exemplos de quando utilizar 0 e quando utilizar 1).

A linha 52 indica que a instrução `while` deve continuar a fazer o loop (também chamado **iteração**) contanto que o valor de `gradeCounter` seja menor ou igual a 10. Enquanto essa condição permanecer verdadeira, a instrução `while` executará repetidamente as instruções entre as chaves que delimitam o corpo da instrução (linhas 54–57).

A linha 54 exibe o prompt `"Enter grade:"`. A linha 55 lê a nota inserida pelo usuário e a atribui à variável `grade`. A linha 56 adiciona então a nova `grade` inserida pelo usuário a `total` e atribui o resultado a `total`, o que substitui seu valor anterior.

A linha 57 adiciona `1` a `gradeCounter` para indicar que o programa processou uma nota e está pronto para inserir a próxima nota fornecida pelo usuário. O incremento da variável `gradeCounter` acaba fazendo com que ela exceda 10 em algum momento. Então o loop termina, porque sua condição (linha 52) torna-se falsa.

Quando o loop termina, a linha 61 realiza o cálculo médio e atribui seu resultado à variável `average`. A linha 64 utiliza o método `printf` da `System.out` para exibir o texto `"Total of all 10 grades is"` seguido pelo valor da variável `total`. Em seguida, a linha 65 utiliza `printf` para exibir o texto `"Class average is "` seguido pelo valor da variável `average`. Depois que alcançar a linha 66, o método `determineClassAverage` retorna o controle ao método de chamada (isto é, `main` em `GradeBookTest` da Figura 4.7).

Classe GradeBookTest

A classe `GradeBookTest` (Figura 4.7) cria um objeto da classe `GradeBook` (Figura 4.6) e demonstra suas capacidades. As linhas 10–11 da Figura 4.7 criam um novo objeto da classe `GradeBook` e o atribuem à variável `myGradeBook`. A `String` na linha 11 é passada para o construtor `GradeBook` (linhas 11–14 da Figura 4.6). A linha 13 chama o método `displayMessage` de `myGradeBook` para exibir uma mensagem "welcome" ao usuário. A linha 14 chama, então, o método `determineClassAverage` de `myGradeBook` para permitir que o usuário insira 10 notas, cuja média o método então calcula e imprime — o método realiza o algoritmo mostrado na Figura 4.5.

```
1  // Figura 4.7: GradeBookTest.java
2  // Cria o objeto da classe GradeBook e invoca seu método
3  // determineClassAverage
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // cria o objeto myGradeBookda classe GradeBook e
9        // passa o nome de cursor para o construtor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // exibe a mensagem welcome
14       myGradeBook.determineClassAverage(); // calcula a média das 10 notas
15    } // fim de main
16 } // fim da classe GradeBookTest
```

```
Welcome to the GradeBook for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Figura 4.7 | A classe `GradeBookTest` cria um objeto da classe `GradeBook` (Figura 4.6) e invoca seu método `determineClassAverage`.

Observações sobre a divisão de inteiros e truncamento

A média calculada pelo método `determineClassAverage` em resposta à chamada de método na linha 14 da Figura 4.7 produz resultados inteiros. A saída do programa indica que a soma dos valores das notas na execução de exemplo é 846, que, quando dividido por 10, deve produzir o número de ponto flutuante 84,6. Entretanto, o resultado do cálculo `total / 10` (linha 61 da Figura 4.6) é o inteiro 84, porque `total` e `10` são ambos inteiros. Dividir dois inteiros resulta em uma **divisão de inteiro** em que qualquer parte fracionária do cálculo é perdida (isto é, **truncada**). Na próxima seção, veremos como obter um resultado de ponto flutuante a partir do cálculo da média.

Erro comum de programação 4.5

Assumir que a divisão de inteiros arredonda (em vez de truncar) pode levar a resultados incorretos. Por exemplo, 7 ÷ 4, que produz 1,75 na aritmética convencional, é truncado para 1 na aritmética de inteiros, em vez de arredondado para 2.

4.9 Formulando algoritmos: repetição controlada por sentinela

Vamos generalizar o problema de média da classe da Seção 4.8. Considere o seguinte problema:

Desenvolva um programa para tirar a média da classe que processe as notas de acordo com um número arbitrário de alunos toda vez que é executado.

No exemplo anterior de média da classe, a declaração do problema especificou o número de alunos, assim o número de notas (10) era conhecido antecipadamente. Nesse exemplo, nenhuma indicação é dada de quantas notas o usuário irá inserir durante a execução do programa. O programa deve processar um número arbitrário de notas. Como podemos determinar quando parar de inserir as notas? Como saber quando calcular e imprimir a média da classe?

Um modo de resolver esse problema é usar um valor especial chamado **valor de sentinela** (também chamado **valor de sinal**, **valor fictício** ou **valor de flag**) para indicar o "fim de entrada de dados". O usuário insere as notas até que todas as notas legítimas tenham sido inseridas. Ele, então, digita o valor de sentinela para indicar que nenhuma outra nota será inserida. A **repetição controlada por sentinela** é frequentemente chamada **repetição indefinida** uma vez que o número de repetições não é conhecido antes de o loop iniciar a execução.

Obviamente, deve-se escolher um valor de sentinela que não possa ser confundido com um valor aceitável de entrada. As notas em um questionário são inteiros não negativos, portanto, para esse problema, –1 é um valor aceitável de sentinela. Portanto, uma execução do programa de média de classe talvez processe um fluxo de entradas como 95, 96, 75, 74, 89 e –1. O programa então computaria e imprimiria a média de classe para as notas 95, 96, 75, 74 e 89; como –1 é o valor de sentinela, ele não deve entrar no cálculo da média).

Erro comum de programação 4.6

Escolher um valor de sentinela que também seja um valor legítimo de dados é um erro de lógica.

Desenvolvendo o algoritmo de pseudocódigo com refinamento passo a passo de cima para baixo: o topo e o primeiro refinamento

Abordamos o programa de média da classe com uma técnica chamada **refinamento passo a passo de cima para baixo**, essencial para o desenvolvimento de programas bem estruturados. Iniciamos com uma representação em pseudocódigo do **topo** — uma única instrução que fornece a função geral do programa:

Determine a média de classe para o questionário

O topo é, de fato, uma representação *completa* de um programa. Infelizmente, o topo raramente fornece detalhes suficientes para escrever um programa em Java. Então, agora iniciamos o processo de refinamento. Dividimos o topo em uma série de tarefas menores e as listamos na ordem em que serão realizadas. Isso resulta no **primeiro refinamento** que se segue:

Inicialize as variáveis
Insira, some e conte as notas do exame
Calcule e imprima a média da classe

Esse refinamento utiliza somente a estrutura de sequência — os passos listados devem ser executados na ordem, um depois do outro.

Observação de engenharia de software 4.2

Cada refinamento, bem como a própria parte superior, é uma especificação completa do algoritmo; somente o nível de detalhe varia.

Observação de engenharia de software 4.3

Muitos programas podem ser divididos logicamente em três fases: uma fase de inicialização que inicializa as variáveis do programa; uma fase de processamento que insere os valores dos dados e ajusta as variáveis do programa de maneira correspondente; e uma fase de término que calcula e insere os resultados finais.

Prosseguindo para o segundo refinamento

A "Observação de engenharia de software" anterior costuma ser tudo o que você precisa para o primeiro refinamento no processo de cima para baixo. Para prosseguir para o próximo nível de refinamento — isto é, o **segundo refinamento** — definimos variáveis específicas. Neste exemplo, precisamos de um total dos números, uma contagem de quantos números foram processados, uma variável para receber o valor de cada nota à medida que é inserida pelo usuário e uma variável para armazenar a média calculada. A instrução de pseudocódigo.

Inicialize as variáveis

pode ser refinada desta maneira:

Inicialize total como zero
Inicialize counter como zero

Somente as variáveis *total* e *counter* precisam ser inicializadas antes de serem utilizadas. As variáveis *average* e *grade* (para a média calculada e a entrada do usuário, respectivamente) não precisam ser inicializadas, uma vez que seus valores serão substituídos à medida que são calculados ou inseridos.

A instrução de pseudocódigo:

Insira, some e conte as notas do exame

requer uma estrutura de repetição (isto é, um loop) que insere sucessivamente cada nota. Não conhecemos antecipadamente quantas notas devem ser processadas, assim utilizaremos a repetição controlada por sentinela. O usuário insere as notas uma por vez. Depois de inserir a última nota, o usuário insere o valor de sentinela. O programa faz um teste para o valor de sentinela depois de cada nota ser inserida e termina o loop quando o usuário insere o valor de sentinela. O segundo refinamento da instrução de pseudocódigo precedente é, então:

Solicite que o usuário insira a primeira nota
Insira a primeira nota (possivelmente a sentinela)
Enquanto o usuário não inserir a sentinela
Adicione essa nota à soma total
Adicione um ao contador de notas
Solicite para o usuário inserir a próxima nota
Insira a próxima nota (possivelmente a sentinela)

No pseudocódigo, não utilizamos chaves em torno das instruções que formam o corpo da estrutura *While*. Nós simplesmente recuamos as instruções abaixo da *While* para mostrar que pertencem ao *While*. Novamente, o pseudocódigo é apenas um auxílio informal ao desenvolvimento de programa.

A instrução de pseudocódigo:

Calcule e imprima a média da classe

pode ser refinada desta maneira:

Se o contador não for igual a zero
Configure a média como o total dividido pelo contador
Imprima a média
Caso contrário
Imprima "Nenhuma nota foi inserida"

Precisamos ter cuidado aqui para testar a possibilidade de divisão por zero — um erro de lógica que, se passar não detectado, resultaria em falha do programa ou produziria saída inválida. O segundo refinamento completo do pseudocódigo para o problema da média da classe é mostrado na Figura 4.8.

Dica de prevenção de erro 4.2

Ao realizar a divisão por uma expressão cujo valor pode ser zero, teste-a e trate-a (p. ex., imprimir uma mensagem de erro) em vez de permitir a ocorrência do erro.

```
1   Inicialize total como zero
2   Inicialize o contador como zero
3
4   Solicita que o usuário insira a primeira nota
5   Insira a primeira nota (possivelmente o sentinela)
6
7   Enquanto o usuário não inserir o sentinela
8           Adicione essa nota à soma total
9           Adicione um ao contador de notas
10          Solicite para o usuário inserir a próxima nota
11          Insira a próxima nota (possivelmente a sentinela)
12
13  Se o contador não for igual a zero
14          Configure a média como o total dividido pelo contador
15          Imprima a média
16  Caso contrário
17          Imprima "Nenhuma nota foi inserida"
```

Figura 4.8 | Algoritmo em pseudocódigo do problema da média da classe com repetição controlada por sentinela.

Nas Figura 4.5 e Figura 4.8, incluímos linhas em branco e recuos no pseudocódigo para torná-lo mais legível. As linhas em branco separam os algoritmos em suas fases e configuram as instruções de controle; o recuo enfatiza os corpos das instruções de controle.

O algoritmo de pseudocódigo na Figura 4.8 resolve o problema da média de classe mais geral. Esse algoritmo foi desenvolvido depois de dois refinamentos. Às vezes, são necessários mais refinamentos.

Observação de engenharia de software 4.4

Termine o processo de refinamento passo a passo de cima para baixo quando tiver especificado o algoritmo de pseudocódigo em detalhes suficientes para você converter o pseudocódigo em Java. Normalmente, implementar o programa Java é então simples e direto.

Observação de engenharia de software 4.5

Alguns programadores não utilizam ferramentas de desenvolvimento de programa como pseudocódigo. Eles acreditam que seu objetivo final é resolver o problema em um computador e que escrever pseudocódigo só retarda a produção das saídas finais. Embora isso talvez funcione para problemas simples e conhecidos, pode levar a erros sérios e atrasos em projetos grandes e complexos.

Implementando a repetição controlada por sentinela na classe GradeBook

A Figura 4.9 mostra a classe Java `GradeBook` contendo o método `determineClassAverage` que implementa o algoritmo de pseudocódigo da Figura 4.8. Embora cada nota seja um número inteiro, o cálculo da média provavelmente produzirá um número com um ponto decimal — em outras palavras, um número real (isto é, ponto flutuante). O tipo `int` não pode representar esse número, portanto, essa classe utiliza o tipo `double` para fazer isso.

```
1   // Figura 4.9: GradeBook.java
2   // Classe GradeBook que resolve o programa da média da classe
3   // utilizando repetição controlada por sentinela.
4   import java.util.Scanner; // programa utiliza a classe Scanner
5
6   public class GradeBook
7   {
8      private String courseName; // nome do curso que essa GradeBook representa
9
10     // construtor inicializa courseName
11     public GradeBook( String name )
12     {
13        courseName = name; // inicializa courseName
14     } // fim do construtor
15
16     // método para configurar o nome do curso
```

```
17      public void setCourseName( String name )
18      {
19         courseName = name; // armazena o nome do curso
20      } // fim do método setCourseName
21
22      // método para recuperar o nome do curso
23      public String getCourseName()
24      {
25         return courseName;
26      } // fim do método getCourseName
27
28      // exibe uma mensagem de boas-vindas para o usuário GradeBook
29      public void displayMessage()
30      {
31         // getCourseName obtém o nome do curso
32         System.out.printf( "Welcome to the GradeBook for\n%s!\n\n",
33            getCourseName() );
34      } // fim do método displayMessage
35
36      // determina a média de um número arbitrário de notas
37      public void determineClassAverage()
38      {
39         // cria Scanner para obter entrada da janela de comando
40         Scanner input = new Scanner( System.in );
41
42         int total; // soma das notas
43         int gradeCounter; // número de notas inseridas
44         int grade; // valor da nota
45         double average; // número com ponto de fração decimal para a média
46
47         // fase de inicialização
48         total = 0; // inicializa o total
49         gradeCounter = 0; // inicializa o contador de loops
50
51         // fase de processamento
52         // solicita entrada e lê a nota do usuário
53         System.out.print( "Enter grade or -1 to quit: " );
54         grade = input.nextInt();
55
56         // faz um loop até ler o valor de sentinela inserido pelo usuário
57         while ( grade != -1 )
58         {
59            total = total + grade; // adiciona grade a total
60            gradeCounter = gradeCounter + 1; // incrementa counter
61
62            // solicita entrada e lê a próxima nota fornecida pelo usuário
63            System.out.print( "Enter grade or -1 to quit: " );
64            grade = input.nextInt();
65         } // fim do while
66
67         // fase de término
68         // se usuário inseriu pelo menos uma nota...
69         if (gradeCounter != 0 )
70         {
71            // calcula a média de todas as notas inseridas
72            average = (double) total / gradeCounter;
73
74            // exibe o total e a média (com 2 dígitos de precisão)
75            System.out.printf( "\nTotal of the %d grades entered is %d\n",
76               gradeCounter, total );
77            System.out.printf( "Class average is %.2f\n", average );
78         } // fim do if
79         else // nenhuma nota foi inserida, então gera mensagem apropriada
80            System.out.println( "No grades were entered" );
81      } // fim do método determineClassAverage
82   } // fim da classe GradeBook
```

Figura 4.9 | Repetição controlada por sentinela: o problema da média da classe.

Nesse exemplo, vemos que as instruções de controle podem ser empilhadas umas sobre as outras (em sequência). A instrução `while` (linhas 57–65) é seguida na sequência por uma instrução `if...else` (linhas 69–80). A maior parte do código nesse programa é idêntica ao código da Figura 4.6, portanto, iremos nos concentrar nos novos conceitos.

A linha 45 declara a variável `double average`, que permite armazenar a média da classe como um número de ponto flutuante. A linha 49 inicializa `gradeCounter` como `0` uma vez que nenhuma nota foi inserida ainda. Lembre-se de que esse programa utiliza a repetição controlada por sentinela para realizar a entrada das notas. Para manter um registro exato do número das notas inseridas, o programa incrementa `gradeCounter` somente quando o usuário insere uma nota válida.

Lógica do programa para repetição controlada por sentinela versus *repetição controlada por contador*

Compare a lógica do programa para repetição controlada por sentinela nesse aplicativo com aquela para a repetição controlada por contador na Figura 4.6. Na repetição controlada por contador, cada iteração da instrução `while` (p. ex., as linhas 52–58 da Figura 4.6) lê um valor do usuário, para o número especificado de iterações. Na repetição controlada por sentinela, o programa lê o primeiro valor (linhas 53–54 da Figura 4.9) antes de alcançar o `while`. Esse valor determina se o fluxo de controle do programa deve entrar no corpo do `while`. Se a condição do `while` for falsa, o usuário inseriu o valor de sentinela, portanto o corpo do `while` não é executado (isto é, nenhuma nota foi inserida). Se, por outro lado, a condição for verdadeira, o corpo inicia a execução e o loop adiciona o valor `grade` a `total` (linha 59). As linhas 63–64 no corpo do loop inserem então o próximo valor a partir do usuário. Em seguida, o controle do programa alcança a chave direita de fechamento (`}`) do corpo do loop na linha 65, assim a execução continua com o teste da condição de `while` (linha 57). A condição utiliza a `grade` mais recente inserida pelo usuário para determinar se o corpo do loop deve executar novamente. Observe que o valor da variável `grade` é sempre a entrada do usuário imediatamente antes de o programa testar a condição `while`. Isso permite que o programa determine se o valor recém-inserido é o valor de sentinela *antes* de o programa processar esse valor (isto é, adiciona-o a `total`). Se o valor de sentinela for inserido, o loop termina e o programa não adiciona –1 a `total`.

Boa prática de programação 4.5

Em um loop controlado por sentinela, os prompts solicitando dados lembram o usuário da sentinela.

Depois de o loop terminar, a instrução `if...else` nas linhas 69–80 é executada. A condição na linha 69 determina se uma nota qualquer foi inserida. Se nenhuma foi inserida, a parte `else` (linhas 79–80) da instrução `if...else` executa e exibe a mensagem `"No grades were entered"` e o método retorna o controle ao método de chamada.

Observe o bloco da instrução `while` na Figura 4.9 (linhas 58–65). Sem as chaves, o loop consideraria o seu corpo como sendo apenas a primeira instrução, o que adiciona `grade` a `total`. As últimas três instruções no bloco iriam cair fora do corpo do loop, fazendo com que o computador interprete o código incorretamente como a seguir:

```
while ( grade != -1 )
   total = total + grade; // adiciona grade a total
gradeCounter = gradeCounter + 1; // incrementa counter

// solicita entrada e lê a próxima nota fornecida pelo usuário
System.out.print( "Enter grade or -1 to quit: " );
grade = input.nextInt();
```

O código anterior resulta em um loop infinito no programa até o usuário inserir o sentinela `-1` na linha 54 (antes da instrução `while`).

Erro comum de programação 4.7

Omitir as chaves que delimitam um bloco pode levar a erros de lógica, como loops infinitos. Para evitar esse problema, alguns programadores incluem o corpo de cada instrução de controle dentro de chaves mesmo se o corpo contiver somente uma única instrução.

Conversão explícita e implícita entre tipos primitivos

Se pelo menos uma das notas foi inserida, a linha 72 da Figura 4.9 calcula a média das notas. Lembre-se na Figura 4.6 de que a divisão de inteiros fornece um resultado inteiro. Mesmo que a variável `average` seja declarada como um `double` (linha 45), o cálculo

```
average = total / gradeCounter;
```

perde a parte fracionária do quociente antes de o resultado da divisão ser atribuído a `average`. Isso ocorre porque `total` e `gradeCounter` são inteiros e a divisão por inteiros fornece um resultado inteiro. Para realizar um cálculo de ponto flutuante com valores inteiros, devemos temporariamente tratar esses valores como números de ponto flutuante para utilização no cálculo. O Java fornece o **operador unário de coerção** para realizar essa tarefa. A linha 72 utiliza o operador de coerção **`(double)`** — um operador unário — para criar uma cópia

temporária de ponto flutuante do seu operando `total` (que aparece à direita do operador). A utilização de um operador de coerção dessa maneira é chamada de **conversão explícita** ou **coerção de tipo**. O valor armazenado em `total` ainda é um inteiro.

O cálculo agora consiste em um valor de ponto flutuante (a versão `double` temporária de `total`) dividida pelo inteiro `gradeCounter`. O Java sabe como avaliar somente expressões aritméticas nas quais os tipos dos operandos são idênticos. Para assegurar que os operandos sejam do mesmo tipo, o Java realiza uma operação chamada **promoção** (ou **conversão implícita**) em operandos selecionados. Por exemplo, em uma expressão que contém valores dos tipos `int` e `double`, os valores de `int` são **promovidos** para valores `double` para uso na expressão. Nesse exemplo, o valor de `gradeCounter` é promovido para o tipo `double`, então a divisão de ponto flutuante é realizada e o resultado do cálculo é atribuído a `average`. Contanto que o operador de coerção `(double)` seja aplicado a qualquer variável no cálculo, o cálculo produzirá um resultado `double`. Mais adiante, neste capítulo, discutiremos todos os tipos primitivos. Você aprenderá mais sobre as regras de promoção na Seção 6.7.

Erro comum de programação 4.8

Um operador de coerção pode ser utilizado para converter entre tipos numéricos e primitivos, como `int` e `double`, e entre tipos por referência relacionados (como discutiremos no Capítulo 10, "Programação orientada a objetos: polimorfismo"). Aplicar uma coerção ao tipo errado pode causar erros de compilação ou erros de tempo de execução.

Os operadores de coerção estão disponíveis para quaisquer tipos. O operador de coerção é formado colocando parênteses em torno do nome de um tipo. O operador é um **operador unário** (isto é, um operador que recebe somente um operando). O Java também suporta versões unárias de operadores de adição (+) e subtração (–), portanto o programador pode escrever expressões como –7 ou +5. Os operadores de coerção são associados da direita para a esquerda e têm a mesma precedência que outros operadores unários como + unário e - unário. Essa precedência é um nível mais alto do que aquela dos **operadores multiplicativos** *, / e %. (Consulte a tabela de precedência de operadores no Apêndice A). Indicamos o operador de coerção com a notação `(tipo)` nas nossas tabelas de precedência para indicar que qualquer nome de tipo pode ser utilizado para formar um operador de coerção.

A linha 77 exibe a média da classe. Nesse exemplo, exibimos a média de classe arredondada para o centésimo mais próximo. O especificador de formato `%.2f` na string de controle de formato de `printf` (linha 77) indica que o valor da variável `average` deve ser exibido com dois dígitos de precisão à direita do ponto de fração decimal — indicado por `.2` no especificador de formato. As três notas inseridas durante a execução do exemplo da classe `GradeBookTest` (Figura 4.10) totalizam 257, o que produz a média 85,666666... O método `printf` utiliza a precisão no especificador de formato para arredondar o valor de acordo com o número especificado de dígitos. Nesse programa, a média é arredondada para a posição dos centésimos e a média é exibida como `85,67`.

```
1  // Figura 4.10: GradeBookTest.java
2  // Cria o objeto da classe GradeBook e invoca seu método determineClassAverage
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // cria o objeto myGradeBook da classe GradeBook e
9        // passa o nome de cursor para o construtor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // exibe a mensagem welcome
14       myGradeBook.determineClassAverage(); // calcula a média
15    } // fim de main
16 } // fim da classe GradeBookTest
```

```
Welcome to the GradeBook for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Figura 4.10 | A classe `GradeBookTest` cria um objeto de classe `GradeBook` (Figura 4.9) e invoca seu método `determineClassAverage`.

4.10 Formulando algoritmos: instruções de controle aninhadas

Para o próximo exemplo, mais uma vez formulamos um algoritmo usando o pseudocódigo e o refinamento passo a passo de cima para baixo e escrevemos um programa Java correspondente. Vimos que as instruções de controle podem ser empilhadas uma sobre as outras (em sequência). Nesse estudo de caso, examinaremos a única outra maneira estruturada como instruções de controle podem ser conectadas — a saber, **aninhando** uma instrução de controle dentro de uma outra.

Considere a seguinte declaração do problema:

> *Uma faculdade oferece um curso que prepara os candidatados a obter licença estadual para corretor de imóveis. No ano passado, dez alunos que concluíram esse curso prestaram o exame. A universidade quer saber como foi o desempenho dos seus alunos nesse exame. Você foi contratado para escrever um programa que resuma os resultados. Para tanto, você recebeu uma lista desses 10 alunos. Ao lado de cada nome é escrito 1 se o aluno passou no exame ou 2 se o aluno foi reprovado.*
>
> *Seu programa deve analisar os resultados do exame da seguinte maneira:*
>
> 1. *Insira cada resultado do exame (isto é, um 1 ou um 2). Exiba a mensagem "Inserir resultado" na tela toda vez que o programa solicitar o resultado de outro exame.*
> 2. *Conte o número de cada tipo de resultado.*
> 3. *Exiba um resumo dos resultados do exame indicando o número de alunos aprovados e reprovados.*
> 4. *Se mais de oito estudantes forem aprovados no exame, imprima a mensagem "Bonus to instructor!"*

Depois de ler a declaração do problema cuidadosamente, fazemos estas observações:

1. O programa deve processar resultados do exame para 10 alunos. Um loop controlado por contador pode ser utilizado, porque o número de resultados do teste é conhecido antecipadamente.
2. Cada resultado do exame tem um valor numérico — 1 ou 2. Toda vez que ler o resultado de um exame, o programa deve determinar se ele é 1 ou 2. Em nosso algoritmo, testamos se o número é 1. Se o número não for 1, suporemos que ele seja 2. (O Exercício 4.24 considera as consequências dessa suposição).
3. Dois contadores são usados para monitorar os resultados do exame — um para contar o número de alunos que foram aprovados e outro para contar o número dos que foram reprovados.
4. Depois que o programa processou todos os resultados, ele deve decidir se mais de oito alunos foram aprovados no exame.

Vamos continuar com o refinamento passo a passo de cima para baixo. Iniciamos com uma representação do pseudocódigo do topo:

```
Analise os resultados do exame e decida se a taxa de matrícula deve ser elevada
```

Mais uma vez, o topo é uma representação *completa* do programa, mas vários refinamentos talvez sejam necessários antes que o pseudocódigo possa ser naturalmente transformado em um programa Java.

Nosso primeiro refinamento é:

```
Inicialize as variáveis
Insira os 10 resultados dos exames e conte as aprovações e reprovações
Imprima um resumo dos resultados do exame e decida se a taxa da matrícula deve ser elevada
```

Aqui, igualmente, mesmo tendo uma representação completa do programa inteiro, é necessário refinamento adicional. Agora empregamos variáveis específicas. Precisamos de contadores para registrar as aprovações e reprovações, de um contador para controlar o processo de loop e de uma variável para armazenar a entrada do usuário. A variável em que a entrada do usuário será armazenada não é inicializada no começo do algoritmo, uma vez que seu valor é lido a partir da entrada fornecida pelo usuário durante cada iteração do loop.

A instrução de pseudocódigo:

```
Inicialize as variáveis
```

pode ser refinada desta maneira:

```
Inicialize as aprovações como zero
Inicialize as reprovações como zero
Inicialize o contador de alunos como um
```

Observe que somente os contadores são inicializados no início do algoritmo.

A instrução de pseudocódigo:

```
Insira os 10 resultados dos exames e conte as aprovações e reprovações
```

requer um loop que sucessivamente insere o resultado de cada exame. Sabemos antecipadamente que há precisamente 10 resultados, portanto um loop controlado por contador é apropriado. Dentro do loop (isto é, aninhada no loop), uma estrutura de seleção dupla determinará se cada resultado é uma aprovação ou uma reprovação e incrementará assim o contador apropriado. O refinamento da instrução de pseudocódigo precedente é então:

```
Enquanto o contador de alunos for menor ou igual a 10
Solicite que o usuário insira o próximo resultado
Insira o próximo resultado de exame
Se o aluno foi aprovado
        Adicione um a aprovações
Caso contrário
        Adicione um a reprovações
Adicione um ao contador de aluno
```

Utilizamos linhas em branco para isolar a estrutura de controle *If...Else*, o que melhora a legibilidade.
A instrução de pseudocódigo:

```
Imprima um resumo dos resultados do exame e decida se a taxa da matrícula deve ser elevada
```

pode ser refinada desta maneira:

```
Imprima o número de aprovações
Imprima o número de reprovações
Se mais de oito alunos forem aprovados
Imprima "Bonus to instructor!"
```

Segundo refinamento completo do pseudocódigo e conversão para a classe `Analysis`

O segundo refinamento completo aparece na Figura 4.11. Note que as linhas em branco também são utilizadas para destacar a estrutura *While* para melhorar a legibilidade de programa. Esse pseudocódigo agora está suficientemente refinado para ser convertido em Java.

```
1  Inicialize as aprovações como zero
2  Inicialize as reprovações como zero
3  Inicialize o contador de alunos como um
4
5  Enquanto o contador de alunos for menor ou igual a 10
6          Solicite que o usuário insira o próximo resultado
7          Insira o próximo resultado do exame
8
9          Se o aluno foi aprovado
10                 Adicione um a aprovações
11         Caso contrário (Else)
12                 Adicione um a reprovações
13
14         Adicione um ao contador de aluno
15
16 Imprima o número de aprovações
17 Imprima o número de reprovações
18
19 Se mais de oito alunos forem aprovados
20         Imprima "Bonus to instructor!"
```

Figura 4.11 | Pseudocódigo para o problema dos resultados do exame.

A classe Java que implementa o algoritmo de pseudocódigo e as duas execuções de exemplo são mostradas na Figura 4.12. As linhas 13–16 de `main` declaram as variáveis que o método `processExamResults` da classe `Analysis` utiliza para processar os resultados do exame. Várias dessas declarações utilizam a capacidade do Java de incorporar a inicialização de variável em declarações (`0` é atribuído a `passes`, `0` é atribuído a `failures` e `1` é atribuído a `studentCounter`). Fazer um loop em programas requer uma inicialização no começo de cada repetição — normalmente realizada por instruções em atribuições em vez de em declarações.

Dica de prevenção de erro 4.3
Inicializar variáveis locais quando são declaradas ajuda-lhe a evitar quaisquer erros de compilação que poderiam surgir de tentativas para utilizar variáveis não inicializadas. Embora o Java não exija que as inicializações das variáveis locais sejam incorporadas a declarações, ele exige que variáveis locais sejam inicializadas antes de seus valores serem utilizados em uma expressão.

A instrução `while` (linhas 19–33) faz um loop 10 vezes. Durante cada iteração, o loop insere e processa um dos resultados do exame. Observe que a instrução `if...else` (linhas 26–29) para processar cada resultado é aninhada na instrução `while`. Se o `result` for `1`, a instrução `if...else` incrementa `passes`; caso contrário, ela supõe que `result` é `2` e incrementa `failures`. A linha 32 incrementa `studentCounter` antes de a condição de loop ser testada novamente na linha 19. Depois que 10 valores foram inseridos, o loop termina e a linha 36 exibe o número de `passes` e o número de `failures`. A instrução `if` nas linhas 39–40 determina se mais que oito alunos foram aprovados no exame e, se foram, gera saída da mensagem `"Bonus to instructor!"`.

```
1  // Figura 4.12: Analysis.java
2  // Análise dos resultados dos exames.
3  import java.util.Scanner; // classe utiliza a classe Scanner
4
5  public class Analysis
6  {
7     public static void main( String[] args )
8     {
9        // cria Scanner para obter entrada a partir da janela de comando
10       Scanner input = new Scanner( System.in );
11
12       // inicializando variáveis nas declarações
13       int passes = 0; // número de aprovações
14       int failures = 0; // número de reprovações
15       int studentCounter = 1; // contador de alunos
16       int result; // um resultado do exame (fornecido pelo usuário)
17
18       // processa 10 alunos com o loop controlado por contador
19       while ( studentCounter <= 10 )
20       {
21          // solicita ao usuário uma entrada e obtém o valor fornecido pelo usuário
22          System.out.print( "Enter result (1 = pass, 2 = fail): " );
23          result = input.nextInt();
24
25          // if...else aninhado em while
26          if ( result == 1 )          // se resultar 1,
27             passes = passes + 1;     // incrementa aprovações;
28          else      // caso contrário, resultado não é 1, então
29             failures = failures + 1; // incrementa reprovações
30
31          // incrementa studentCounter até o loop terminar
32          studentCounter = studentCounter + 1;
33       } // fim do while
34
35       // fase de término; prepara e exibe os resultados
36       System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38       // determina se mais de 8 alunos foram aprovados
38       if ( passes >  8 )
40          System.out.println( "Bonus to instructor!" );
41    } // fim de main
42 } // fim da classe Analysis
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Figura 4.12 | Estruturas de controle aninhadas: problema dos resultados do exame.

A Figura 4.12 mostra a entrada e a saída de duas execuções de exemplo do programa. Durante a primeira execução de exemplo, a condição na linha 39 do método `main` é `true` — mais de oito alunos passaram no exame, portanto, o programa imprime uma mensagem que indica que o instrutor deve receber um bônus.

4.11 Operadores de atribuição composta

Os operadores de atribuição composta abreviam expressões de atribuição. Instruções como:

variável = *variável* *operador* *expressão*;

onde *operador* é um dos operadores binários +, -, *, / ou % (ou outros que discutiremos mais adiante), pode ser escrita na forma:

variável *operador*= *expressão*;

Por exemplo, você pode abreviar a instrução:

```
c = c + 3;
```

com o **operador de atribuição composta de adição**, +=, como:

```
c += 3;
```

O operador += adiciona o valor da expressão na sua direita ao valor da variável na sua esquerda e armazena o resultado na variável no lado esquerdo do operador. Portanto, a expressão de atribuição `c += 3` adiciona 3 a `c`. A Figura 4.13 mostra os operadores aritméticos de atribuição composta, expressões de exemplo que utilizam os operadores e explicações do que os operadores fazem.

Operador de atribuição	Expressão de exemplo	Explicação	Atribuições
Suponha: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
+=	`c += 7`	`c = c + 7`	10 a c
-=	`d -= 4`	`d = d - 4`	1 a d
*=	`e *= 5`	`e = e * 5`	20 a e
/=	`f /= 3`	`f = f / 3`	2 a f
%=	`g %= 9`	`g = g % 9`	3 a g

Figura 4.13 | Operadores aritméticos de atribuição composta.

4.12 Operadores de incremento e decremento

O Java fornece dois operadores unários para adicionar 1 a ou para subtrair 1 do valor de uma variável numérica. Esses são o **operador de incremento** unário, ++, e o **operador de decremento** unário, --, que são resumidos na Figura 4.14. Um programa pode incrementar por 1 o valor de uma variável chamada `c` utilizando o operador de incremento, ++, em vez da expressão `c = c + 1` ou `c += 1`. Um operador de incremento ou de decremento que é colocado antes de uma variável é chamado **operador de pré-incremento** ou **operador de pré-decremento**, respectivamente. Um operador de incremento ou de decremento que é colocado depois de uma variável é chamado **operador de pós-incremento** ou **operador de pós-decremento**, respectivamente.

Operador	Nome do operador	Expressão de exemplo	Explicação
`++`	pré-incremento	`++a`	Incrementa `a` por 1, então utiliza o novo valor de `a` na expressão em que `a` reside.
`++`	pós-decremento	`a++`	Utiliza o valor atual de `a` na expressão em que `a` reside, então incrementa `a` por 1.
`--`	pré-incremento	`--b`	Decrementa `b` por 1, então utiliza o novo valor de `b` na expressão em que `b` reside.
`--`	pós-decremento	`b--`	Utiliza o valor atual de `b` na expressão em que `b` reside, então decrementa `b` por 1.

Figura 4.14 | Operadores de incremento e decremento.

Usar o operador de pré-incremento (ou de pré-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pré-incrementar** (ou **pré-decrementar**) a variável. Pré-incrementar (ou pré-decrementar) uma variável faz com que a variável seja incrementada (ou decrementada) por 1, e o novo valor da variável é usado na expressão em que ela aparece. Usar o operador de pós-incremento (ou pós-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pós-incrementar** (ou **pós-decrementar**) a variável. Isso faz com que o valor atual da variável seja usado na expressão em que ela aparece, e o valor da variável é incrementado (ou decrementado) por 1.

Boa prática de programação 4.6

Diferentemente dos operadores binários, os operadores de incremento e decremento unários devem ser colocados ao lado dos seus operandos, sem espaços no meio.

A Figura 4.15 demonstra a diferença entre as versões de pré-incremento e pós-incremento do operador de incremento ++. O operador de decremento (--) funciona de maneira semelhante. Observe que esse exemplo contém somente uma classe, com o método `main` realizando todo o trabalho da classe. Neste capítulo e no Capítulo 3, vimos exemplos que consistem em duas classes — uma classe contendo os métodos que realizam as tarefas úteis e outra contendo o método `main`, que cria um objeto da outra classe e chama seus métodos. Nesse exemplo, queremos apenas mostrar a mecânica do operador ++, então, utilizamos somente uma declaração de classe que contém o método `main`. Ocasionalmente, quando não fizer sentido tentar criar uma classe reutilizável para demonstrar um conceito simples, utilizaremos um exemplo "mecânico" contido inteiramente dentro do método `main` de uma classe única.

```
1  // Figura 4.15: Increment.java
2  // operadores de pré-incremento e pós-incremento.
3
4  public class Increment
5  {
6     public static void main( String[] args )
7     {
8        int c;
9
10       // demonstra o operador de pós-incremento
11       c = 5; // atribui 5 à variável c
12       System.out.println( c );   // imprime 5
13       System.out.println( c++ ); // imprime 5 e pós-incrementa
14       System.out.println( c );   // imprime 6
15
16       System.out.println(); // pula uma linha
17
18       // demonstra o operador de pré-incremento
19       c = 5; // atribui 5 à variável c
20       System.out.println( c );   // imprime 5
21       System.out.println( ++c ); // pré-incrementa e imprime 6
22       System.out.println( c );   // imprime 6
23    } // fim de main
24 } // fim da classe Increment
```

```
5
5
6

5
6
6
```

Figura 4.15 | Pré-incrementando e pós-incrementando.

A linha 11 inicializa a variável `c` como `5` e a linha 12 gera a saída do valor inicial da `c`. A linha 13 gera a saída do valor da expressão `c++`. Essa expressão pós-incrementa a variável `c`, assim o valor original de `c` (`5`) é enviado para a saída e, então, o valor de `c` é incrementado (para `6`). Portanto, a linha 13 gera a saída do valor inicial de `c` (`5`) novamente. A linha 14 gera a saída do novo valor de `c` (`6`) para provar que o valor da variável foi de fato incrementado na linha 13.

A linha 19 reinicializa o valor da `c` como `5` e a linha 20 envia o valor de `c` para a saída. A linha 21 gera a saída do valor da expressão `++c`. Essa expressão pré-incrementa `c`, assim seu valor é incrementado; então o novo valor (`6`) é enviado para a saída. A linha 22 gera a saída do valor de `c` novamente para mostrar que o valor de `c` ainda é `6` depois que a linha 21 é executada.

Os operadores aritméticos de atribuição composta e os operadores de incremento e decremento podem ser utilizados para simplificar as instruções de um programa. Por exemplo, as três instruções de atribuição na Figura 4.12 (linhas 27, 29 e 32):

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

podem ser escritas mais concisamente com operadores de atribuição composta, como:

```
passes += 1;
failures += 1;
studentCounter += 1;
```

com operadores de pré-incremento, como:

```
++passes;
++failures;
++studentCounter;
```

ou com operadores de pós-incremento, como:

```
passes++;
failures++;
studentCounter++;
```

Ao incrementar ou decrementar uma variável em uma instrução isolada, as formas de pré-incremento ou pós-incremento têm o mesmo efeito, assim como ocorre com as formas de pré-decremento ou pós-decremento. Apenas quando uma variável aparece no contexto de uma expressão maior é que pré-incrementar e pós-incrementar a variável tem efeitos diferentes (e correspondentemente para pré-decrementar e pós-decrementar).

Erro comum de programação 4.9

Tentar utilizar o operador de incremento ou decremento em uma expressão diferente daquela a que um valor pode ser atribuído é um erro de sintaxe. Por exemplo, escrever `++(x + 1)` é um erro de sintaxe porque `(x + 1)` não é uma variável.

A Figura 4.16 mostra a precedência e a associatividade dos operadores que apresentamos. Eles são mostrados de cima para baixo em ordem decrescente de precedência. A segunda coluna descreve a associatividade dos operadores em cada nível de precedência. O operador condicional (`?:`); os operadores unários de incremento (`++`), decremento (`--`), adição (`+`) e de subtração (`-`); os operadores de coerção e os de atribuição `=`, `+=`, `-=`, `*=`, `/=` e `%=` associam da direita para a esquerda. Todos os outros operadores na tabela de precedência de operadores na Figura 4.16 associam da esquerda para a direita. A terceira coluna lista o tipo de cada grupo de operadores.

Operadores	Associatividade	Tipo
`++ --`	da direita para a esquerda	unário pós-fixo
`++ -- + - (` *tipo* `)`	da direita para a esquerda	unário pré-fixo
`* / %`	da esquerda para a direita	multiplicativo
`+ -`	da esquerda para a direita	aditivo
`=`	da esquerda para a direita	relacional
`== !=`	da esquerda para a direita	igualdade
`?:`	da direita para a esquerda	ternário condicional
`= += -= *= /= %=`	da direita para a esquerda	atribuição

Figura 4.16 | Precedência e associatividade dos operadores discutidos até agora.

4.13 Tipos primitivos

A tabela no Apêndice D lista os oito tipos primitivos em Java. Como ocorre com suas linguagens predecessoras, C e C++, o Java requer que todas as variáveis tenham um tipo. Por essa razão, o Java é referido como uma **linguagem fortemente tipada**.

Em C e C++, os programadores frequentemente têm de escrever versões separadas dos programas a fim de que ele suporte diferentes plataformas de computador, uma vez que não há garantia de que tipos primitivos sejam idênticos entre um computador e outro. Por exemplo, um valor `int` em uma máquina talvez seja representado por 16 bits (2 bytes) de memória, e em outra máquina por 32 bits (4 bytes) de memória. No Java, valores `int` são sempre de 32 bits (4 bytes).

Dica de portabilidade 4.1

Os tipos primitivos em Java são portáveis entre todas as plataformas de computador que suportam Java.

Cada tipo no Apêndice D é listado com seu tamanho em bits (há oito bits em um byte) e seu intervalo de valores. Como os projetistas do Java querem assegurar a portabilidade, eles utilizam padrões internacionalmente reconhecidos para os dois formatos de caracteres (Unicode; para informações adicionais, visite `www.unicode.org`) e números de ponto flutuante (IEEE 754; para informações adicionais, visite `grouper.ieee.org/groups/754/`).

Lembre-se na Seção 3.5 de que as variáveis dos tipos primitivos declaradas fora de um método como campos de uma classe recebem automaticamente valores padrão a menos que explicitamente inicializadas. As variáveis de instância dos tipos `char`, `byte`, `short`, `int`, `long`, `float` e `double` recebem o valor `0` por padrão. Atribui-se às variáveis de instância do tipo `boolean` o valor `false` por padrão. As variáveis de instância de tipo por referência são inicializadas por padrão para o valor `null`.

4.14 (Opcional) Estudo de caso de GUI e imagens gráficas: criando desenhos simples

Um recurso atraente do Java é o suporte a gráficos, que permite a programadores aprimorar aplicativos visualmente. Esta seção introduz uma das capacidades gráficas do Java — desenhar linhas. Ela também aborda os princípios básicos da criação de uma janela para exibir um desenho na tela do computador.

Sistema de coordenadas do Java

Para desenhar em Java, você deve primeiro entender o **sistema de coordenadas** do Java (Figura 4.17), um esquema para identificar pontos na tela. Por padrão, o canto superior esquerdo de um componente GUI tem as coordenadas (0, 0). Um par de coordenadas é composto de uma **coordenada x** (a **coordenada horizontal**) e uma **coordenada y** (a **coordenada vertical**). A coordenada *x* é a localização horizontal que se estende da esquerda para a direita. A coordenada y é a localização vertical que se estende de cima para baixo. O **eixo *x*** descreve cada coordenada horizontal e o **eixo *y*** cada coordenada vertical.

As coordenadas indicam onde elementos gráficos devem ser exibidos em uma tela. Unidades coordenadas são medidas em **pixels**. (O termo pixel significa "picture element" [elemento de imagem]). Um pixel é a menor unidade de resolução de tela.

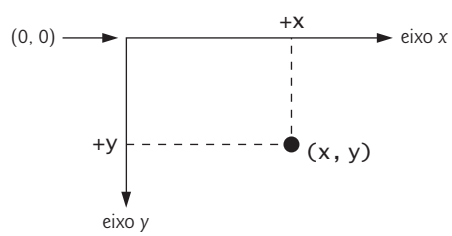


Figura 4.17 | Sistema de coordenadas Java. As unidades são medidas em pixels.

Primeiro aplicativo de desenho

Nosso primeiro aplicativo de desenho simplesmente desenha duas linhas. A classe `DrawPanel` (Figura 4.18) realiza o desenho real, enquanto a classe `DrawPanelTest` (Figura 4.19) cria uma janela para exibir o desenho. Na classe `DrawPanel`, as instruções `import` nas linhas 3–4 permitem utilizar a classe **`Graphics`** (do pacote `java.awt`), que fornece vários métodos para desenhar texto e formas na tela, e a classe **`JPanel`** (do pacote `javax.swing`), que fornece uma área em que podemos desenhar.

```
1  // Figura 4.18: DrawPanel.java
2  // Utilizando DrawLine para conectar os cantos de um painel.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
```

```
6  public class DrawPanel extends JPanel
7  {
8     // desenha um X a partir dos cantos do painel
9     public void paintComponent( Graphics g )
10    {
11       // chama paintComponent para assegurar que o painel seja exibido corretamente
12       super.paintComponent( g );
13
14       int width = getWidth(); // largura total
15       int height = getHeight(); // altura total
16
17       // desenha uma linha a partir do canto superior esquerdo até o inferior direito
18       g.drawLine( 0, 0, width, height );
19
20       // desenha uma linha a partir do canto inferior esquerdo até o superior direito
21       g.drawLine( 0, height, width, 0 );
22    } // fim do método paintComponent
23 } // fim da classe DrawPanel
```

Figura 4.18 | Utilizando `drawLine` para conectar os cantos de um painel.

```
1  // Figura 4.19: DrawPanelTest.java
2  // Aplicativo para exibir uma DrawPanel.
3  import javax.swing.JFrame;
4
5  public class DrawPanelTest
6  {
7     public static void main( String[] args )
8     {
9        // cria um painel que contém nosso desenho
10       DrawPanel panel = new DrawPanel();
11
12       // cria um novo quadro para armazenar o painel
13       JFrame application = new JFrame();
14
15       // configura o frame para ser encerrado quando ele é fechado
16       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
17
18       application.add( panel ); // adiciona o painel ao frame
19       application.setSize( 250, 250 ); // configura o tamanho do frame
20       application.setVisible( true ); // torna o frame visível
21    } // fim de main
22 } // fim da classe DrawPanelTest
```

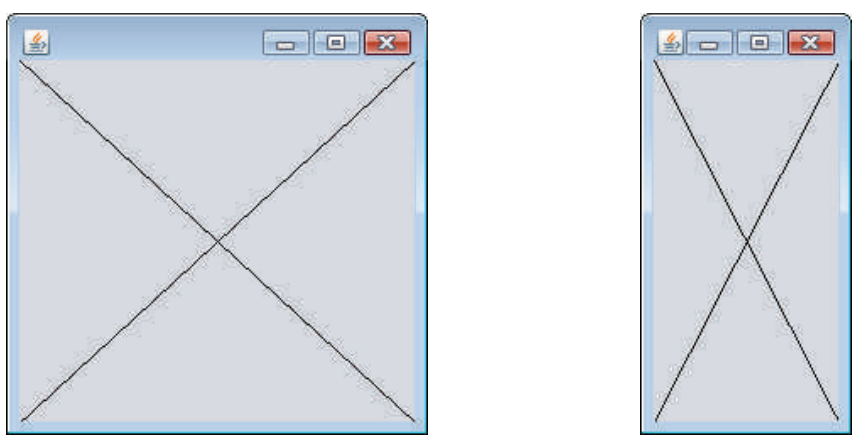


Figura 4.19 | Criando `JFrame` para exibir `DrawPanel`.

A linha 6 utiliza a palavra-chave **extends** para indicar que a classe DrawPanel é um tipo aprimorado de JPanel. A palavra-chave extends representa o relacionamento conhecido como herança, no qual nossa nova classe DrawPanel inicia com os membros existentes (dados e métodos) a partir da classe JPanel. A classe a partir da qual DrawPanel **herda**, JPanel, aparece à direita da palavra-chave extends. Nessa relação de herança, JPanel é chamado de **superclasse** e DrawPanel é chamado de **subclasse**. Isso resulta em uma classe DrawPanel com os atributos (dados) e comportamentos (métodos) da classe JPanel bem como os novos recursos que estamos adicionando à nossa declaração da classe DrawPanel — especificamente, a capacidade de desenhar duas linhas ao longo das diagonais do painel. A herança é explicada detalhadamente no Capítulo 9. Por enquanto, você deve simular nossa classe DrawPanel criando os seus próprios programas gráficos.

Método paintComponent

Todo JPanel, incluindo nosso DrawPanel, contém um método **paintComponent** (linhas 9–22), que o sistema chama automaticamente sempre que precisa exibir o JPanel. O método paintComponent deve ser declarado conforme mostrado na linha 9 — caso contrário, o sistema não o chamará. Esse método é chamado quando um JPanel é exibido na tela pela primeira vez, quando é ocultado e então exibido por uma janela na tela e quando a janela em que aparece é redimensionada. O método paintComponent requer um argumento, um objeto de Graphics, que é oferecido pelo sistema quando ele chama paintComponent.

A primeira instrução em cada método paintComponent que você cria sempre deve ser:

```
super.paintComponent( g );
```

que assegura que o painel seja exibido corretamente antes de começarmos a desenhá-lo. Em seguida, as linhas 14–15 chamam os métodos que a classe DrawPanel herda de JPanel. Como DrawPanel estende JPanel, DrawPanel pode utilizar alguns métodos public de JPanel. Os métodos **getWidth** e **getHeight** retornam a largura e a altura de JPANEL, respectivamente. As linhas 14–15 armazenam esses valores nas variáveis locais width e height. Por fim, as linhas 18–21 utilizam a variável g de Graphics para chamar o método **drawLine** a fim de desenhar as duas linhas. O método drawLine desenha uma linha entre dois pontos representados pelos seus quatro argumentos. Os dois primeiros argumentos são as *coordenadas x* e *y* para uma extremidade, e os dois últimos argumentos são as coordenadas para a outra extremidade. Se você redimensionar a janela, as linhas serão dimensionadas de maneira correspondente, uma vez que os argumentos estão baseados na largura e altura do painel. Redimensionar a janela nesse aplicativo resulta em uma chamada de sistema a paintComponent para redesenhar o conteúdo de DrawPanel.

Classe DrawPanelTest

Para exibir a DrawPanel na tela, você deve colocá-la em uma janela. Você cria uma janela com um objeto da classe **JFrame**. Em DrawPanelTest.java (Figura 4.19), a linha 3 importa a classe JFrame a partir do pacote javax.swing. A linha 10 em main cria um objeto DrawPanel, que contém nosso desenho, e a linha 13 cria um novo JFrame que pode armazenar e exibir o nosso painel. A linha 16 chama o método JFrame **setDefaultCloseOperation** com o argumento **JFrame.EXIT_ON_CLOSE** para indicar que o aplicativo deve terminar quando o usuário fecha a janela. A linha 18 usa o método **add** da classe JFrame para anexar o DrawPanel a JFrame. A linha 19 configura o tamanho da JFrame. O método **setSize** recebe dois parâmetros que representam a largura e a altura do JFrame, respectivamente. Por fim, a linha 20 exibe JFrame chamando seu **método setVisible** com o argumento true. Quando a JFrame é exibida, o método paintComponent de DrawPanel (linhas 9–22 da Figura 4.18) é implicitamente chamado e as duas linhas são desenhadas (veja as saídas de exemplo na Figura 4.19). Tente redimensionar a janela para ver que as linhas sempre são desenhadas com base na largura e na altura atual da janela.

Exercícios do Estudo de caso de GUI e imagens gráficas

4.1 O uso de loops e instruções de controle para desenhar linhas pode resultar em muitos projetos interessantes.

a) Crie o projeto na captura de tela esquerda da Figura 4.20. Esse projeto desenha linhas do canto superior esquerdo, estendendo-as até que cubram a metade superior esquerda do painel. Uma abordagem é dividir a largura e altura em um número igual de passos (descobrimos que 15 passos funcionam bem). A primeira extremidade de uma linha sempre estará no canto superior esquerdo (0, 0). A segunda extremidade pode ser encontrada iniciando no canto inferior esquerdo e movendo-se para cima num passo vertical e movendo-se para direita num passo horizontal. Desenhe uma linha entre as duas extremidades. Continue movendo-se para cima e para o passo à direita para encontrar cada extremidade sucessiva. A figura deve ser dimensionada de maneira correspondente à medida que você redimensiona a janela.

b) Modifique sua resposta na parte (a) para que as linhas se estendam dos quatro cantos, como mostrado na captura de tela direita da Figura 4.20. As linhas de cantos opostos devem se cruzar no meio.

4.2 A Figura 4.21 exibe dois projetos criados com loops while e drawLine.

a) Crie o projeto na captura de tela esquerda da Figura 4.21. Comece dividindo cada borda em um número igual de incrementos (escolhemos 15 novamente). A primeira linha inicia no canto superior esquerdo e termina um passo à direita na extremidade inferior. Para cada linha sucessiva, mova-se para baixo um incremento na borda esquerda e um incremento para a direita na borda inferior. Continue desenhando linhas até alcançar o canto inferior direito. A figura deve ser dimensionada à medida que você redimensiona a janela de modo que as extremidades sempre toquem as bordas.

b) Modifique sua resposta na parte (a) para espelhar o projeto em todos os quatro cantos, como mostrado na tela direita da Figura 4.21.

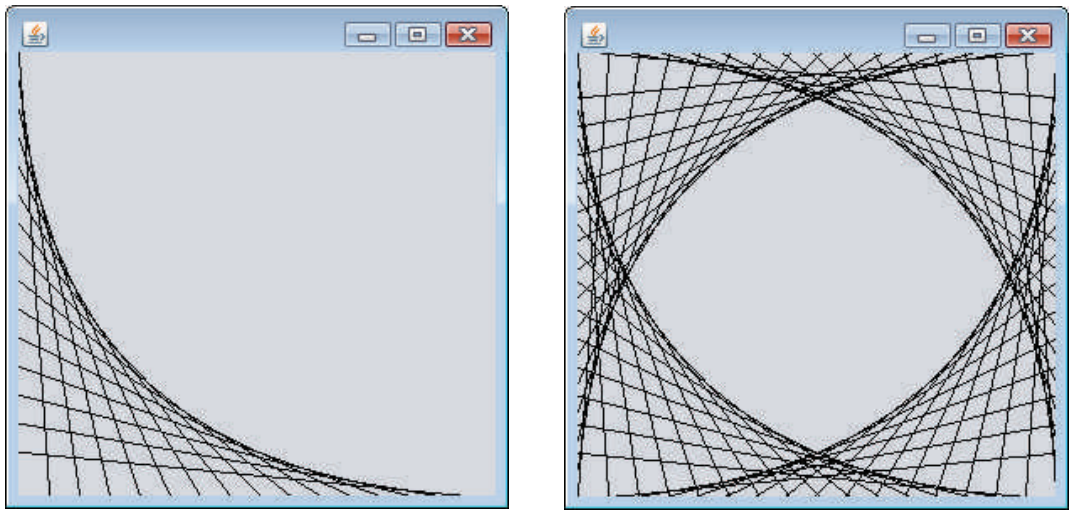


Figura 4.20 | Linhas que se estendem a partir de um canto.

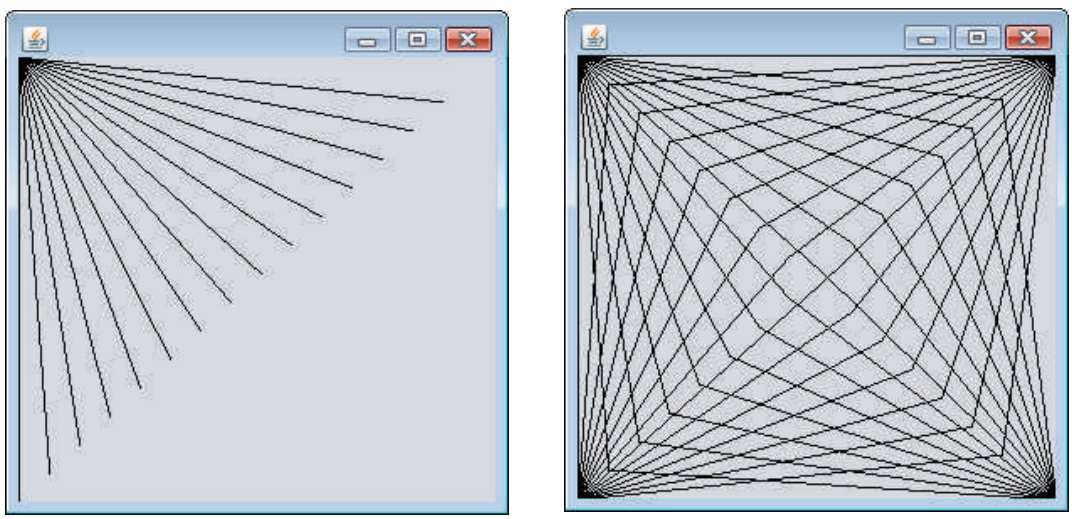


Figura 4.21 | Desenho de linhas com loops e `drawLine`.

4.15 Conclusão

Este capítulo apresentou as estratégias básicas para a solução de problemas que os programadores utilizam na construção de classes e desenvolvimento de métodos para essas classes. Demonstramos como construir um algoritmo (isto é, uma abordagem para resolver um problema) e então como refinar o algoritmo por meio de várias fases de desenvolvimento do pseudocódigo, resultando em um código Java que pode ser executado como parte de um método. O capítulo mostrou como utilizar o refinamento passo a passo de cima para baixo para planejar as ações específicas que um método deve realizar e a ordem em que o método deve realizar essas ações.

Somente três tipos de estruturas de controle — sequência, seleção e repetição — são necessários a fim de desenvolver qualquer algoritmo para solução de problemas. Especificamente, este capítulo demonstrou a instrução de seleção única `if`, a instrução de seleção dupla `if...else` e a instrução de repetição `while`. Essas instruções são alguns blocos de construção utilizados para construir soluções para muitos problemas. Utilizamos o empilhamento de instruções de controle para totalizar e calcular a média de um conjunto de notas de alunos com a repetição controlada por contador e por sentinela e utilizamos o aninhamento de instruções de controle para analisar e tomar decisões com base em um conjunto de resultados de um exame. Apresentamos os operadores de atribuição composta do Java e seus operadores de incremento e decremento. Por fim, discutimos o tipo primitivo disponível para os programadores em Java. No Capítulo 5, continuamos nossa discussão sobre as instruções de controle, introduzindo as instruções `for`, `do...while` e `switch`.

Resumo

Seção 4.1 Introdução

- Antes de escrever um programa para resolver um problema, você deve ter um entendimento completo do problema e uma abordagem cuidadosamente planejada para resolvê-lo. Você também deve entender os blocos de construção que estão disponíveis e empregar técnicas de construção do programa comprovadas.

Seção 4.2 Algoritmos

- Qualquer problema de computação pode ser resolvido executando uma série de ações em uma ordem específica.
- O procedimento para resolver um problema em termos das ações a serem executadas e da ordem em que são executadas é chamado algoritmo.
- Especificar a ordem em que as instruções são executadas em um programa chama-se controle de programa.

Seção 4.3 Pseudocódigo

- O pseucódigo é uma linguagem informal que ajuda os programadores a desenvolver algoritmos sem se preocuparem com os estritos detalhes da sintaxe da linguagem Java.
- O pseudocódigo é similar à língua cotidiana — é conveniente e amigável ao usuário, mas não é uma linguagem de programação de computador real.
- O pseudocódigo ajuda o programador a considerar um programa antes de tentar escrevê-lo em uma linguagem de programação, como Java.
- O pseudocódigo cuidadosamente preparado pode ser facilmente convertido em um programa Java correspondente.

Seção 4.4 Estruturas de controle

- Em geral, instruções em um programa são executadas uma após a outra na ordem em que são escritas. Esse processo é chamado execução sequencial.
- Várias instruções Java permitem ao programador especificar que a próxima instrução a executar não seja necessariamente a próxima na sequência. Isso é chamado transferência de controle.
- Bohm e Jacopini demonstraram que todos os programas poderiam ser escritos em termos de somente três estruturas de controle — a estrutura de sequência, a estrutura de seleção e a estrutura de repetição.
- O termo "estruturas de controle" vem do campo das ciências da computação. A *Java Language Specification* refere-se a "estruturas de controle" (*control structures*) como "instruções de controle" (*control statements*).
- A estrutura de sequência está incorporada ao Java. A não ser que seja instruído de outra forma, o computador executa as instruções Java uma depois da outra na ordem em que elas são escritas, isto é, em sequência.
- Em qualquer lugar que uma ação única pode ser colocada, várias ações podem ser colocadas em sequência.
- Os diagramas de atividades são parte da UML. Um diagrama de atividades modela o fluxo de trabalho (também chamado atividade) de uma parte de um sistema de software.
- Os diagramas de atividade são compostos de símbolos — como símbolos de estado de ação, losangos e pequenos círculos — que são conectados por setas de transição, que representam o fluxo da atividade.
- Os estados de ação contêm expressões de ação que especificam determinadas ações a ser realizadas.
- As setas em um diagrama de atividades representam transições, que indicam a ordem em que ocorrem as ações representadas pelos estados da ação.
- O círculo sólido localizado na parte superior de um diagrama de atividade representa o estado inicial da atividade — o começo do fluxo de trabalho antes do programa realizar as ações modeladas.
- O círculo sólido cercado por um círculo vazio que aparece na parte inferior do diagrama representa o estado final — o fim do fluxo de trabalho depois que o programa realiza suas ações.
- Os retângulos com o canto superior direito dobrado são notas de UML — observações explicativas que descrevem o objetivo dos símbolos no diagrama.
- O Java tem três tipos de instrução de seleção.
- A instrução de seleção única `if` seleciona ou ignora uma única ação ou um único grupo de ações.
- A instrução de seleção dupla `if...else` seleciona entre duas ações ou grupos de ações.
- A instrução `switch` é chamada de instrução de seleção múltipla uma vez que seleciona entre muitas ações diferentes (ou grupos de ações).
- O Java fornece as instruções de repetição `while`, `do...while` e `for` (loop) que permitem que programas executem instruções repetidamente enquanto uma condição de continuação de loop permanece verdadeira.
- As instruções `while` e `for` realizam a(s) ação(ões) no seu corpo zero ou mais vezes — se a condição de continuação do loop for inicialmente falsa, a(s) ação(ões) não será(ão) executada(s). A instrução `do...while` realiza a(s) ação(ões) no seu corpo uma ou várias vezes.
- As palavras `if`, `else`, `switch`, `while`, `do` e `for` são palavras-chave Java. As palavras-chave não podem ser utilizadas como identificadores, por exemplo, nos nomes de variáveis.

- Cada programa é formado combinando o número de instruções de sequência, seleção e de repetição conforme apropriado no algoritmo que o programa implementa.
- As instruções de controle de entrada única/saída única são anexadas uma a outra conectando o ponto de saída de uma instrução ao ponto de entrada da instrução seguinte. Isso é conhecido como empilhamento de instruções de controle.
- Uma instrução de controle também pode ser aninhada dentro de outra instrução de controle.

Seção 4.5 A instrução de seleção única `if`

- Os programas utilizam instruções de seleção para escolher entre cursos alternativos de ações.
- O diagrama de atividade da instrução de seleção única `if` contém o símbolo de losango, que indica que uma decisão deve ser tomada. O fluxo de trabalho segue um caminho determinado pelas condições de guarda associadas do símbolo. Se uma condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta de transição correspondente aponta.
- A instrução `if` é uma instrução de controle de uma única entrada e uma única saída.

Seção 4.6 As instrução de seleção dupla `if...else`

- A instrução `if` de seleção única só realiza uma ação indicada quando a condição for `true`.
- A instrução de seleção dupla `if...else` realiza uma ação quando a condição for verdadeira e uma ação diferente quando a condição for falsa.
- O operador condicional `(?:)` é o único operador ternário do Java — ele leva três operandos. Juntos, os operandos e o símbolo `?:` formam uma expressão condicional.
- Um programa pode testar múltiplos casos com instruções `if...else` aninhadas.
- O compilador Java sempre associa um `else` com o `if` imediatamente precedente a menos que seja instruído a fazer de outro modo pela colocação de chaves (`{}`).
- O enunciado `if` espera uma instrução no seu corpo. Para incluir várias instruções no corpo de uma `if` (ou no corpo de um `else` de uma instrução `if...else`), inclua as instruções dentro de chaves (`{` e `}`).
- Um bloco de instruções pode ser colocado em qualquer lugar em um programa em que uma única instrução pode ser colocada.
- Um erro de lógica tem seu efeito no tempo de execução. Um erro de lógica fatal faz com que um programa falhe e finalize prematuramente. Um erro de lógica não fatal permite a um programa continuar a executar, mas faz com que o programa produza resultados incorretos.
- Assim como um bloco pode ser colocado em qualquer lugar em que uma instrução única pode ser colocada, você também pode utilizar uma instrução vazia, representada colocando-se um ponto-e-vírgula (`;`) onde normalmente entraria uma instrução.

Seção 4.7 A instrução de repetição `while`

- A instrução de repetição `while` permite ao programador especificar que um programa deve repetir uma ação enquanto alguma condição permanecer verdadeira.
- O símbolo de agregação da UML une dois fluxos de atividade em um único fluxo.
- Os símbolos de decisão e agregação podem ser separados pelo número de setas de transição "entrantes" e "saintes". Um símbolo de decisão contém uma seta de transição apontando para o losango e duas ou mais setas de transição apontando a partir do losango para indicar possíveis transições a partir desse ponto. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda. Um símbolo de agregação contém duas ou mais setas de transição apontando para o losango e somente uma seta de transição apontando a partir do losango, para indicar a conexão de múltiplos fluxos de atividades a fim de continuar com ela. Nenhuma das setas de transição associadas com um símbolo de agregação contém uma condição de guarda.

Seção 4.8 Formulando algoritmos: repetição controlada por contador

- A repetição controlada por contador utiliza uma variável chamada contador (ou variável de controle) para controlar o número de vezes que um conjunto de instruções é executado.
- A repetição controlada por contador costuma ser chamada de repetição definida, porque o número de repetições é conhecido antes de o loop começar a executar.
- Um total é uma variável utilizada para acumular a soma de vários valores. Variáveis utilizadas para armazenar totais normalmente são inicializadas como zero antes de serem utilizadas em um programa.
- A declaração de uma variável local deve aparecer antes de a variável ser utilizada nesse método. Uma variável local não pode ser acessada fora do método em que é declarada.
- Quando a divisão de um inteiro por outro resulta num número fracionário, a parte fracionária do cálculo é truncada.

Seção 4.9 Formulando algoritmos: repetição controlada por sentinela

- Na repetição controlada por sentinela, um valor especial chamado de valor de sentinela (também chamado de valor de sinal, valor fictício ou valor de flag) é utilizado para indicar o "fim da entrada de dados".
- Deve-se escolher um valor de sentinela que não possa ser confundido com um valor aceitável de entrada.
- O refinamento passo a passo de cima para baixo é essencial para o desenvolvimento de programas bem estruturados.

- A divisão por zero é um erro de lógica.
- Para realizar um cálculo de ponto flutuante com valores inteiros, faça a coerção de um dos números inteiros para o tipo `double`.
- O Java sabe como avaliar somente expressões aritméticas nas quais os tipos dos operandos são idênticos. Para assegurar isso, o Java realiza uma operação chamada de promoção em operandos selecionados.
- O operador de coerção unário é formado colocando-se parênteses em torno do nome de um tipo.

Seção 4.11 Operadores de atribuição composta

- Os operadores de atribuição composta abreviam expressões de atribuição. Instruções da forma:

  variável = *variável* *operador* *expressão*;

  em que o *operador* é um dos operadores binários `+`, `-` `*`, `/` ou `%`, podem ser escritas na forma:

  variável *operador*= *expressão*;
- O operador `+=` adiciona o valor da expressão à direita do operador ao valor da variável à esquerda do operador e armazena o resultado na variável à esquerda do operador.

Seção 4.12 Operadores de incremento e decremento

- O operador de incremento unário, `++`, e o operador de decremento unário, `--`, adicionam 1 ao, ou subtraem 1 do, valor de uma variável numérica.
- O operador de decremento ou incremento que é prefixado a uma variável é o operador de incremento de prefixo ou decremento de prefixo, respectivamente. O operador de incremento ou decremento que é pós-fixado a uma variável é o operador de pós-incremento ou pós-decremento, respectivamente.
- Utilizar o operador de pré-incremento ou pré-decremento para adicionar ou subtrair 1 é conhecido como preincrementar ou predecrementar, respectivamente.
- Pré-incrementar (ou pré-decrementar) uma variável faz com que a variável seja incrementada ou decrementada por 1; então o novo valor da variável é utilizado na expressão em que ela aparece.
- Utilizar o operador de pós-incremento ou pós-decremento para adicionar ou subtrair 1 é conhecido como pós-incrementar ou pós-decrementar, respectivamente.
- Pós-incrementar ou pós-decrementar a variável faz com que seu valor seja utilizado na expressão em que ele aparece; então o valor da variável é incrementado ou decrementado por 1.
- Ao incrementar ou decrementar uma variável em uma instrução isolada, o pré-incremento ou o pós-incremento têm o mesmo efeito, assim como o pré-decremento ou o pós-decremento.

Seção 4.13 Tipos primitivos

- O Java requer que todas as variáveis tenham um tipo. Assim, o Java é conhecido como uma linguagem fortemente tipada.
- O Java usa caracteres de Unicode e números de ponto flutuante IEEE 754.

Terminologia

`?:`, operador condicional ternário, 86
`+=`, operador de atribuição de adição, 102
`++`, pré-incremento/pós-incremento, 102
`--`, pré-decremento/pós-decremento, 102
ação a executar, 82
adição, operador de atribuição composta, `+=`, 102
agregação, símbolo (na UML), 89
algoritmo, 82
atividade (na UML), 83
`boolean`, tipo primitivo, 86
círculo sólido (na UML), 84
círculo sólido cercado por um círculo vazio (na UML), 84
coerção de tipo, 98
condição de guarda (na UML), 85
contador, 90
controle de programa, 82
conversão explícita, 98
conversão implícita, 98
coordenada horizontal, 105
coordenada vertical, 105
decremento, operador, 102
`(double)`, coerção, 98
`drawLine`, método da classe `Graphics`, 107
`else` oscilante, problema, 87
erro de lógica, 88
erro de lógica fatal, 88
erro de lógica não fatal, 88
erro fatal, 88
estado da ação (na UML), 83
estado final (na UML), 84
estado inicial (na UML), 84
execução sequencial, 83
`EXIT_ON_CLOSE`, constante da classe `JFrame`, 107
expressão condicional, 86
expressão de ação (na UML), 84
`extends`, palavra-chave, 107
fluxo de trabalho, 83
`getHeight`, método da classe `JPanel`, 107
`getWidth`, método da classe `JPanel`, 107
`goto`, instrução, 83
`Graphics`, classe, 105
herdar, 107
`if...else`, instrução de seleção dupla, 85
`if...else` aninhado, instrução de seleção, 86
instrução de controle, 82
instrução vazia (um ponto-e-vírgula, `;`), 88
iteração, 92
`JFrame`, classe, 107
`JFrame`, classe `EXIT_ON_CLOSE`, constante, 107
`JPanel`, classe, 105

linguagens fortemente tipadas, 105
linha pontilhada (na UML), 84
loop infinito, 89
losango (na UML), 83
nota (na UML), 84
operador de decremento, --, 102
operadores aritméticos de atribuição composta, 102
operadores de atribuição, 102
operadores de atribuição composta, 102
operadores multiplicativos, *, / e %, 98
operador ternário, 86
operador unário, 97
ordem em que as ações devem ser executadas, 82
paintComponent, método da classe JComponent, 107
pequenos círculos (na UML), 83
pixel (*picture element*), 105
pós-decrementar, 103
pós-decremento, operador, 102
pós-incrementar, 103
pós-incremento, operador, 102
pré-decrementar, 103
pré-decremento, operador, 102
pré-incrementar, 103
programação estruturada, 83
promoção, 98
pseudocódigo, 82
repetição, instrução de, 83, 84, 88
repetição controlada por sentinela, 93
repetição definida, 90
repetição indefinida, 93
seleção, instrução, 83, 84
seleção dupla, instrução, 84
seleção múltipla, instrução, 84
seleção única, instrução, 84
sequência, estrutura, 83
setDefaultCloseOperation, método da classe JFrame, 107
setSize, método da classe JFrame, 107
setVisible, método da classe JFrame, 107
símbolo de estado de ação, 83
sistema de coordenadas, 105
subclasse, 107
superclasse, 107
topo, 93
total, 90
transferência de controle, 83
transição (na UML), 83
true, palavra reservada, 86
truncar parte fracionária de um cálculo, 93
valor de flag, 93
valor de sentinela, 93
valor de sinal, 93
valor fictício, 93
variável de controle, 90
while, instrução de repetição, 88
X, eixo, 105
x, coordenada, 105
Y, eixo, 105
y, coordenada, 105

Exercícios de autorrevisão

4.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Todos os programas podem ser escritos em termos de três tipos de estruturas de controle: _____, _____ e _____.

b) A instrução _____ é utilizada para executar uma ação quando uma condição for verdadeira e outra quando for falsa.

c) Repetir um conjunto de instruções por um número específico de vezes é chamado de repetição _____.

d) Quando não se sabe antecipadamente quantas vezes um conjunto de instruções será repetido, um valor de _____ pode ser utilizado para terminar a repetição.

e) A estrutura _____ é construída em Java — por padrão, instruções são executadas na ordem em que elas aparecem.

f) As variáveis de instância dos tipos `char`, `byte`, `short`, `int`, `long`, `float` e `double` recebem o valor _____ por padrão.

g) O Java é uma linguagem _____ — ela exige que todas as variáveis tenham um tipo.

h) Se o operador de incremento for _____ para uma variável, primeiro ela é incrementada por 1 e, então, seu novo valor é usado na expressão.

4.2 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Um algoritmo é um procedimento para resolver um problema em termos das ações a ser executadas e da ordem em que essas ações são executadas.

b) Um conjunto de instruções dentro de um par de parênteses é chamado bloco.

c) Uma instrução de seleção especifica que uma ação deve ser repetida enquanto algumas condições permanecem verdadeiras.

d) Uma instrução de controle aninhada aparece no corpo de uma outra instrução de controle.

e) O Java fornece os operadores aritméticos de atribuição composta +=, -=, *=, /= e %= para abreviar expressões de atribuição.

f) Os tipos primitivos (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`) são portáveis somente em plataformas Windows.

g) Especificar a ordem em que as instruções (ações) são executadas em um programa é chamado controle de programa.

h) O operador de coerção unário `(double)` cria uma cópia temporária do tipo inteiro do seu operando.

i) Atribui-se às variáveis de instância do tipo `boolean` o valor `true` por padrão.

j) O pseudocódigo ajuda um programador a pensar sobre um programa antes de tentar escrevê-lo em uma linguagem de programação.

4.3 Escreva quatro instruções Java diferentes que adicionam 1 à variável de inteiro `x`.

4.4 Escreva instruções Java para realizar cada uma das seguintes tarefas:

a) Utilize uma instrução para atribuir a soma de `x` e `y` a `z`, em seguida, incremente `x` por 1.

b) Teste se a variável `contador` é maior do que 10. Se for, imprima `"Contador é maior que 10"`.

c) Use uma instrução para decrementar a variável `x` por 1, subtraia-o da variável `total` e armazene o resultado na variável `total`.

d) Calcule o resto após `q` ser dividido pelo `divisor` e atribua o resultado a `q`. Escreva essa instrução de duas maneiras diferentes.

4.5 Escreva uma instrução Java para realizar cada uma destas tarefas:

a) Declare variáveis `sum` e `x` que serão de tipo `int`.

b) Atribua `1` à variável `x`.

c) Atribua 0 à variável sum.

d) Adicione a variável x à variável sum, e atribua o resultado à variável sum.

e) Imprima "A soma é: " seguido pelo valor da variável sum.

4.6 Combine as instruções escritas no Exercício 4.5 em um aplicativo Java que calcule e imprima a soma dos inteiros de 1 a 10. Utilize a instrução while para fazer loop pelas instruções de cálculo e incremento. O loop deve terminar quando o valor de x torna-se 11.

4.7 Determine o valor das variáveis na instrução `product *= x++;` depois que o cálculo é realizado. Suponha que todas as variáveis sejam do tipo int e inicialmente tenham o valor 5.

4.8 Identifique e corrija os erros em cada um dos seguintes conjuntos de código:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```

b)
```
if ( gender == 1 )
   System.out.println( "Woman" );
else;
   System.out.println( "Man" );
```

4.9 O que há de errado com a instrução while a seguir?
```
while ( z >= 0 ) = 0 )
   sum += z;
```

Respostas dos exercícios de autorrevisão

4.1 a) sequência, seleção, repetição. b) if...else. c) controlada por contador (ou definida). d) sentinela, sinal, flag ou fictício. e) sequência. f) 0 (zero). g) fortemente tipada. h) prefixada.

4.2 a) Verdadeiro. b) Falso. Um conjunto de instruções dentro de um par de chaves ({ e }) é chamado bloco. c) Falso. Uma instrução de repetição especifica que uma ação deve ser repetida enquanto alguma condição permanecer verdadeira. d) Verdadeiro. e) Verdadeiro. f) Falso. Os tipos primitivos (boolean, char, byte, short, int, long,float e double) são portáveis em todas as plataformas de computador que suportam o Java. g) Verdadeiro. h) Falso. O operador de coerção unário (double) cria uma cópia temporária de ponto flutuante do seu operando. i) Falso. Variáveis de instância do tipo boolean recebem o valor false por padrão. j) Verdadeiro.

4.3
```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a)
```
z = x++ + y;
```

b)
```
if ( count > 10 )
   System.out.println( "Count is greater than 10" );
```

c)
```
total -= --x;
```

d)
```
q %= divisor;
q = q % divisor;
```

4.5 a)
```
int sum;
int x;
```

b)
```
x = 1;
```

c)
```
sum = 0;
```

d)
```
sum += x; ou sum = sum + x;
```

e)
```
System.out.printf( "The sum is: %d\n", sum );
```

4.6 O programa é o seguinte:

```
1  // Exercícios 4.6: Calculate.java
2  // Calcula a soma dos inteiros de 1 a 10
3  public class Calculate
4  {
5     public static void main( String[] args )
6     {
7        int sum;
8        int x;
9
10       x = 1;   // inicializa x como 1 para contagem
11       sum = 0; // inicializa a soma como 0 para totalização
12
13       while ( x <= 10 ) // enquanto x é menor ou igual a 10
```

```
14          {
15             sum += x; // adiciona x a soma
16             ++x; // incrementa x
17          } // fim do while
18
19          System.out.printf( "The sum is: %d\n", sum );
20       } // fim de main
21    } // fim da classe Calculate
```

```
The sum is: 55
```

4.7 `product` = 25, `x` = 6

4.8 a) Erro: Está faltando a chave direita de fechamento do corpo da instrução `while`.
Correção: Adicione uma chave direita de fechamento depois da instrução `++c;`.

b) Erro: O ponto-e-vírgula depois de `else` resulta em um erro de lógica. A segunda instrução de saída sempre será executada.
Correção: Remover ponto-e-vírgula depois de `else`.

4.9 O valor da variável z nunca muda na instrução while. Portanto, se a condição de continuação do loop (`z >= 0`) for verdadeira, um loop infinito é criado. Para evitar que um loop infinito ocorra, `z` deve ser decrementado até se tornar menor que 0.

Exercícios

4.10 Compare e contraste a instrução de seleção única `if` e a instrução de repetição `while`. Qual é a semelhança entre essas duas instruções? Qual a diferença?

4.11 Explique o que acontece quando um programa Java tenta dividir um inteiro por um outro. O que acontece para a parte fracionária do cálculo? Como um programador pode evitar esse resultado?

4.12 Descreva as duas maneiras como as instruções de controle podem ser combinadas.

4.13 Que tipo de repetição seria apropriada para o cálculo da soma dos primeiros 100 inteiros positivos? Que tipo de repetição seria apropriada para calcular a soma de um número arbitrário de inteiros positivos? Descreva brevemente como cada uma dessas tarefas poderia ser realizada.

4.14 Qual é a diferença entre pré-incrementar e pós-incrementar uma variável?

4.15 Identifique e corrija os erros em cada um dos seguintes fragmentos de código. [*Nota*: pode haver mais de um erro em cada trecho de código.]

a)
```
if ( age >= 65 );
   System.out.println( "Age is greater than or equal to 65" );
else
   System.out.println( "Age is less than 65 )";
```

b)
```
int x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```

c)
```
while ( x <= 100 )
            total += x;
            ++x;
```

d)
```
while ( y > 0 )
{
   System.out.println( y );
   ++y;
```

4.16 O que o seguinte programa imprime?

```
1  // Exercícios 4.16: Mystery.java
2  public class Mystery
3  {
4     public static void main( String[] args )
5     {
6        int y;
7        int x = 1;
8        int total = 0;
9
10       while ( x <= 10 )
11       {
12          y = x * x;
13          System.out.println( y );
14          total += y;
```

```
15            ++x;
16         } // fim do while
17
18         System.out.printf( "Total is %d\n", total );
19      } // end main
20   } // fim da classe Mystery
```

Para os exercícios 4.17–4.20, siga estes passos:

a) Leia a declaração do problema.

b) Formule o algoritmo utilizando pseudocódigo e refinamento passo a passo de cima para baixo (top-down stepwise).

c) Escreva um programa Java.

d) Teste, depure e execute o programa Java.

e) Processe três conjuntos completos de dados.

4.17 **(*Quilometragem de combustível*)** Os motoristas se preocupam com a quilometragem obtida por seus automóveis. Um motorista monitorou vários tanques cheios de gasolina registrando a quilometragem dirigida e a quantidade de combustível em litros utilizados para cada tanque cheio. Desenvolva um aplicativo Java que receba como entrada os quilômetros dirigidos e os litros de gasolina consumidos (ambos como inteiros) para cada tanque cheio. O programa deve calcular e exibir o consumo em quilômetros/litro para cada tanque cheio e imprimir a quilometragem combinada e a soma total de litros de combustível consumidos até esse ponto. Todos os cálculos de média devem produzir resultados de ponto flutuante. Utilize classe `Scanner` e repetição controlada por sentinela para obter os dados do usuário.

4.18 **(*Calculador de limite de crédito*)** Desenvolva um aplicativo Java que determina se um cliente de uma loja de departamentos excedeu o limite de crédito em uma conta-corrente. Para cada cliente, os seguintes fatos estão disponíveis:

a) número de conta.

b) saldo no início do mês.

c) total de todos os itens cobrados desse cliente no mês.

d) total de créditos aplicados ao cliente no mês.

e) limite de crédito autorizado.

O programa deve inserir todos esses fatos como inteiros, calcular o novo saldo (= *saldo inicial* + *despesas* – *créditos*), exibir o novo saldo e determinar se o novo saldo excede ao limite de crédito do cliente. Para aqueles clientes cujo limite de crédito foi excedido, o programa deve exibir a mensagem `"Limite de crédito excedido"`.

4.19 **(*Calculador de comissão de vendas*)** Uma grande empresa paga seu pessoal de vendas por comissão. O pessoal de vendas recebe R$ 200 por semana mais 9% de suas vendas brutas durante essa semana. Por exemplo, um vendedor que realiza um total de vendas de mercadorias de R$ 5.000 em uma semana recebe R$ 200 mais 9% de R$ 5.000, ou um total de R$ 650. Foi-lhe fornecida uma lista dos itens vendidos por cada vendedor. Os valores desses itens são como segue:

```
Item     Value
1        239.99
2        129.75
3         99.95
4        350.89
```

Desenvolva um aplicativo Java que recebe entrada de itens vendidos por um vendedor durante a última semana e calcula e exibe os rendimentos do vendedor. Não existe nenhum limite para o número de itens que pode ser vendido.

4.20 **(*Calculador de salários*)** Desenvolva um aplicativo Java que determina o salário bruto de cada um de três empregados. A empresa paga as horas normais pelas primeiras 40 horas trabalhadas de cada funcionário e 50% a mais pelas horas trabalhadas além das 40 horas. Você recebe uma lista de empregados, o número de horas trabalhadas que eles trabalharam na semana passada e o salário-hora de cada empregado. Seu programa deve aceitar a entrada dessas informações para cada empregado e, então, determinar e exibir o salário bruto do empregado. Utilize a classe `Scanner` para inserir os dados.

4.21 **(*Localize o maior número*)** O processo de localizar o maior valor é muito utilizado em aplicativos de computador. Por exemplo, um programa que determina o vencedor de uma competição de vendas inseriria o número de unidades vendidas por cada vendedor. O vendedor que vende mais unidades ganha a competição. Escreva um programa em pseudocódigo e, então, um aplicativo Java que aceita como entrada uma série de 10 inteiros e determina e imprime o maior dos inteiros. Seu programa deve utilizar pelo menos três variáveis a seguir:

a) `counter`: um contador para contar até 10 (isto é, monitorar quantos números foram inseridos e determinar quando todos os 10 números foram processados).

b) `number`: o inteiro mais recentemente inserido pelo usuário.

c) `largest`: o maior número encontrado até agora.

4.22 **(*Saída no formato de tabela*)** Escreva um aplicativo Java que utiliza um loop para imprimir a seguinte tabela de valores:

```
N       10*N    100*N   1000*N
1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.23 (***Encontre os dois números maiores***) Usando uma abordagem semelhante àquela do Exercício 4.21, encontre os dois maiores valores entre os 10 valores inseridos. [*Nota*: Você só pode inserir cada número uma vez.]

4.24 (***Validando entrada de usuário***) Modifique o programa na Figura 4.12 para validar suas entradas. Para qualquer entrada, se o valor inserido for diferente de 1 ou 2, continue o loop até o usuário inserir um valor correto.

4.25 O que o seguinte programa imprime?

```
1  // Exercícios 4.25: Mystery2.java
2  public class Mystery2
3  {
4     public static void main( String[] args )
5     {
6        int count = 1;
7
8        while ( count <= 10 )
9        {
10          System.out.println( count % 2 == 1 ? "****" : "++++++++" );
11          ++count;
12       } // fim do while
13    } // fim de main
14 } // fim da classe Mystery2
```

4.26 O que o seguinte programa imprime?

```
1  // Exercícios 4.26: Mystery3.java
2  public class Mystery3
3  {
4     public static void main( String[] args )
5     {
6        int row = 10;
7        int column;
8
9        while ( row >= 1 )
10       {
11          column = 1;
12
13          while ( column <= 10 )
14          {
15             System.out.print( row % 2 == 1 ? "<" : ">" );
16             ++column;
17          } // fim do while
18
19          --row;
20          System.out.println();
21       } // fim do while
22    } // fim de main
23 } // fim da classe Mystery3
```

4.27 (***Problema do* `else` *oscilante***) Determine a saída para cada um dos conjuntos dados de código quando `x` é `9` e `y` é `11` e quando `x` é `11` e `y` é `9`. Observe que o compilador ignora o recuo em um programa Java. Da mesma forma, o compilador Java sempre associa um `else` com o `if` imediatamente precedente a menos que instruído a fazer de outro modo pela colocação de chaves (`{}`). À primeira vista, o programador pode não ter certeza de qual `if` um `else` particular corresponde — essa situação é conhecida como "problema do `else` oscilante" Eliminamos o recuo do seguinte código para tornar o problema mais desafiador. [*Dica*: aplique as convenções de recuo que você aprendeu.]

a)
```
if ( x < 10 )
if ( y > 10 )
System.out.println( "*****" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
```

b)
```
if ( x < 10 )
{
```

```
if ( y > 10 )
System.out.println( "*****"  );
}
else
{
System.out.println( "#####" );
System.out.println( "$$$$$" );
}
```

4.28 (***Outro problema de*** `else` ***oscilante***) Modifique o código dado para produzir a saída mostrada em cada parte do problema. Use técnicas de recuo adequadas. Não faça nenhuma alteração além de inserir chaves e alterar o recuo do código. O compilador ignora recuos em um programa Java. Eliminamos o recuo do código fornecido para tornar o problema mais desafiador. [*Nota*: Talvez não sejam necessárias modificações para algumas partes].

```
if ( y == 8 )
if ( x == 5 )
System.out.println( "@@@@@" );
else
System.out.println( "#####" );
System.out.println( "$$$$$" );
System.out.println( "&&&&&" );
```

a) Supondo que `x = 5` e `y = 8`, a seguinte saída é produzida:

```
@@@@@
$$$$$
&&&&&
```

b) Supondo que `x = 5` e `y = 8`, a seguinte saída é produzida:

```
@@@@@
```

c) Supondo que `x = 5` e `y = 8`, a seguinte saída é produzida:

```
@@@@@
```

d) Supondo que `x = 5` e `y = 7`, a seguinte saída é produzida. [*Nota*: todas as três últimas instruções de saída depois do `else` são partes de um bloco].

```
#####
$$$$$
&&&&&
```

4.29 (***Quadrado de asteriscos***) Escreva um aplicativo que solicita para o usuário inserir o tamanho do lado de um quadrado e, então, exibe um quadrado vazio desse tamanho com asteriscos. Seu programa deve trabalhar com quadrados de todos os comprimentos de lado possíveis entre 1 e 20.

4.30 (***Palíndromos***) Um palíndromo é uma sequência de caracteres que é lida da esquerda para a direita ou da direita para a esquerda. Por exemplo, cada um dos seguintes inteiros de cinco dígitos é um palíndromo: 12321, 55555, 45554 e 11611. Escreva um aplicativo que lê em um inteiro de cinco dígitos e determina se ele é ou não um palíndromo. Se o número não for de cinco dígitos, exiba uma mensagem de erro e permita que o usuário insira um novo valor.

4.31 (***Imprimindo o equivalente decimal de um número binário***) Escreva um aplicativo que insere um número inteiro que contém somente 0s e 1s (isto é, um número inteiro binário) e imprime seu equivalente decimal. [*Dica*: utilize os operadores de resto e divisão para pegar os dígitos do número binário um de cada vez da direita para a esquerda. No sistema de números decimais, o dígito mais à direita tem um valor posicional de 1 e o próximo dígito à esquerda um valor posicional de 10, depois 100, depois 1000 e assim por diante. O número decimal 234 pode ser interpretado como 4 * 1 + 3 * 10 + 2 * 100. No sistema de número binários, o dígito mais à direita tem um valor posicional de 1, o próximo dígito à esquerda um valor posicional de 2, depois 4, depois 8 e assim por diante. O equivalente decimal do binário 1101 é 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 ou 1 + 0 + 4 + 8 ou, 13].

4.32 (***Padrão de tabuleiro de damas de asteriscos***) Escreva um aplicativo que utiliza apenas as instruções de saída

```
System.out.print( "* " );
System.out.print( " " );
System.out.println();
```

para exibir o padrão de tabuleiro de damas a seguir. Observe que uma chamada de método `System.out.println` sem argumentos faz com o programa gere saída de um único caractere de nova linha. [*Dica:* instruções de repetição são necessárias].

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.33 (***Múltiplos de 2 com um loop infinito***) Escreva um aplicativo que continue exibindo na janela de comando os múltiplos do inteiro 2 — a saber, 2, 4, 8, 16, 32, 64 e assim por diante. Seu loop não deve terminar (isto é, deve criar um loop infinito). O que acontece quando se executa esse programa?

4.34 O que há de errado com a seguinte instrução? Forneça a instrução correta para adicionar um à soma de `x` e `y`.

```
System.out.println( ++(x + y) );
```

4.35 (***Lados de um triângulo***) Escreva um aplicativo que lê três valores diferentes de zero inseridos pelo usuário e determina e imprime se eles poderiam representar os lados de um triângulo.

4.36 (***Lados de um triângulo direito***) Escreva um aplicativo que lê três inteiros diferentes de zero e determina e imprime se eles poderiam representar os lados de um triângulo direito.

4.37 (***Fatorial***) O fatorial de um inteiro não negativo n é escrito como $n!$ (pronuncia-se "n fatorial") e é definido como segue:

$n! = n \cdot (n - 1) * (n - 2) * \ldots \cdot 1$ (para valores de n maiores ou iguais a 1)

e

$n! = 1$ (para $n = 0$).

Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, o que dá 120.

a) Escreva um aplicativo que lê um inteiro não negativo, calcula e imprime seu fatorial.

b) Escreva um aplicativo que estima o valor da constante matemática e utilizando a fórmula a seguir. Permita ao usuário inserir o número de termos a calcular.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

c) Escreva um aplicativo que computa o valor de e^x utilizando a fórmula a seguir. Permita ao usuário inserir o número de termos a calcular.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Fazendo a diferença

4.38 (***Impondo privacidade com criptografia***) O explosivo crescimento das comunicações e o armazenamento de dados na Internet e em computadores conectados pela Internet aumentou muito a preocupação com a privacidade. O campo da criptografia envolve a codificação de dados para torná-los difíceis de acessar (e espera-se — com os esquemas mais avançados — impossíveis de acessar) para usuários sem autorização de leitura. Nesse exercício você investigará um esquema simples para criptografar e descriptografar dados. Uma empresa que quer enviar dados pela Internet pediu-lhe para escrever um programa que criptografe dados para que estes possam ser transmitidos com maior segurança. Todos os dados são transmitidos como inteiros de quatro dígitos. Seu aplicativo deve ler um inteiro de quatro dígitos inserido pelo usuário e criptografá-lo desta maneira: substitua cada dígito pelo resultado da adição de 7 ao dígito e obtendo o resto depois da divisão do novo valor por 10. Troque então o primeiro dígito pelo terceiro e o segundo dígito pelo quarto. Então, imprima o inteiro criptografado. Escreva um aplicativo separado que recebe entrada de um inteiro de quatro dígitos criptografado e o descriptografa (revertendo o esquema de criptografia) para formar o número original. [*Projeto de leitura opcional:* pesquise "criptografia de chave pública" em geral e, especificamente, o esquema de chave pública PGP (Pretty Good Privacy). Você também pode querer investigar o esquema RSA, que é amplamente usado em aplicativos robustos de capacidads industriais.]

4.39 (***Crescimento demográfico mundial***) A população mundial cresceu consideravelmente ao longo dos séculos. O crescimento contínuo pode, por fim, desafiar os limites de ar respirável, água potável, terra fértil para agricultura e outros recursos limitados. Há evidência de que o crescimento tem reduzido a velocidade nos últimos anos e que a população mundial pode chegar ao ponto máximo em algum momento nesse século e, então, começar a diminuir.

Para esse exercício, pesquise questões de crescimento demográfico mundial online. *Não deixe de investigar vários pontos de vista.* Obtenha estimativas da população mundial atual e sua taxa de crescimento (a porcentagem pela qual provavelmente aumentará esse ano). Escreva um programa que calcule o crescimento demográfico mundial anual dos próximos 75 anos, *utilizando a premissa simplificadora de que a taxa de crescimento atual ficará constante*. Imprima os resultados em uma tabela. A primeira coluna deve exibir o ano do ano 1 ao ano 75. A segunda coluna deve exibir a população mundial esperada no fim desse ano. A terceira deve exibir o aumento numérico na população mundial que ocorreria nesse ano. Utilizando os seus resultados, determine o ano em que a população dobraria ao que é hoje, se a taxa de crescimento deste ano persistisse.

Not everything that can be counted counts, and not every thing that counts can be counted. [Nem tudo que pode ser contado importa e nem tudo que importa pode ser contado.]
— Albert Einstein

Quem pode controlar seu destino?
— William Shakespeare

A chave utilizada é sempre brilhante.
— Benjamin Franklin

Inteligência... é a faculdade de criar objetos artificiais, especialmente ferramentas para fazer ferramentas.
— Henri Bergson

Toda vantagem no passado é julgada à luz do resultado final.
— Demóstenes

5

Instruções de controle: Parte 2

Objetivos

Neste capítulo, você aprenderá:

- Os princípios básicos da repetição controlada por contador.
- A utilizar as instruções de repetição `for` e `do...while` para executar instruções em um programa repetidamente.
- A entender a seleção múltipla utilizando a instrução de seleção `switch`.
- A utilizar as instruções `break` e `continue` para alterar o fluxo de controle.
- A utilizar os operadores lógicos para formar expressões condicionais complexas em instruções de controle.

Sumário

5.1 Introdução
5.2 Princípios básicos de repetição controlada por contador
5.3 Instrução de repetição `for`
5.4 Exemplos com a estrutura `for`
5.5 Instrução de repetição `do...while`
5.6 A estrutura de seleção múltipla `switch`
5.7 Instruções `break` e `continue`
5.8 Operadores lógicos
5.9 Resumo de programação estruturada
5.10 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando retângulos e ovais
5.11 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

5.1 Introdução

Este capítulo continua nossa apresentação da teoria e dos princípios da programação estruturada introduzindo todos, exceto uma das instruções de controle restantes do Java. Demonstramos as instruções `for`, `do...while` e `switch` do Java. Por uma série de breves exemplos que utilizam `while` e `for`, exploramos os princípios básicos da repetição controlada por contador. Criamos uma versão da classe `GradeBook` que utiliza uma instrução `switch` para contar o número de equivalentes de notas A, B, C, D e F em um conjunto de notas numéricas inseridas pelo usuário. Introduzimos as instruções de controle de programa `break` e `continue`. Discutimos os operadores lógicos do Java, que permitem-lhe utilizar expressões condicionais mais complexas em instruções de controle. Por fim, resumimos as instruções de controle e as comprovadas técnicas de resolução de problemas do Java apresentadas neste capítulo e no Capítulo 4.

5.2 Princípios básicos de repetição controlada por contador

Esta seção utiliza a instrução de repetição `while` introduzida no Capítulo 4 para formalizar os elementos necessários para realizar a repetição controlada por contador. Repetição controlada por contador requer:

1. Uma **variável de controle** (ou contador de loop).
2. O **valor inicial** da variável de controle.
3. O **incremento** (ou **decremento**) pelo qual a variável de controle é modificada a cada passagem pelo loop (também conhecido como **cada iteração do loop**).
4. A **condição de continuação do loop** que determina se o loop deve continuar.

Para ver esses elementos da repetição controlada por contador, considere o aplicativo da Figura 5.1, que utiliza um loop para exibir os números de 1 a 10.

```
1  // Figura 5.1: WhileCounter.java
2  // Repetição controlada por contador com a instrução de repetição while.
3
4  public class WhileCounter
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // declara e inicializa a variável de controle
9
10       while (counter <= 10 ) // condição de continuação do loop
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // incrementa a variável de controle por 1
14       } // fim do while
15
16       System.out.println(); // imprime uma nova linha
17    } // fim de main
18 } // fim da classe WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.1 | Repetição controlada por contador com a instrução de repetição `while`.

Na Figura 5.1, os elementos da repetição controlada por contador são definidos nas linhas 8, 10 e 13. A linha 8 declara a variável de controle (`counter`) como um `int`, reserva espaço para ele na memória e configura seu valor inicial como `1`. A variável `counter` também poderia ter sido declarada e inicializada com as seguintes instruções de declaração e atribuição de variável local:

```
int counter; // declara contador
counter = 1; // inicializa o contador como 1
```

A linha 12 exibe o valor da variável de controle `counter` durante cada iteração do loop. A linha 13 incrementa a variável de controle por 1 para cada iteração do loop. A condição de continuação do loop no `while` (linha 10) testa se o valor da variável de controle é menor ou igual a `10` (o valor final para o qual a condição é `true`). Observe que o programa realiza o corpo desse `while` mesmo quando a variável de controle é `10`. O loop termina quando a variável de controle excede `10` (isto é, `counter` torna-se `11`).

Erro comum de programação 5.1

Uma vez que valores de ponto flutuante podem ser aproximados, controlar loops com variáveis de ponto flutuantes pode resultar em valores de contador imprecisos e testes de terminação imprecisos.

Dica de prevenção de erro 5.1

Utilize números inteiros para controlar loops de contagem.

Boa prática de programação 5.1

Coloque linhas em branco acima e abaixo das instruções de controle de repetição e seleção e recue os corpos da instrução para aprimorar a legibilidade.

O programa na Figura 5.1 pode tornar-se mais conciso inicializando `counter` como `0` na linha 8 e pré-incrementando `counter` na condição `while` como segue:

```
while ( ++counter <= 10 ) // condição de continuação do loop
   System.out.printf( "%d  ", counter );
```

Esse código poupa uma instrução (e elimina a necessidade de colocar o corpo do loop entre chaves), porque a condição `while` realiza o incremento antes de testar a condição. (Como discutido na Seção 4.12, a precedência de `++` é mais alta que a de `<=`.) A codificação de um modo tão condensado exige prática, talvez torne o código mais difícil de ler, depurar, modificar e manter e, em geral, deve ser evitada.

Observação de engenharia de software 5.1

"Manter a coisa simples" é um bom conselho para a maior parte do código que você escreverá.

5.3 Instrução de repetição for

A Seção 5.2 apresentou os princípios básicos da repetição controlada por contador. A instrução `while` pode ser utilizada para implementar qualquer loop controlado por contador. O Java também fornece a **instrução de repetição** `for`, que especifica os detalhes da repetição controlada por contador em uma única linha de código. A Figura 5.2 reimplementa o aplicativo da Figura 5.1 utilizando `for`.

```
1  // Figura 5.2: ForCounter.java
2  // Repetição controlada por contador com a instrução de repetição for.
3
4  public class ForCounter
5  {
6     public static void main( String[] args )
7     {
8        // cabeçalho da instrução for inclui inicialização,
9        // condição de continuação do loop e incremento
10       for ( int counter = 1; counter <= 10; counter++ )
11          System.out.printf( "%d  ", counter );
12
13       System.out.println(); // imprime uma nova linha
14    } // fim de main
15 } // fim da classe ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.2 | Repetição controlada por contador com a instrução de repetição `for`.

O método `main` do aplicativo opera como segue: quando a instrução `for` (linhas 10–11) começar a executar, a variável de controle `counter` é declarada e inicializada como `1`. (A partir do que foi discutido na Seção 5.2, lembre-se de que os primeiros dois elementos da repetição controlada por contador são a variável de controle e seu valor inicial.) Em seguida, o programa verifica a condição de continuação do loop, `counter <= 10` , que está entre os dois pontos-e-vírgulas requeridos. Como o valor inicial de `counter` é `1`, a condição é inicialmente verdadeira. Portanto, a instrução de corpo (linha 11) exibe o valor da variável de controle `counter`, a saber `1`. Depois de executar o corpo do loop, o programa incrementa `counter` na expressão `counter++`, que aparece à direita do segundo ponto-e-vírgula. Então o teste de continuação do loop é realizado novamente para determinar se o programa deve continuar com a próxima iteração do loop. Nesse ponto, o valor da variável de controle é `2`, então a condição ainda é verdadeira (o valor final não é excedido) — portanto, o programa realiza a instrução de corpo novamente (isto é, a próxima iteração do loop). Esse processo continua até que os números de 1 a 10 tenham sido exibidos e o valor de `counter` torne-se `11`, fazendo com que o teste de continuação do loop falhe e que a repetição seja finalizada (na linha 11, depois de 10 repetições do corpo do loop). Então o programa realiza a primeira instrução depois do `for` — nesse caso, a linha 13.

Observe que a Figura 5.2 utiliza (na linha 10) a condição de continuação do loop `counter <= 10`. Se você especificasse `counter < 10` incorretamente como a condição, o loop só iteraria nove vezes. Esse é um erro de lógica comum chamado um **erro por um** (*off-by-one*).

Erro comum de programação 5.2

Um operador relacional incorreto ou um valor final incorreto de um contador de loop na condição de continuação do loop de uma instrução de repetição pode causar um erro por um.

Dica de prevenção de erro 5.2

Usar o valor final na condição de uma instrução `while` ou `for` e usar o operador relacional `<=` ajuda a evitar erros por um. Para um loop que imprime os valores de 1 a 10, a condição de continuação do loop deve ser `counter <= 10` em vez de `counter < 10` (o que causa um erro por um) ou `counter < 11` (o que é correto). Muitos programadores preferem a chamada contagem baseada em zero, em que para contar 10 vezes, `counter` seria inicializado como zero e o teste de continuação do loop seria `counter < 10`.

A Figura 5.3 examina mais detalhadamente a instrução `for` na Figura 5.2. A primeira linha de `for` (incluindo a palavra-chave `for` e tudo mais entre parênteses depois de `for`) — linha 10 na Figura 5.2 — é às vezes chamado **cabeçalho da instrução `for`**. Observe que o cabeçalho de `for` "faz tudo" — ele especifica cada item necessário para repetição controlada por contador com uma variável de controle. Se houver mais de uma instrução no corpo do `for`, as chaves (`{` e `}`) são exigidas para definir o corpo do loop.

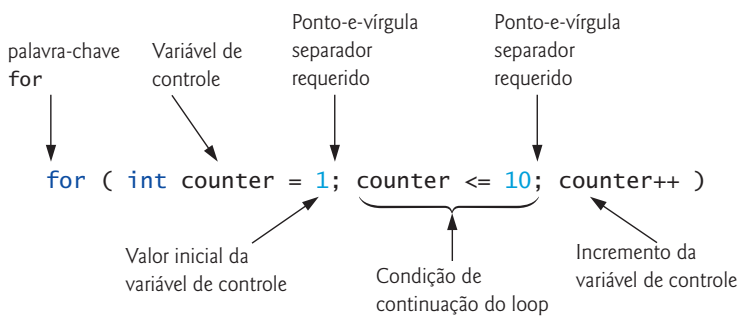


Figura 5.3 | Componentes de cabeçalho de instrução `for`.

O formato geral da instrução `for` é:

```
for ( inicialização; condiçãoDeContinuaçãoDoLoop; incremento )
   instrução
```

onde a expressão inicialização nomeia a variável de controle do loop e fornece opcionalmente seu valor inicial, *condiçãoDeContinuaçãoDoLoop* determina se o loop deve continuar executando e incremento modifica o valor da variável de controle (possivelmente um incremento ou decremento), para que a condição de continuação do loop por fim se torne falsa. Os dois pontos-e-vírgulas no cabeçalho `for` são necessários.

Erro comum de programação 5.3

Utilizar vírgulas em vez dos dois pontos-e-vírgulas obrigatórios em um cabeçalho `for` é um erro de sintaxe.

Na maioria dos casos, a instrução `for` pode ser representada com uma instrução `while` equivalente como segue:

```
inicialização;

while ( condiçãoDeContinuaçãoDoLoop )
{
   instrução
   incremento;
}
```

Na Seção 5.7, mostramos um caso em que uma instrução `for` não pode ser representada com uma instrução `while` equivalente.

Em geral, as instruções `for` são utilizadas para repetição controlada por contador e as instruções `while` são utilizadas para repetição controlada por sentinela. Entretanto, `while` e `for` podem ser utilizados para qualquer tipo de repetição.

Se a expressão inicialização no cabeçalho `for` declara a variável de controle (isto é, o tipo da variável de controle é especificado antes do nome variável, como na Figura 5.2), a variável de controle pode ser utilizada somente nessa instrução `for` — ela não existirá fora dela. Esse uso restrito é conhecido como **escopo** da variável. O escopo de uma variável define onde ele pode ser utilizado em um programa. Por exemplo, uma variável local só pode ser utilizada no método que a declara e somente a partir do ponto de declaração até o fim do método. O escopo é discutido em detalhes no Capítulo 6, "Métodos: uma visão mais aprofundada".

Erro comum de programação 5.4

Quando a variável de controle de uma instrução `for` for declarada na seção de inicialização do cabeçalho de `for`, utilizar a variável de controle depois do corpo de `for` é um erro de compilação.

Todas as três expressões em um cabeçalho `for` são opcionais. Se a *condiçãoDeContinuaçãoDoLoop* for omitida, o Java assume que a condição de continuação do loop é sempre verdadeira, criando assim um loop infinito. Você pode omitir a expressão de *inicialização* se o programa inicializar a variável de controle antes do loop. Você pode omitir a expressão de *incremento* se o programa calcular o incremento com instruções no corpo do loop ou se nenhum incremento for necessário. A expressão incremento em uma instrução `for` atua como se ela fosse uma instrução independente no fim do corpo de `for`. Portanto, as expressões:

```
counter = counter + 1
counter += 1
++counter
counter++
```

são expressões de incremento equivalentes em uma instrução `for`. Muitos programadores preferem `counter++` porque é conciso e porque um loop `for` avalia sua expressão de incremento depois que seu corpo executa. Portanto, a forma de incremento pós-fixa parece mais natural. Nesse caso, a variável sendo incrementada não aparece em uma expressão maior, então pré-incrementar a pós-incrementar realmente têm o mesmo efeito.

Dica de desempenho 5.1

Há uma ligeira vantagem de desempenho em pré-incrementar, mas se você escolher pós-incrementar porque parece mais natural (como no cabeçalho de um `for`), otimizar os compiladores normalmente irá gerar o bytecode Java que, de todo modo, utiliza a forma mais eficiente. Dessa forma, você deve utilizar o idioma com o qual você se sente mais à vontade nessas situações.

Erro comum de programação 5.5

Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito do cabeçalho de um `for` torna o corpo desse `for` uma instrução vazia. Em geral, isso é um erro de lógica.

Dica de prevenção de erro 5.3

Os loops infinitos ocorrem quando a condição de continuação do loop em uma instrução de repetição nunca se torna `false`. Para evitar essa situação em um loop controlado por contador, assegure que a variável de controle é incrementada (ou decrementada) a cada iteração do loop. Em um loop controlado por sentinela, certifique-se de que o valor da sentinela é capaz de ser inserido.

A inicialização, condição de continuação de loop e partes de incremento de uma estrutura `for` pode conter expressões aritméticas. Por exemplo, assuma que `x = 2` e `y = 10`. Se `x` e `y` não forem modificados no corpo do loop, a instrução

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

é equivalente à instrução

```
for ( int j = 2; j <= 80; j += 5 )
```

O incremento de uma instrução `for` também pode ser negativo, o que na verdade é um decremento e o loop conta para baixo.

Se a condição de continuação do loop for inicialmente `false`, o programa não executará o corpo da instrução `for`. Em vez disso, a execução prossegue com a instrução seguinte ao `for`.

Os programas costumam exibir o valor de variável de controle ou utilizá-lo em cálculos no corpo do loop, mas essa utilização não é necessária. A variável de controle é comumente utilizada para controlar a repetição sem ser mencionada no corpo da `for`.

Dica de prevenção de erro 5.4

Embora o valor da variável de controle possa ser alterado no corpo de um loop `for`, evite fazê-lo assim porque essa prática pode levar a erros sutis.

O diagrama de atividades UML da instrução `for` é semelhante ao da instrução `while` (Figura 4.4). A Figura 5.4 mostra o diagrama de atividades da instrução `for` na Figura 5.2. O diagrama deixa bem claro que a inicialização ocorre *uma vez* antes que o teste de continuação do loop seja avaliado pela primeira vez, e que o incremento ocorre a *cada* passagem pelo loop depois que a instrução do corpo executa.

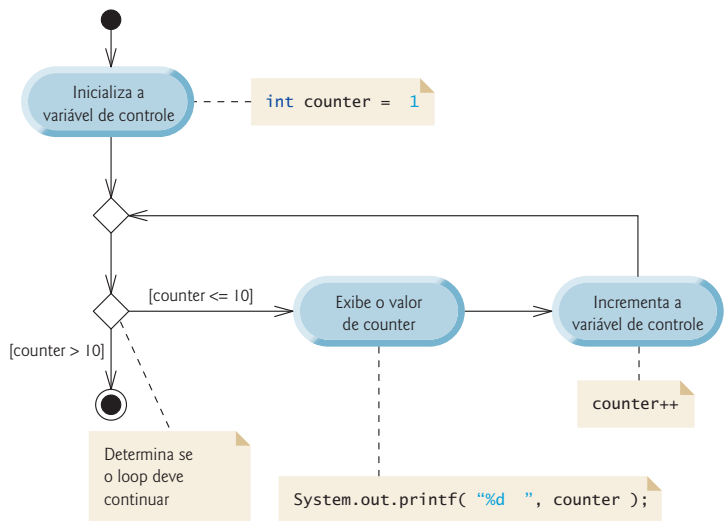


Figura 5.4 | Diagrama de atividades UML para a instrução `for` na Figura 5.2.

5.4 Exemplos com a estrutura for

Os próximos exemplos mostram técnicas de variar a variável de controle em uma instrução `for`. Em cada caso, escrevemos o cabeçalho `for` apropriado. Observe a alteração no operador relacional para loops que decrementam a variável de controle.

a) Varie a variável de controle de 1 a 100 em incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

b) Varie a variável de controle de 100 a 1 em decrementos de 1.

```
for ( int i = 100; i >= 1; i-- )
```

c) Varie a variável de controle de 7 a 77 em incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Varie a variável de controle de 20 a 2 em decrementos de 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Varie a variável de controle pelos valores 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Varie a variável de controle pelos valores 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Erro comum de programação 5.6

Utilizar um operador relacional incorreto na condição de continuação de um loop que conta para baixo (por exemplo, utilizar `i <= 1` em vez de `i >= 1` em uma contagem de loop para baixo até 1) normalmente é um erro de lógica.

Aplicativo: somando os inteiros pares de 2 a 20

Agora consideramos dois aplicativos de exemplo que demonstram as utilizações simples de `for`. O aplicativo na Figura 5.5 utiliza uma instrução `for` para somar os inteiros pares de 2 a 20 e armazenar o resultado em uma variável `int` chamada `total`.

```
1  // Figura 5.5: Sum.java
2  // Somando inteiros com a instrução for.
3
4  public class Sum
5  {
6     public static void main( String[] args )
7     {
8        int total = 0; // inicializa o total
9
10       // total de inteiros pares de 2 a 20
11       for ( int number = 2; number <= 20; number += 2 )
12          total += number;
13
14       System.out.printf( "Sum is %d\n", total ); // exibe os resultados
15    } // fim de main
16 } // fim da classe Sum
```

```
Sum is 110
```

Figura 5.5 | Somando inteiros com a instrução `for`.

As expressões inicialização e incremento podem ser listas separadas por vírgulas de expressões que permitem-lhe utilizar múltiplas expressões de inicialização ou múltiplas expressões de incremento. Por exemplo, embora não seja aconselhável, o corpo da instrução `for` nas linhas 11–12 da Figura 5.5 poderia ser mesclado na parte de incremento do cabeçalho `for` utilizando uma vírgula, como a seguir:

```
for ( int number = 2; number <= 20; total += number, number += 2 )
   ; // estrutura vazia
```

Boa prática de programação 5.2

Para maior legibilidade, limite o tamanho de cabeçalhos da instrução de controle a uma única linha se possível.

Aplicativo: cálculos de juros compostos

O próximo aplicativo utiliza a instrução `for` para calcular juros compostos. Considere o seguinte problema:

Uma pessoa investe $1.000 em uma conta-poupança que rende juros de 5% ao ano. Assumindo que todo o juro é deixado em depósito, calcule e imprima a quantidade de dinheiro na conta no fim de cada ano por 10 anos. Utilize a seguinte fórmula para determinar as quantidades:

$$a = p\,(1 + r)^n$$

onde

p é a quantidade original investida (isto é, o principal);
r é a taxa de juros anual (por exemplo, use 0,05 para 5%);
n é o número de anos;
a é a quantidade em depósito no fim do n-ésimo ano.

A solução para esse problema (Figura 5.6) envolve um loop que realiza o cálculo indicado para cada um dos 10 anos que o dinheiro permanece em depósito. As linhas 8–10 no método `main` declaram as variáveis `double` `amount`, `principal` e `rate`, e inicializam `principal` em `1000.0` e `rate` em `0.05`. O Java trata as constantes de ponto flutuante como `1000.0` e `0.05` como tipo `double`. De maneira semelhante, o Java trata as constantes inteiras, por exemplo, `7` e `-22`, como tipo `int`.

```
1  // Figura 5.6: Interest.java
2  // Cálculos de juros compostos com for.
3
4  public class Interest
5  {
6     public static void main( String[] args )
7     {
8        double amount; // quantia em depósito ao fim de cada ano
9        double principal = 1000.0; // quantidade inicial antes dos juros
10       double rate = 0.05; // taxa de juros
11
12       // exibe cabeçalhos
13       System.out.printf( "%s%20s \n", "Year", "Amount on deposit" );
14
15       // calcula quantidade de depósito para cada um dos dez anos
16       for ( int year = 1; year <= 10; year++ )
17       {
18          // calcula nova quantidade durante ano especificado
19          amount = principal * Math.pow( 1.0 + rate, year );
20
21          // exibe o ano e a quantidade
22          System.out.printf( "%4d%,20.2f\n", year, amount );
23       } // for final
24    } // fim de main
25 } // fim da classe Interest
```

```
Year   Amount on deposit
   1           1,050.00
   2           1,102.50
   3           1,157.63
   4           1,215.51
   5           1,276.28
   6           1,340.10
   7           1,407.10
   8           1,477.46
   9           1,551.33
  10           1,628.89
```

Figura 5.6 | Cálculos de juros compostos com `for`.

Formatando strings com tamanhos de campo e justificação

A linha 13 gera a saída dos cabeçalhos nessas duas colunas do aplicativo de saída. A primeira coluna exibe o ano, e a segunda o valor depositado no final deste ano. Observe que utilizamos o especificador de formato `%20s` para gerar a saída da `String` `"Amount on Deposit"`. O inteiro `20` entre o `%` e o caractere de conversão `s` indica que a saída de valor deve ser exibida com uma **largura de campo** de 20 — isto é, `printf` exibe o valor com pelo menos 20 posições de caractere. Se o valor enviado para a saída for menor que 20 posições de caractere (17 caracteres neste exemplo), o valor é **alinhado à direita** no campo por padrão. Se o valor `year` enviado para a saída tivesse largura maior do que quatro posições de caractere, a largura de campo seria estendida à direita para acomodar o valor inteiro — isso empurraria o campo `amount` para a direita, desalinhando as colunas de nossa saída tabular. Para indicar que os valores devem ser enviados para a saída **alinhados à esquerda**, simplesmente preceda a largura de campo com o **flag de formatação do sinal de subtração (–)**. (isto é, `%-20s`).

Calculando os juros

A instrução `for` (linhas 16–23) executa seu corpo 10 vezes, variando a variável de controle `year` de 1 a 10 em incrementos de 1. Esse loop termina quando a variável de controle `year` torna-se 11. (Observe que `year` representa *n* na definição do problema.)

As classes fornecem métodos que executam tarefas comuns em objetos. De fato, a maioria dos métodos deve ser chamada em um objeto específico. Por exemplo, para gerar saída de texto na Figura 5.6, a linha 13 chama o método `printf` no objeto `System.out`. Muitas classes também fornecem métodos que realizam tarefas comuns e não exigem objetos. Estes são chamados métodos `static`. Por exemplo, o Java não inclui um operador de exponenciação, então os engenheiros da classe `Math` do Java definiram o método `static` `pow` para elevar um valor a uma potência. Você pode chamar um método `static` especificando o nome da classe seguido por um ponto (`.`) e o nome de método:

NomeDaClasse`.`*nomeDoMétodo*`(` *argumentos* `)`

No Capítulo 6, você aprenderá a implementar métodos `static` em suas próprias classes.

Utilizamos o método `static` **pow** da classe **Math** para realizar o cálculo de juros compostos na Figura 5.6. `Math.pow(x, y)` calcula o valor de x elevado à y-ésima potência. O método recebe dois argumentos `double` e retorna um valor `double`. A linha 19 realiza o cálculo $a = p\ (1 + r)^n$, onde a é `amount`, p é `principal`, r é `rate` e n é `year`. Observe que a classe `Math` é definida no pacote `java.lang`, portanto você não precisa importar a classe `Math` para utilizá-la.

Observe que o corpo da instrução `for` contém o cálculo `1.0 + rate`, que aparece como um argumento para o método `Math.pow`. De fato, esse cálculo produz o mesmo resultado toda vez pelo loop, então repeti-lo a cada iteração do loop é perda de tempo.

Dica de desempenho 5.2

Em loops, evite cálculos para os quais o resultado nunca muda — esses cálculos em geral devem ser colocados antes do loop. [*Nota: Muitos compiladores de otimização sofisticados de hoje colocarão esses cálculos fora de loops no código compilado.*]

Formatando números de ponto flutuante

Depois de cada cálculo, a linha 22 envia para a saída o ano e a quantia em depósito no fim desse ano. O ano é enviado para a saída na largura de um campo de quatro caracteres (como especificado por `%4d`). A quantidade enviada para a saída é como um número de ponto flutuante com o especificador de formato `%,20.2f`. O **flag de formatação vírgula (,)** indica que o valor de ponto flutuante deve ser enviado para a saída com um **separador de agrupamento**. O separador real utilizado é específico à localidade do usuário (isto é, país). Por exemplo, nos Estados Unidos, o número será enviado para saída utilizando vírgulas para separar cada três dígitos e um ponto de fração decimal para separar a parte fracionária do número, como em 1,234.45. O número `20` na especificação de formato indica que o valor deve ser enviado para a saída alinhado à direita em uma largura de campo de 20 caracteres. O `.2` especifica a precisão do número formatado — nesse caso, o número é arredondado para o centésimo mais próximo e enviado para saída com dois dígitos à direita do ponto de fração decimal.

Uma advertência sobre a exibição de valores arredondados

Declaramos as variáveis `amount`, `principal` e `rate` como sendo do tipo `double` nesse exemplo. Lidaremos com partes fracionárias de valores monetários e, portanto, precisamos de um tipo que permita pontos de fração decimal em seus valores. Infelizmente, os números de ponto flutuante podem causar problemas. Eis uma explicação simples do que pode dar errado ao utilizar `double` (ou `float`) para representar quantias monetárias (assumindo que quantias monetárias são exibidas com dois dígitos à direita do ponto decimal): duas quantidades de dólar `double` armazenadas na máquina poderiam ser 14,234 (que normalmente seria arredondado para 14,23 para propósitos de exibição) e 18,673 (que normalmente seria arredondado para 18,67 para propósitos de exibição). Quando essas quantidades são somadas, produz-se a soma interna 32,907, que normalmente seria arredondada para 32,91 para propósitos de exibição. Portanto, sua saída poderia aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

mas uma pessoa que adiciona os números individuais como exibido esperaria que a soma fosse 32,90. Você foi avisado!

Dica de prevenção de erro 5.5

Não utilizar variáveis de tipo `double` (ou `float`) para realizar cálculos monetários precisos. A imprecisão dos números de ponto flutuante pode causar erros. Nos exercícios, você aprenderá como utilizar números inteiros para realizar cálculos monetários precisos. O Java também fornece a classe `java.math.BigDecimal` para realizar cálculos monetários precisos. Para informações adicionais, consulte `java.sun.com/javase/6/docs/api/java/math/BigDecimal.html`.

5.5 Instrução de repetição do...while

A **instrução de repetição do...while** é semelhante à instrução `while`. Na instrução `while`, o programa testa a condição de continuação do loop no início do loop, antes de executar o corpo do loop; se a condição for falsa, o corpo nunca será executado. A instrução `do...while` testa a condição de continuação do loop depois de executar o corpo do loop; portanto, o corpo sempre executa pelo menos uma vez. Quando uma instrução `do...while` termina, a execução continua com a próxima instrução na sequência. A Figura 5.7 usa uma instrução `do...while` (linhas 10–14) para gerar saída dos números 1–10.

```
1  // Figura 5.7: DoWhileTest.java
2  // instrução de repetição do…while.
3
4  public class DoWhileTest
5  {
```

```
6       public static void main( String[] args )
7       {
8          int counter = 1; // inicializa o contador
9
10         do
11         {
12            System.out.printf( "%d  ", counter );
13            ++counter;
14         } while ( counter <= 10 ); // fim da instrução do…while
15
16         System.out.println(); // gera saída de um caractere nova linha
17      } // fim de main
18   } // fim da classe DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 5.7 | Instrução de repetição do…while.

A linha 8 declara e inicializa a variável de controle `counter`. Ao entrar na instrução do...`while`, a linha 12 gera a saída do valor de `counter` e a linha 13 incrementa `counter`. Então o programa avalia o teste de continuação do loop na parte inferior do loop (linha 14). Se a condição for verdadeira, o loop continua a partir da primeira instrução de corpo na do...`while` (linha 12). Se a condição for falsa, o loop termina e o programa continua com a próxima instrução depois do loop.

A Figura 5.8 contém o diagrama de atividades da UML para a instrução do...`while`. Esse diagrama deixa bem claro que a condição de continuação do loop só é avaliada depois que o loop executa o estado de ação pelo menos uma vez. Compare esse diagrama de atividades com aquele da instrução `while` (Figura 4.4).

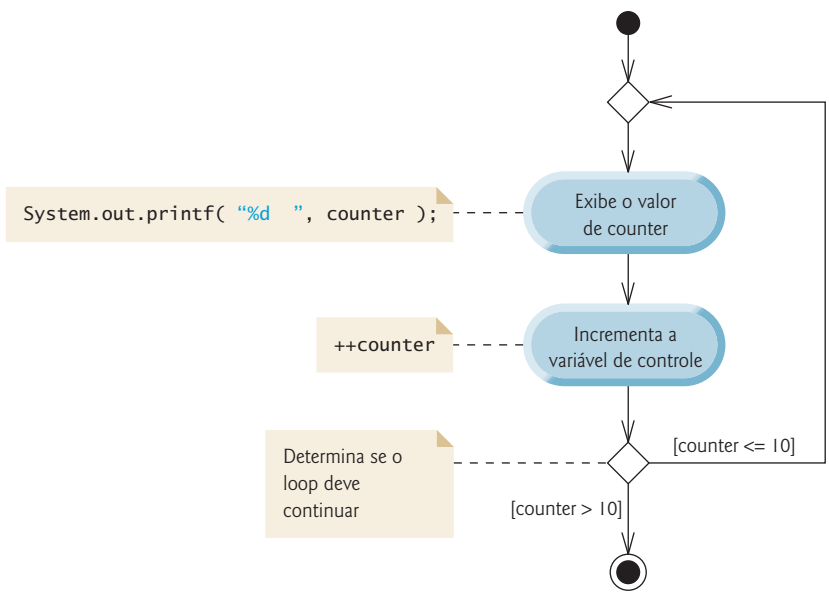


Figura 5.8 | Diagrama de atividades UML da instrução de repetição do…`while`.

Não é necessário usar chaves na instrução de repetição do...`while` se houver apenas uma instrução no corpo. Mas a maioria dos programadores inclui as chaves, para evitar confusão entre as instruções `while` e do...`while`. Por exemplo:

```
while ( condição )
```

em geral é a primeira linha de uma instrução `while`. Um do...`while` sem chaves em torno do corpo de uma única instrução aparece como:

```
do
   instrução
while ( condição );
```

o que pode ser confuso. Um leitor pode interpretar erroneamente a última linha — `while( condição );` — como uma instrução `while` contendo uma instrução vazia (o ponto-e-vírgula sozinho). Portanto, a instrução `do...while` com uma instrução no corpo é normalmente escrita assim:

```
do
{
   instrução
} while ( condição );
```

Boa prática de programação 5.3

Sempre inclua chaves em uma instrução `do...while`, mesmo se não forem necessárias. Isso ajuda a eliminar ambiguidade entre a instrução `while` e uma instrução `do...while` que contém apenas uma instrução.

5.6 A estrutura de seleção múltipla switch

O Capítulo 4 discutiu a instrução de seleção única `if` e a instrução de seleção dupla `if...else`. A **instrução de seleção múltipla `switch`** realiza ações diferentes com base nos possíveis valores de uma **expressão integral constante** do tipo `byte`, `short`, `int` ou `char`.

Classe GradeBook com a instrução switch para contar as notas A, B, C, D e F

A Figura 5.9 aprimora a classe `GradeBook` dos capítulos 3–4. A nova versão que agora apresentamos não apenas calcula a média de um conjunto de notas numéricas inseridas pelo usuário, mas também utiliza uma instrução `switch` para determinar se cada nota é o equivalente a um A, B, C, D ou F e incrementar o contador de notas escolares apropriado. A classe também exibe um resumo do número de alunos que recebeu cada nota. Consulte a Figura 5.10 para entrada e saída de exemplo do aplicativo `GradeBookTest` que utiliza a classe `GradeBook` para processar um conjunto de notas.

```
1  // Figura 5.9: GradeBook.java
2  // A classe GradeBook utiliza a instrução switch para contar as letras das notas escolares.
3  import java.util.Scanner; // programa utiliza a classe Scanner
4
5  public class GradeBook
6  {
7     private String courseName; // nome do curso que essa GradeBook representa
8     // variáveis de instância int são inicializadas em 0 por padrão
9     private int total; // soma das notas
10    private int gradeCounter; // número de notas inseridas
11    private int aCount; // contagem de notas A
12    private int bCount; // contagem de notas B
13    private int cCount; // contagem de notas C
14    private int dCount; // contagem de notas D
15    private int fCount; // contagem de notas F
16
17    // construtor inicializa courseName;
18    public GradeBook( String name )
19    {
20       courseName = name; // inicializa courseName
21    } // fim do construtor
22
23    // método para configurar o nome do curso
24    public void setCourseName( String name )
25    {
26       courseName = name; // armazena o nome do curso
27    } // fim do método setCourseName
28
29    // método para recuperar o nome do curso
30    public String getCourseName()
31    {
32       return courseName;
33    } // fim do método getCourseName
34
35    // exibe uma mensagem de boas-vindas para o usuário GradeBook
36    public void displayMessage()
37    {
38       // getCourseName obtém o nome do curso
```

```
39          System.out.printf( "Welcome to the grade book for\n%s!\n\n",
40             getCourseName() );
41       } // fim do método displayMessage
42
43       // insere número arbitrário de notas do usuário
44       public void inputGrades()
45       {
46          Scanner input = new Scanner( System.in );
47
48          int grade; // nota inserida pelo usuário
49
50          System.out.printf( "%s\n%s\n   %s\n   %s\n",
51             "Enter the integer grades in the range 0-100.",
52             "Type the end-of-file indicator to terminate input:",
53             "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
54             "On Windows type <Ctrl> z then press Enter" );
55
56          // faz loop até usuário inserir o indicador de fim do arquivo
57          while (input.hasNext() )
58          {
59             grade = input.nextInt(); // lê a nota
60             total += grade; // adiciona grade a total
61             ++gradeCounter; // incrementa o número de notas
62
63             // chama método para incrementar o contador adequado
64             incrementLetterGradeCounter( grade );
65          } // fim do while
66       } // fim do método inputGrades
67
68       // adiciona 1 ao contador adequado da nota especificada
69       private void incrementLetterGradeCounter( int grade )
70       {
71          // determina a nota que foi inserida
72          switch ( grade / 10 )
73          {
74             case 9:  // a nota estava entre 90
75             case 10: // e 100, inclusivo
76                ++aCount; // incrementa aCount
77                break; // necessário para sair de switch
78
79             case 8: // nota estava entre 80 e 89
80                ++bCount; // incrementa bCount
81                break; // sai do switch
82
83             case 7: // nota estava entre 70 e 79
84                ++cCount; // incrementa cCount
85                break; // sai do switch
86
87             case 6: // nota estava entre 60 e 69
88                ++dCount; // incrementa dCount
89                break; // sai do switch
90
91             default: // a nota era menor que 60
92                ++fCount; // incrementa fCount
93                break; // opcional; sairá do switch de qualquer jeito
94          } // fim do switch
95       } // fim do método incrementLetterGradeCounter
96
97       // exibe um relatório baseado nas notas inseridas pelo usuário
98       public void displayGradeReport()
99       {
100         System.out.println( "\nGrade Report:" );
101
102         // se usuário inseriu pelo menos uma nota...
103         if ( gradeCounter != 0 )
104         {
105            // calcula a média de todas as notas inseridas
106            double average = (double) total / gradeCounter;
107
```

```
108            // gera a saída de resumo de resultados
109            System.out.printf( "Total of the %d grades entered is %d\n",
110               gradeCounter, total );
111            System.out.printf( "Class average is %.2f\n", average );
112            System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
113               "Number of students who received each grade:",
114               "A: ", aCount,   // exibe número de notas A
115               "B: ", bCount,   // exibe número de notas B
116               "C: ", cCount,   // exibe número de notas C
117               "D: ", dCount,   // exibe número de notas D
118               "F: ", fCount ); // exibe número de notas F
119         } // fim do if
120         else // notas não foram inseridas, portanto imprime mensagem apropriada
121            System.out.println( "No grades were entered" );
122      } // fim do método displayGradeReport
123   } // fim da classe GradeBook
```

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares.

Como as versões anteriores da classe, a classe `GradeBook` (Figura 5.9) declara a variável de instância `courseName` (linha 7) e contém os métodos `setCourseName` (linhas 24–27), `getCourseName` (linhas 30–33) e `displayMessage` (linhas 36–41), que configuram o nome de curso, armazenam o nome de curso e exibem uma mensagem de boas-vindas para o usuário, respectivamente. A classe também contém um construtor (linhas 18–21) que inicializa o nome de curso.

A classe `GradeBook` também declara as variáveis de instância `total` (linha 9) e `gradeCounter` (linha 10), que monitoram a soma das notas inseridas pelo usuário e o número de notas inseridas, respectivamente. As linhas 11–15 declaram as variáveis contadoras para cada categoria de nota. A classe `GradeBook` mantém `total`, `gradeCounter` e os cinco contadores de notas baseadas em letras como variáveis de instância, de modo que possam ser usadas ou modificadas em qualquer dos métodos da classe. Note que o construtor da classe (linhas 18–21) configura apenas o nome de curso, porque as sete variáveis de instância restantes são `int`s e são inicializadas em `0` por padrão.

A classe `GradeBook` (Figura 5.9) contém três métodos adicionais — `inputGrades`, `incrementLetterGradeCounter` e `displayGradeReport`. O método `inputGrades` (linhas 44–66) lê um número arbitrário de notas integrais fornecidas pelo usuário utilizando repetição controlada por sentinela e atualiza as variáveis de instância `total` e `gradeCounter`. Esse método chama o método `incrementLetterGradeCounter` (linhas 69–95) para atualizar o contador adequado de notas baseadas em letras para cada nota inserida. O método `displayGradeReport` (linhas 98–122) gera a saída de um relatório contendo o total de todas as notas inseridas, a média das notas e o número de alunos que recebeu cada nota baseada em letra. Examinemos esses métodos em mais detalhes.

Método `inputGrades`

A linha 48 no método `inputGrades` declara a variável `grade`, que armazenará a entrada do usuário. As linhas 50–54 solicitam para o usuário inserir as notas integrais e digitar o indicador de fim do arquivo para terminar a entrada. O **indicador de fim do arquivo** é uma combinação de pressionamentos de tecla dependente de sistema que o usuário insere para indicar que não há dados a serem inseridos. No Capítulo 17, "Arquivos, fluxos e serialização de objetos", veremos como o indicador de fim do arquivo é utilizado quando um programa lê sua entrada a partir de um arquivo.

Nos sistemas UNIX/Linux/Mac OS X, o fim do arquivo é inserido digitando a sequência:

<Ctrl> d

em uma linha isolada. Essa notação significa pressionar simultaneamente a tecla Ctrl e a tecla d. Em sistemas Windows, o fim do arquivo pode ser inserido digitando:

<Ctrl> z

[*Nota*: em alguns sistemas, você deve pressionar *Enter* depois de digitar a sequência de teclas de fim do arquivo. Além disso, em geral o Windows exibe os caracteres `^Z` na tela quando o indicador de fim do arquivo é digitado, como é mostrado na saída da Figura 5.10.]

Dica de portabilidade 5.1

As combinações de teclas pressionadas para inserir o fim do arquivo são dependentes de sistema.

A instrução `while` (linhas 57–65) obtém a entrada de usuário. A condição na linha 57 chama o método `Scanner` **`hasNext`** para determinar se há mais entrada de dados. Esse método retorna o valor `boolean` `true` se houver mais dados; do contrário, ele retorna `false`. O valor retornado é então utilizado como o valor da condição na instrução `while`. Enquanto o indicador de fim do arquivo não tiver sido digitado, o método `hasNext` retornará `true`.

A linha 59 insere um valor de nota do usuário. A linha 60 utiliza o operador `+=` para adicionar `grade` a `total`. A linha 61 incrementa `gradeCounter`. O método `displayGradeReport` da classe utiliza essas variáveis para calcular a média das notas. A linha 64 chama o

método `incrementLetterGradeCounter` da classe (declarada nas linhas 69–95) para incrementar o contador apropriado de notas por letras com base na nota numérica inserida.

Método `incrementLetterGradeCounter`

O método `incrementLetterGradeCounter` contém uma instrução `switch` (linhas 72–94) que determina qual contador incrementar. Assumimos que o usuário insere uma nota válida no intervalo 0–100. Uma nota no intervalo 90–100 representa A, 80–89 representa B, 70–79 representa C, 60–69 representa D e 0–59 representa F. A instrução `switch` consiste em um bloco que contém uma sequência de **rótulos `case`** e um **caso `default`** opcional. Essas sequências são utilizadas nesse exemplo para determinar qual contador incrementar com base na nota.

Quando o fluxo de controle alcança o `switch`, o programa avalia a expressão nos parênteses (`grade / 10`) que se segue à palavra-chave `switch`. Essa é a **expressão de controle** da instrução `switch`. O programa compara o valor da expressão controladora (que deve ser avaliada como um valor integral do tipo `byte`, `char`, `short` ou `int`) com cada rótulo `case`. A expressão controladora na linha 72 realiza a divisão de inteiro, que trunca a parte fracionária do resultado. Portanto, ao dividirmos um valor de 0 a 100 por 10, o resultado é sempre um valor de 0 a 10. Utilizamos vários desses valores em nossos rótulos `case`. Por exemplo, se o usuário inserir o inteiro `85`, a expressão controladora é avaliada como 8. A `switch` compara o `8` com cada rótulo `case`. Se ocorrer uma correspondência (`case 8:` na linha 79), o programa executa essas instruções de `case`. Para o inteiro `8`, a linha 80 incrementa `bCount`, porque uma nota na dezena 80 é um B. A **instrução `break`** (linha 81) faz com que o controle de programa prossiga para a primeira instrução depois da `switch` — nesse programa, alcançamos o fim do corpo do método `incrementLetterGradeCounter`, então o método finaliza e o controle retorna à linha 65 do método `inputGrades` (a primeira linha depois da chamada para `incrementLetterGradeCounter`). A linha 65 é o final de um corpo do loop `while`, portanto o controle flui até a condição de `while` (linha 57) a fim de determinar se o loop deve continuar executando.

Os `case`s no nosso `switch` testam explicitamente os valores `10`, `9`, `8`, `7` e `6`. Observe os casos nas linhas 74–75 que testam os valores `9` e `10` (ambos os quais representam a nota A). Listar os casos consecutivamente dessa maneira sem instruções entre eles permite que os casos executem o mesmo conjunto de instruções — quando a expressão controladora é avaliada como `9` ou `10`, as instruções nas linhas 76–77 serão executadas. A instrução `switch` não fornece um mecanismo para testar séries de valores, então todo valor que você precisa testar deve ser listado em um rótulo `case` separado. Observe que cada `case` pode ter múltiplas instruções. A instrução `switch` difere de outras instruções de controle porque não exige que as múltiplas instruções em um `case` estejam entre chaves.

Sem as instruções `break`, toda vez que ocorre uma correspondência nas instruções `switch`, as instruções para esse caso e os casos subsequentes são executadas até que uma instrução `break` ou o fim do `switch` seja encontrado. Isso costuma ser referido como "falling through", que é o processo em que a instrução percorre sucessivamente os `case`s subsequentes. (Esse recurso é perfeito para escrever um programa conciso que exibe a canção iterativa "The Twelve Days of Christmas" no Exercício 5.29.)

Erro comum de programação 5.7

Esquecer uma instrução `break` quando esta for necessária em um `switch` é um erro de lógica.

Se o valor da expressão controladora não coincidir com nenhum rótulo `case`, o caso `default` (linhas 91–93) é executado. Utilizamos o caso `default` nesse exemplo para processar todos os valores da expressão controladora que são menores que `6` — isto é, todas as notas que reprovam. Se não ocorrer nenhuma correspondência e o `switch` não contiver um caso `default`, o controle de programa simplesmente continua com a primeira instrução depois do `switch`.

A classe `GradeBookTest` que demonstra a classe `GradeBook`

A classe `GradeBookTest` (Figura 5.10) cria um objeto `GradeBook` (linhas 10–11). A linha 13 invoca o método `displayMessage` do objeto para gerar a saída de uma mensagem de boas-vindas para o usuário. A linha 14 invoca o método `inputGrades` do objeto para ler um conjunto de notas do usuário e monitorar a soma de todas as notas inseridas e o número de notas. Lembre-se de que o método `inputGrades` também chama o método `incrementLetterGradeCounter` para monitorar o número de alunos que recebeu cada nota baseada em letra. A linha 15 invoca o método `displayGradeReport` da classe `GradeBook`, que gera a saída de um relatório baseado nos notas inseridas (como na janela entrada/saída na Figura 5.10). A linha 103 da classe `GradeBook` (Figura 5.9) determina se o usuário inseriu pelo menos uma nota — isso ajuda a evitar a divisão por zero. Se tiver inserido, a linha 106 calcula a média das notas. As linhas 109–118 então geram a saída do total de todas as notas, a média da classe e o número de alunos que recebeu cada nota de letra. Se nenhuma nota foi inserida, a linha 121 gera a saída de uma mensagem apropriada. A saída na Figura 5.10 mostra um relatório de nota de exemplo baseado em 10 notas.

```
1  // Figura 5.10: GradeBookTest.java
2  // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // cria o objeto myGradeBookda classe GradeBook e
```

```
9          // passa o nome de cursor para o construtor
10         GradeBook myGradeBook = new GradeBook(
11            "CS101 Introduction to Java Programming" );
12
13         myGradeBook.displayMessage(); // exibe a mensagem welcome
14         myGradeBook.inputGrades(); // lê notas fornecidas pelo usuário
15         myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
16      } // fim de main
17   } // fim da classe GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z
Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Figura 5.10 | `GradeBookTest` cria um objeto `GradeBook` e invoca seus métodos.

Observe que a classe `GradeBookTest` (Figura 5.10) não chama diretamente o método `incrementLetterGradeCounter` `GradeBook` (linhas 69–95 da Figura 5.9). Esse método é utilizado exclusivamente pelo método `inputGrades` de classe `GradeBook` para atualizar o contador apropriado de notas baseadas em letras à medida que cada nota nova é inserida pelo usuário. O método `incrementLetterGradeCounter` existe apenas para suportar as operações dos outros métodos `GradeBook`, portanto ele é declarado `private`.

Observação de engenharia de software 5.2

Com base no Capítulo 3, lembre-se de que os métodos declarados com o modificador de acesso `private` só podem ser chamados por outros métodos da classe em que os métodos `private` são declarados. Esses métodos são comumente chamados de ***métodos utilitários*** *ou* ***métodos auxiliares*** *porque eles são tipicamente utilizados para suportar a operação dos outros métodos da classe.*

Diagrama de atividades UML da instrução `switch`

A Figura 5.11 mostra o diagrama de atividades UML para a instrução `switch` geral. A maioria das instruções `switch` utiliza uma `break` em cada `case` para terminar a instrução `switch` depois de processar um `case`. A Figura 5.11 enfatiza isso incluindo instruções `break` no diagrama de atividades. O diagrama torna claro que a instrução `break` no fim de um `case` faz com o controle saia imediatamente da instrução `switch`.

A instrução `break` não é necessária para o último `case` da `switch` (ou o caso `default` opcional, quando ele aparece por último), porque a execução continua com a próxima instrução depois da `switch`.

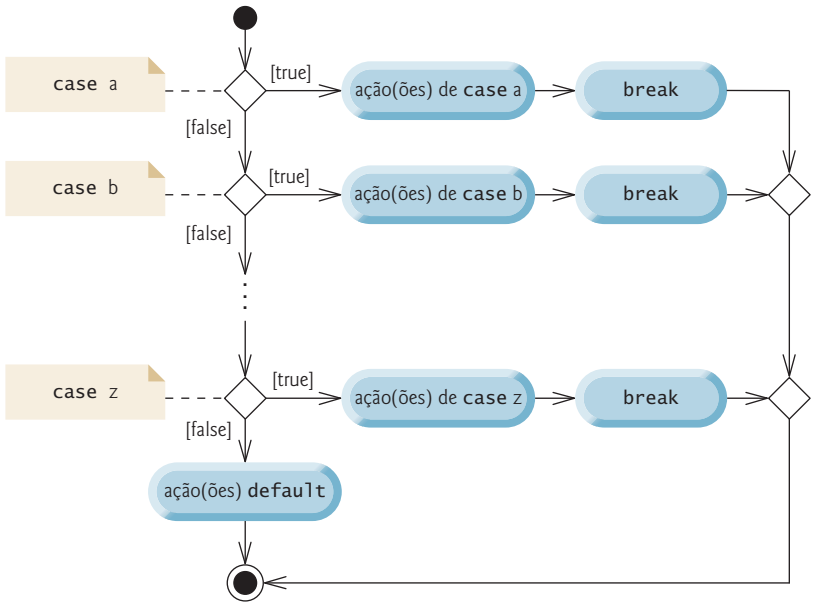


Figura 5.11 | Diagrama de atividades UML de instrução de seleção múltipla `switch` com instruções `break`.

Observação de engenharia de software 5.3

Forneça um case `default` nas instruções `switch`. Incluir um caso `default` faz com que você se concentre na necessidade de processar condições excepcionais.

Boa prática de programação 5.4

Embora cada `case` e o caso `default` em uma `switch` possam ocorrer em qualquer ordem, coloque o caso `default` por último. Quando o caso `default` é listado por último, o `break` para esse caso não é necessário.

Notas sobre a expressão em cada `case` de uma `switch`

Ao utilizar a instrução `switch`, lembre-se de que cada `case` deve conter uma expressão integral constante — isto é, qualquer combinação de constantes integrais que é avaliada como um valor integral constante (por exemplo, –7, 0 ou 221). Uma constante de inteiro é simplesmente um valor de inteiro. Além disso, você pode utilizar **constantes de caractere** — caracteres específicos entre aspas simples, como `'A'`, `'7'` ou `'$'` — que representa os valores integrais de caracteres.

(O Apêndice B mostra os valores integrais dos caracteres no conjunto de caracteres ASCII, que é um subconjunto do conjunto de caracteres Unicode utilizado pelo Java.)

A expressão em cada `case` também pode ser uma **variável constante** — uma variável que contém um valor que não muda no programa inteiro. Essa variável é declarada com a palavra-chave `final` (discutida no Capítulo 6). O Java tem um recurso chamado enumerações, que também apresentamos no Capítulo 6. As constantes de enumeração também podem ser utilizadas em rótulos `case`. No Capítulo 10, "Programação orientada a objetos: polimorfismo", apresentamos uma maneira mais elegante de implementar a lógica `switch` — utilizamos uma técnica chamada polimorfismo para criar programas que são frequentemente mais claros, mais fáceis de manter e mais fáceis de estender do que os programas que utilizam a lógica `switch`.

5.7 Instruções `break` e `continue`

Além das instruções de seleção e repetição, o Java fornece as instruções `break` e **`continue`** (apresentadas nesta seção e no Apêndice O) para alterar o fluxo do controle. A seção precedente mostrou como `break` pode ser utilizado para terminar a execução de uma instrução `switch`. Esta seção discute como utilizar `break` em instruções de repetição.

Instrução `break`

A instrução `break`, quando executada em um `while`, `for`, `do...while` ou `switch`, ocasiona a saída imediata dessa instrução. A execução continua com a primeira instrução depois da instrução de controle. Usos comuns da instrução `break` são escapar no começo de um loop ou pular o restante de uma instrução `switch` (como na Figura 5.9). A Figura 5.12 demonstra uma instrução `break` saindo de um `for`.

```
1  // Figura 5.12: BreakTest.java
2  // a instrução break sai de uma instrução for.
3  public class BreakTest
4  {
5     public static void main( String[] args )
6     {
7        int count; // variável de controle também usada depois que o loop termina
8
9        for ( count = 1; count <= 10; count++ ) // faz o loop 10 vezes
10       {
11          if ( count == 5 ) // se contagem for 5,
12             break;         // termina o loop
13
14          System.out.printf( "%d ", count );
15       } // for final
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // fim de main
19 } // fim da classe BreakTest
```

```
1 2 3 4
Broke out of loop at count = 5
```

Figura 5.12 | Instrução `break` saindo de uma instrução `for`.

Quando a instrução `if` aninhada na linha 11 da instrução `for` (linhas 9–15) detecta que `count` é `5`, a instrução `break` na linha 12 é executada. Isso termina a instrução `for` e o programa prossegue para a linha 17 (logo depois da instrução `for`), que exibe uma mensagem que indica o valor da variável de controle quando o loop termina. O loop executa completamente seu corpo somente quatro vezes em vez de 10.

Instrução `continue`

A instrução `continue`, quando executada em um `while`, `for` ou `do...while`, pula as instruções restantes no corpo do loop e prossegue com a próxima iteração do loop. Nas instruções `while` e `do...while`, o programa avalia o teste de continuação do loop imediatamente depois que a instrução `continue` é executada. Em uma instrução `for`, a expressão incremento é executada, então o programa avalia o teste de continuação do loop.

A Figura 5.13 usa `continue` para pular para a instrução na linha 12 quando o `if` aninhado (linha 9) determina que o valor de `count` é `5`. Quando a instrução `continue` executa, o programa continua com o incremento da variável de controle na instrução `for` (linha 7).

```
1  // Figura 5.13: ContinueTest.java
2  // continua instrução que termina iteração de uma instrução for.
3  public class ContinueTest
4  {
5     public static void main( String[] args )
6     {
7        for ( int count = 1; count <= 10; count++ ) // faz o loop 10 vezes
8        {
9           if ( count == 5 ) // se contagem for 5,
10             continue; // pula o código restante no loop
11
12          System.out.printf( "%d ", count );
13       } // for final
14
15       System.out.println( "\nUsed continue to skip printing 5" );
16    } // fim de main
17 } // fim da classe ContinueTest
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Figura 5.13 | Instrução `continue` terminando uma iteração de uma instrução `for`.

Na Seção 5.3, declaramos que `while` poderia ser utilizado na maioria dos casos no lugar de `for`. Esse não é o caso quando a expressão de incremento em `while` segue uma instrução `continue`. Nesse caso, o incremento não executa antes de o programa avaliar a condição de continuação da repetição, então o `while` não é executado da mesma maneira que o `for`.

Observação de engenharia de software 5.4

Alguns programadores acham que `break` e `continue` violam a programação estruturada. Como os mesmos efeitos são alcançáveis com as técnicas de programação estruturada, esses programadores não utilizam `break` ou `continue`.

Observação de engenharia de software 5.5

Há uma tensão entre alcançar engenharia de software de qualidade e alcançar o software de melhor desempenho. Frequentemente, um desses objetivos é alcançado à custa do outro. Para todas as situações, exceto as de desempenho muito alto, aplique a seguinte regra geral: primeiro, faça seu código simples e correto; então, torne-o rápido e pequeno, mas apenas se necessário.

5.8 Operadores lógicos

As instruções `if`, `if...else`, `while`, `do...while` e `for` exigem uma condição para determinar como continuar o fluxo de um programa de controle. Até agora, só estudamos condições simples, como `count <= 10`, `number != sentinelValue` e `total > 1000`. As condições simples são expressas em termos dos operadores relacionais >, <, >= e <= e os operadores de igualdade == e !=, e cada expressão testa apenas uma condição. Para testar múltiplas condições no processo de tomada de uma decisão, realizamos esses testes em instruções separadas ou em instruções `if` ou `if...else` aninhadas. Às vezes, as instruções de controle requerem condições mais complexas para determinar o fluxo de controle de um programa.

Os **operadores lógicos** do Java permitem-lhe formar condições mais complexas combinando condições simples. Os operadores lógicos são && (E condicional), || (OU condicional), & (E lógico booleano), | (OU inclusivo lógico booleano), ^ (OU exclusivo lógico booleano) e ! (NÃO lógico). [*Nota*: os operadores &, | e ^ também são operadores de bits quando eles são aplicados a operandos de números inteiros. Discutiremos os operadores de bits no Apêndice N.]

Operador E condicional (&&)

Suponha que queiramos assegurar em algum ponto de um programa que duas condições sejam, ambas, `true` antes de escolher um certo caminho de execução. Nesse caso, podemos utilizar o operador **&&** (**E condicional**), como segue:

```
if ( gender == FEMALE && age >= 65 )
   ++seniorFemales;
```

Essa instrução `if` contém duas condições simples. A condição `gender == FEMALE` compara a variável `gender` à constante `FEMALE` para determinar se uma pessoa é do sexo feminino. A condição `age >= 65` poderia ser avaliada para determinar se uma pessoa é um idoso. A instrução `if` considera a condição combinada

```
gender == FEMALE && age >= 65
```

que é verdadeira se e somente se ambas as condições simples forem verdadeiras. Nesse caso, o corpo da instrução `if` incrementa `seniorFemales` por 1. Se qualquer uma ou ambas as condições simples forem falsas, o programa pula o incremento. Alguns programadores acham que preceder condição combinada é mais legível quando os parênteses redundantes são adicionados, como em:

```
( gender == FEMALE ) && ( age >= 65 )
```

A tabela na Figura 5.14 resume o operador &&. A tabela mostra todas as quatro possíveis combinações de valores `false` e `true` para *expressão1* e *expressão2*. Essas tabelas são chamadas de **tabelas-verdade**. O Java avalia todas as expressões `false` ou `true` que incluem operadores relacionais, operadores de igualdade ou operadores lógicos.

expressão1	expressão2	expressão1 && expressão2
false	false	false
false	true	false
true	false	false
true	true	true

Figura 5.14 | Tabela-verdade do operador && (E condicional).

Operador OU condicional (||)

Agora suponha que queiramos assegurar que qualquer uma ou ambas as condições sejam verdadeiras antes de escolhermos certo caminho de execução. Nesse caso, utilizamos o operador **||** (**OU condicional**), como no seguinte segmento de programa:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   System.out.println ( "Student grade is A" );
```

Essa instrução também contém duas condições simples. A condição `semesterAverage >= 90` é avaliada para determinar se o aluno merece um A no curso por causa de um desempenho estável por todo o semestre. A condição `finalExam >= 90` é avaliada para determinar se o aluno merece um A no curso por um desempenho destacado no exame final. A instrução `if` então considera a condição combinada

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

e premia o aluno com um A se qualquer uma ou ambas as condições simples forem verdadeiras. A única vez em que a mensagem `"Student grade is A"` *não* é impressa é quando ambas as condições simples forem falsas. A Figura 5.15 é uma tabela-verdade para o operador condicional OU (`||`). O operador `&&` tem uma precedência mais alta do que operador `||`. Ambos os operadores associam-se da esquerda para direta.

expressão1	expressão2	expressão1 || expressão2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 5.15 | Tabela-verdade do operador || (OU condicional).

Avaliação de curto-circuito de condições complexas

As partes de uma expressão contendo os operadores `&&` ou `||` só são avaliadas até que se saiba se a condição é verdadeira ou falsa. Portanto, avaliação da expressão:

```
( gender == FEMALE ) && ( age >= 65 )
```

parar imediatamente se `gender` não for igual a `FEMALE` (isto é, a expressão inteira for `false`) e continua se `gender` *for* igual a `FEMALE` (isto é, a expressão inteira poderia ainda ser `true` se a condição `age >= 65` fosse `true`). Esse recurso das expressões E condicional e OU condicional chama-se **avaliação em curto-circuito**.

Erro comum de programação 5.8

Em expressões que usam o operador `&&`, uma condição — que chamaremos de condição dependente — pode exigir que outra condição seja verdadeira para que a avaliação da condição dependente tenha significado. Nesse caso, a condição dependente deve ser colocada depois da outra condição, para não ocorrer um erro. Por exemplo, na expressão `( i != 0 ) && ( 10 / i == 2 )`, a segunda condição deve aparecer depois da primeira para não ocorrer um erro de divisão por zero.

Operadores lógicos booleanos E (&) e OU inclusivo (|)

Os operadores **lógicos booleanos E (&)** e **OU inclusivo (|)** são idênticos aos operadores `&&` e `||`, exceto que os operadores `&` e `|` *sempre* avaliam seus dois operandos (isto é, eles não realizam a avaliação em curto circuito). Assim, a expressão:

```
( gender == 1 ) & ( age >= 65 )
```

avalia `age >= 65` independente de `gender` ser ou não igual a `1`. Isso é útil se o operando à direita do operador lógico booleano E ou do operador lógico booleano OU inclusivo tiverem um **efeito colateral** requerido — uma mudança no valor de uma variável. Por exemplo, a expressão:

```
( birthday == true ) | ( ++age >= 65 )
```

garante que a condição `++age >= 65` será avaliada. Portanto, a variável `age` é incrementada, mesmo se a expressão geral for `true` ou `false`.

Dica de prevenção de erro 5.6

Por clareza, evite expressões com efeitos colaterais nas condições. Efeitos colaterais talvez pareçam inteligentes, mas dificultam o entendimento do código e levam a erros de lógica sutis.

OU exclusivo lógico booleano (^)

Uma condição simples que contém o operador **OU exclusivo lógico booleano (^)** é `true` se e somente se um de seus operandos for `true` e o outro for `false`. Se ambos forem `true` ou ambos forem `false`, a condição inteira é `false`. A Figura 5.16 é uma tabela-verdade para o operador OR exclusivo lógico booleano (^). É garantido que esse operador avaliará seus dois operandos.

expressão1	expressão2	expressão1 ^ expressão2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 5.16 | Tabela-verdade do operador ^ (OU exclusivo lógico booleano).

Operador de negação lógica (!)

O operador `!` (**NÃO lógico**, também chamado **negação lógica** ou **complemento lógico**) "inverte" o significado de uma condição. Diferentemente dos operadores lógicos &&, ||, &, | e ^, que são operadores binários que combinam duas condições, o operador de negação lógica é um operador unário que tem apenas uma única condição como um operando. O operador lógico de negação é colocado antes de uma condição para escolher um caminho de execução se a condição original (sem o operador lógico de negação) for `false`, como no segmento de programa:

```
if ( ! ( grade == sentinelValue ) )
   System.out.printf( "The next grade is %d\n", grade );
```

que executa a chamada `printf` somente se `grade` não for igual a `sentinelValue`. Os parênteses em torno da condição `grade == sentinelValue` são necessários, uma vez que o operador lógico de negação tem uma precedência mais alta que o operador de igualdade.

Na maioria dos casos, você pode evitar a utilização da negação lógica expressando a condição diferentemente com um operador relacional ou de igualdade apropriado. Por exemplo, a instrução precedente também pode ser escrita como segue:

```
if ( grade != sentinelValue )
   System.out.printf( "The next grade is %d\n", grade );
```

Essa flexibilidade pode ajudar-lhe a expressar uma condição de uma maneira mais conveniente. A Figura 5.17 é uma tabela-verdade para o operador lógico de negação.

expressão	! expressão
false	true
true	false

Figura 5.17 | Tabela-verdade do operador ! (negação lógica ou NÃO lógico).

Exemplo de operadores lógicos

A Figura 5.18 produz as tabelas-verdade discutidas nesta seção. A saída mostra a expressão `boolean` que foi avaliada e seu resultado. Observe que utilizamos o **especificador de formato %b** para exibir a palavra "true" ou a palavra "false" com base em um valor `boolean` da expressão. As linhas 9–13 produzem a tabela-verdade para &&. As linhas 16–20 produzem a tabela-verdade para ||. As linhas 23–27 produzem a tabela-verdade para &. As linhas 30–35 produzem a tabela-verdade para |. As linhas 38–43 produzem a tabela-verdade para ^. As linhas 46–47 produzem a tabela-verdade para !.

```
1  // Figura 5.18: LogicalOperators.java
2  // Operadores lógicos.
3
4  public class LogicalOperators
5  {
6     public static void main( String[] args )
7     {
8        // cria a tabela-verdade para o operador && (E condicional)
9        System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
10          "Conditional AND (&&)", "false && false", ( false && false ),
11          "false && true", ( false && true ),
12          "true && false", ( true && false ),
13          "true && true", ( true && true ) );
14
15       // cria a tabela-verdade para o operador || (OU condicional)
16       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n\n",
17          "Conditional OR (||)", "false || false", ( false || false ),
18          "false || true", ( false || true ),
19          "true || false", ( true || false ),
20          "true || true", ( true || true ) );
21
22       // cria a tabela-verdade para o operador & (E lógico booleano)
23       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
24          "Boolean logical AND (&)", "false & false", ( false & false ),
25          "false & true", ( false & true ),
26          "true & false", ( true & false ),
27          "true & true", ( true & true ) );
28
29       // cria a tabela-verdade para o operador | (OU inclusivo lógico booleano)
30       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
31          "Boolean logical inclusive OR (|)",
32          "false | false", ( false | false ),
33          "false | true", ( false | true ),
34          "true | false", ( true | false ),
35          "true | true", ( true | true ) );
36
37       // cria a tabela-verdade para o operador ^ (OU lógico booleano exclusivo)
38       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
39          "Boolean logical exclusive OR (^)",
40          "false ^ false", ( false ^ false ),
41          "false ^ true", ( false ^ true ),
42          "true ^ false", ( true ^ false ),
43          "true ^ true", ( true ^ true ) );
44
45       // cria a tabela-verdade para o operador ! (negação lógica)
46       System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
47          "!false", ( !false ), "!true", ( !true ) );
48    } // fim de main
49 } // fim da classe LogicalOperators
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true
```

```
Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Figura 5.18 | Operadores lógicos.

A Figura 5.19 mostra a precedência e a associatividade dos operadores Java que discutimos até agora. Os operadores são apresentados de cima para baixo em ordem decrescente de precedência.

Operadores	Associatividade	Tipo
`++ --`	da direita para a esquerda	unário pós-fixo
`++ -- + - !` (*tipo*)	da direita para a esquerda	unário pré-fixo
`* / %`	da esquerda para a direita	multiplicativo
`+ -`	da esquerda para a direita	aditivo
`< <= > >=`	da esquerda para a direita	relacional
`== !=`	da esquerda para a direita	igualdade
`&`	da esquerda para a direita	E lógico booleano
`^`	da esquerda para a direita	OU lógico booleano exclusivo
`|`	da esquerda para a direita	OU lógico booleano inclusivo
`&&`	da esquerda para a direita	E condicional
`||`	da esquerda para a direita	OU condicional
`?:`	da direita para a esquerda	ternário condicional
`= += -= *= /= %=`	da direita para a esquerda	atribuição

Figura 5.19 | Precedência/associatividade dos operadores discutidos até agora.

5.9 Resumo de programação estruturada

Assim como os arquitetos projetam edifícios empregando o conhecimento coletivo de sua profissão, os programadores projetam programas. Nosso campo é muito mais jovem que a arquitetura e nossa sabedoria coletiva é consideravelmente mais esparsa. Aprendemos que a programação estruturada produz programas que são mais fáceis de entender, testar, depurar, modificar e até demonstrar como corretos em um sentido matemático do que os programas não estruturados.

A Figura 5.20 usa diagramas de atividade UML para resumir instruções de controle do Java. Os estados inicial e final indicam o único ponto de entrada e o único ponto de saída de cada instrução de controle. Conectar símbolos individuais arbitrariamente em um diagrama de atividades pode levar a programas não estruturados. Portanto, costuma-se utilizar um conjunto limitado de instruções de controle que só pode ser combinado de duas maneiras simples para criar programas estruturados.

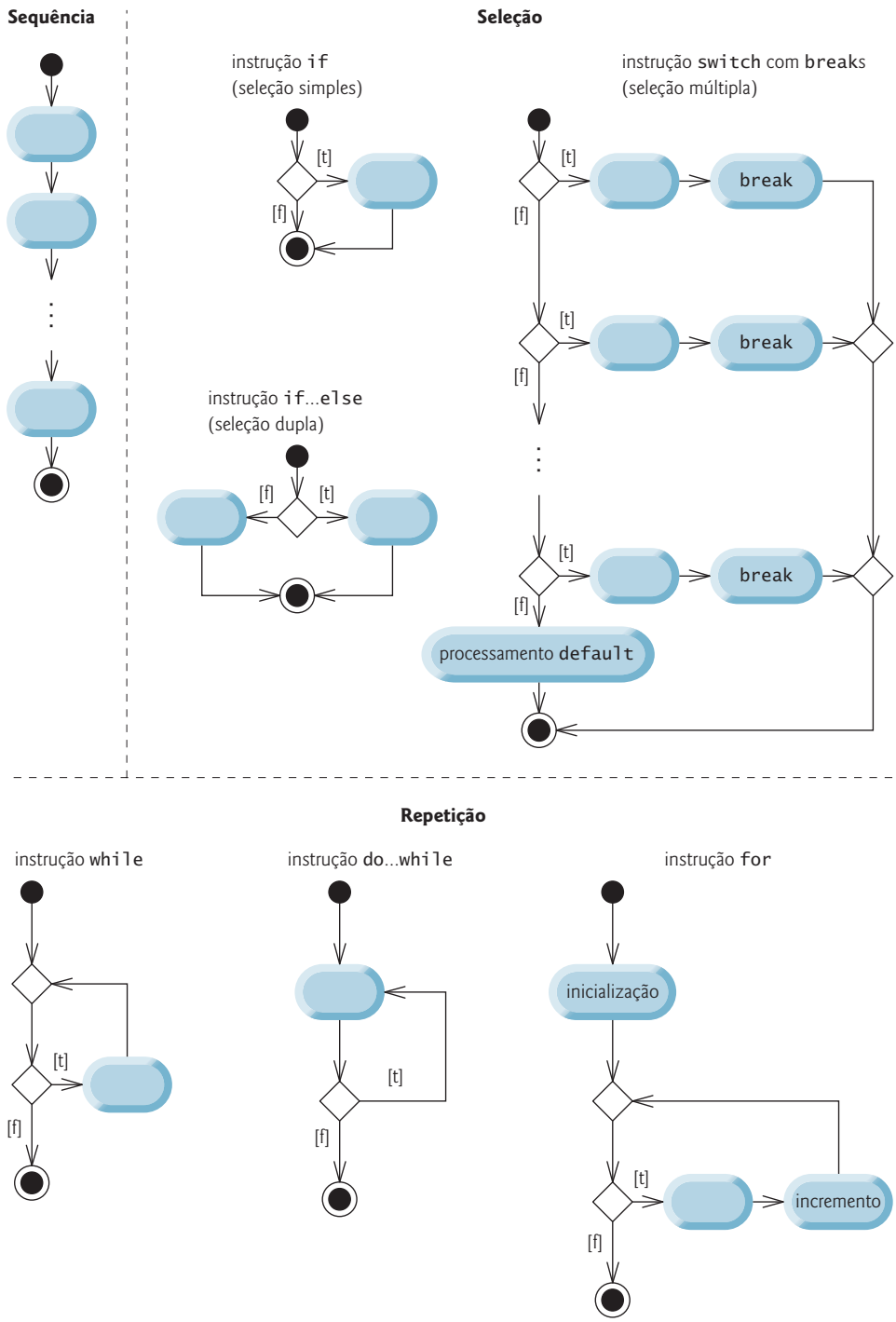


Figura 5.20 | As instruções de sequência de entrada única/saída única, seleção e repetição do Java.

Por simplicidade, o Java inclui apenas instruções de controle de entrada única/saída única — há só uma maneira de entrar e só uma de sair de cada instrução de controle. É simples conectar instruções de controle em sequência para formar programas estruturados. O estado final de uma instrução de controle é conectado ao inicial da próxima — isto é, as instruções de controle são colocadas uma depois da outra em um programa em sequência. Chamamos isso de **empilhamento de instruções de controle**. As regras para formar programas estruturados também permitem que instruções de controle sejam aninhadas.

A Figura 5.21 mostra as regras para formar programas estruturados. As regras assumem que os estados de ação podem ser utilizados para indicar qualquer ação. As regras também assumem que iniciamos com o diagrama de atividades simples (Figura 5.22) consistindo somente um estado inicial, um estado de ação, um estado final e setas de transição.

Regras para formar programas estruturados

1. Comece com o diagrama de atividades mais simples (Figura 5.22).
2. Qualquer estado de ação pode ser substituído por dois estados de ação em sequência.
3. Qualquer estado de ação pode ser substituído por qualquer instrução de controle (sequência de estados de ação, `if`, `if...else`, `switch`, `while`, `do...while` ou `for`).
4. As regras 2 e 3 podem ser aplicadas com a frequência que você quiser em qualquer ordem.

Figura 5.21 | As regras para formar programas estruturados.

Figura 5.22 | Diagrama de atividades mais simples.

A aplicação das regras mostradas na Figura 5.21 sempre resulta em um diagrama de atividades adequadamente estruturado com uma apresentação elegante de blocos de construção. Por exemplo, aplicar a regra 2 repetidamente ao diagrama de atividades mais simples resulta em um diagrama de atividades que contém muitos estados de ação em sequência (Figura 5.23). A regra 2 gera uma pilha de instruções de controle, razão pela qual vamos chamá-la de **regra de empilhamento**. [*Nota*: as linhas tracejadas verticais na Figura 5.23 não fazem parte da UML. Utilizamos essas linhas para separar os quatro diagramas de atividade que demonstram a regra 2 da Figura 5.21 sendo aplicada].

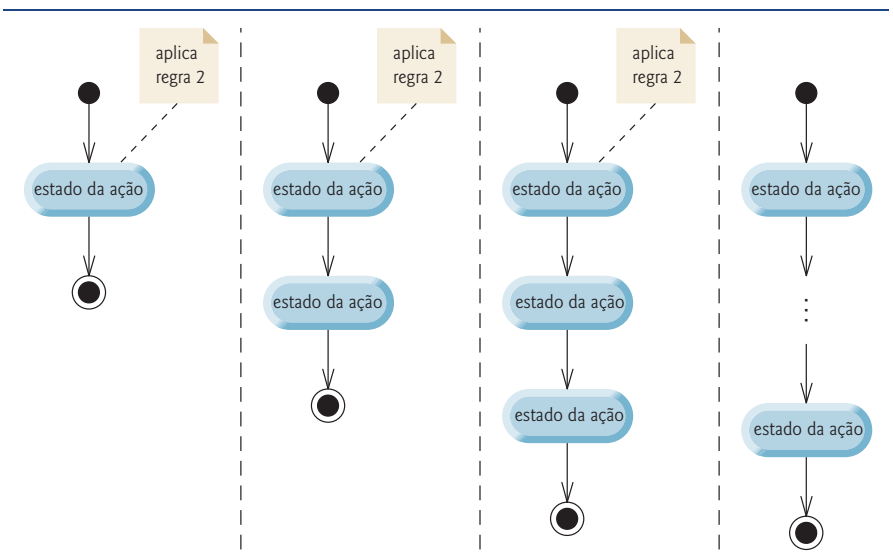


Figura 5.23 | Aplicando a regra de empilhamento repetidamente (regra 2) da Figura 5.21 ao diagrama de atividades mais simples.

A regra 3 é chamada **regra de aninhamento**. Aplicar a regra 3 repetidamente ao diagrama de atividades mais simples resulta em um diagrama de atividades com instruções de controle organizadamente aninhadas. Por exemplo, na Figura 5.24, o estado de ação no diagrama de atividades mais simples é substituído por uma de instrução (`if...else`) de seleção dupla.

Então, a regra 3 é aplicada novamente aos estados de ação na instrução de seleção dupla, substituindo cada um por uma instrução de seleção dupla. O símbolo de estado de ação tracejado em torno de cada instrução de seleção dupla representa o estado de ação que foi substituído. [*Nota*: os símbolos de setas tracejadas e de estados de ação tracejados mostrados na Figura 5.24 não fazem parte da UML. Esses são utilizados aqui para ilustrar que qualquer estado de ação pode ser substituído por uma instrução de controle].

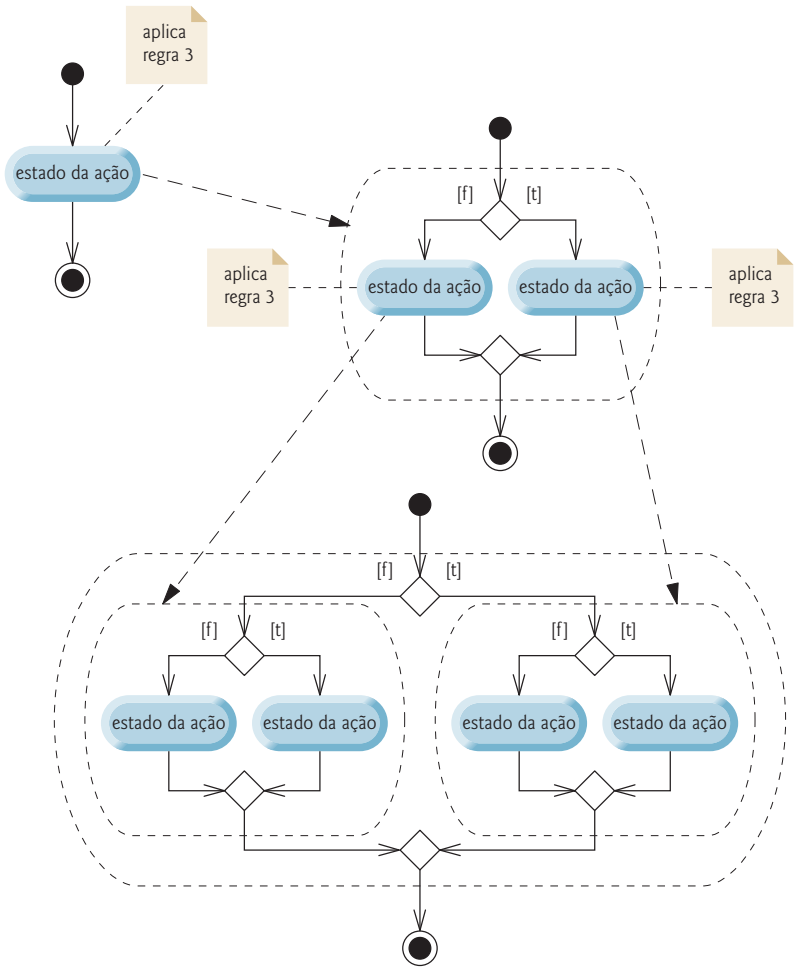


Figura 5.24 | Aplicando a regra de aninhamento repetidamente (regra 3) da Figura 5.21 ao diagrama de atividades mais simples.

A regra 4 gera estruturas maiores, mais complexas e mais profundamente aninhadas. Os diagramas que emergem da aplicação das regras na Figura 5.21 constituem o conjunto de todos os possíveis diagramas de atividade estruturados e, portanto, o conjunto de todos os programas estruturados possíveis. A beleza da abordagem estruturada é que utilizamos apenas sete instruções simples de entrada única/saída única e os montamos de apenas duas maneiras simples.

Se as regras na Figura 5.21 forem seguidas, um diagrama de atividades "não estruturado" (como o da Figura 5.25) não pode ser criado. Se você não tiver certeza se um diagrama particular é estruturado, aplique as regras da Figura 5.21 na ordem inversa para reduzir o diagrama ao diagrama de atividades mais simples. Se puder reduzi-lo, o diagrama original é estruturado; caso contrário, não.

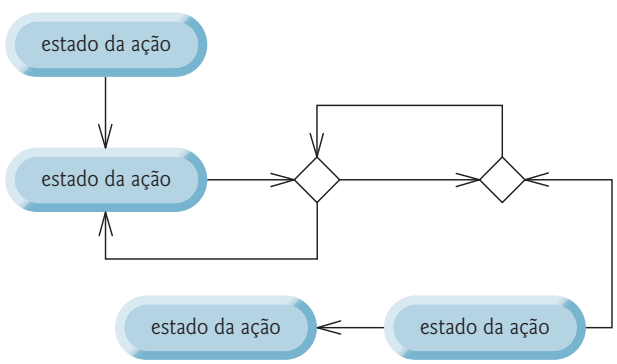


Figura 5.25 | Diagrama de atividades "não estruturado".

Programação estruturada promove a simplicidade. Bohm e Jacopini nos deram o resultado de que apenas três formas de controle são necessárias para implementar um algoritmo:

- sequência;
- seleção;
- repetição.

A estrutura de sequência é trivial. Liste simplesmente as instruções para executar na ordem em que elas devem executar. A seleção é implementada de uma destas três maneiras:

- instrução `if` (seleção única);
- instrução `if...else` (seleção dupla);
- instrução `switch` (seleção múltipla).

De fato, é fácil provar que uma instrução `if` simples é suficiente para fornecer qualquer forma de seleção — tudo que pode ser feito com a instrução `if...else` e a instrução `switch` pode ser implementado combinando-se instruções `if` (embora talvez não de modo tão claro e eficiente).

A repetição é implementada de uma destas três maneiras:

- instrução `while`;
- instrução `do...while`;
- instrução `for`.

[*Nota*: há uma quarta instrução de repetição — a instrução `for` aprimorada — que discutiremos na Seção 7.6.] É simples provar que a instrução `while` é suficiente para fornecer qualquer forma de repetição. Tudo o que pode ser feito com `do...while` e `for` pode ser feito com `while` (embora talvez de maneira não tão conveniente).

A combinação desses resultados ilustra que qualquer forma de controle que possa ser necessária um dia em um programa Java pode ser expressa em termos de:

- sequência;
- instrução `if` (seleção);
- instrução `while` (repetição).

e que podem ser combinadas apenas de duas maneiras — empilhamento e aninhamento. De fato, a programação estruturada é a essência da simplicidade.

5.10 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando retângulos e ovais

Esta seção demonstra como desenhar retângulos e ovais, utilizando os métodos `Graphics` **`drawRect`** e **`drawOval`**, respectivamente. Esses métodos são demonstrados na Figura 5.26.

A linha 6 inicia a declaração de classe para `Shapes`, que estende `JPanel`. A variável de instância, `choice`, declarada na linha 8, determina se `paintComponent` deve desenhar retângulos ou ovais. O construtor `Shapes` nas linhas 11–14 inicializa `choice` com o valor passado no parâmetro `userChoice`.

O método `paintComponent` (linhas 17–36) realiza o desenho real. Lembre-se, a primeira instrução em cada método `paintComponent` deve ser uma chamada a `super.paintComponent`, como na linha 19. As linhas 21–35 repetem-se 10 vezes para desenhar 10 formas. A instrução `switch` aninhada (linhas 24–34) escolhe entre desenhar retângulos e desenhar ovais.

Se `choice` for 1, então o programa desenha retângulos. As linhas 27–28 chamam o método `Graphics` `drawRect`. O método `drawRect` requer quatro argumentos. As duas primeiras representam as *coordenadas x* e *y* do canto superior esquerdo do retângulo; as duas seguintes representam a largura e altura do retângulo. Nesse exemplo, iniciamos em uma posição de 10 pixels para baixo e 10 pixels à direita do canto superior esquerdo e cada iteração do loop move o canto superior esquerdo outros 10 pixels para baixo e para a direita. A largura e a altura do retângulo iniciam a 50 pixels e aumentam 10 pixels a cada iteração.

Se `choice` for 2, o programa desenha ovais. Ele cria um retângulo imaginário chamado **retângulo delimitador** e posiciona dentro dele uma oval que toca os pontos centrais dos quatro lados. O método `drawOval` (linhas 31–32) requer os mesmos quatro argumentos como método `drawRect`. Os argumentos especificam a posição e o tamanho do retângulo para a elipse. Os valores passados para `drawOval` nesse exemplo são exatamente os mesmos que aqueles passados para `drawRect` nas linhas 27–28. Visto que a largura e a altura do retângulo delimitador são idênticas nesse exemplo, as linhas 27–28 desenham um círculo. Como exercício, tente modificar o programa para desenhar tanto retângulos como elipses para ver como `draw-Oval` e `drawRect` estão relacionados.

```
1  // Figura 5.26: Shapes.java
2  // Demonstra o desenho de diferentes formas.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Shapes extends JPanel
7  {
8     private int choice; // escolha do usuário de qual forma desenhar
9
10    // construtor configura a escolha do usuário
11    public Shapes( int userChoice )
12    {
13       choice = userChoice;
14    } // fim do construtor Shapes
15
16    // desenha uma cascata de formas que iniciam do canto superior esquerdo
17    public void paintComponent( Graphics g )
18    {
19       super.paintComponent( g );
20
21       for ( int i = 0; i < 10; i++ )
22       {
23          // seleciona a forma com base na escolha do usuário
24          switch ( choice )
25          {
26             case 1: // desenha retângulos
27                g.drawRect( 10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10 );
29                break;
30             case 2: // desenha elipses
31                g.drawOval( 10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10 );
33                break;
34          } // fim do switch
35       } // for final
36    } // fim do método paintComponent
37 } // fim da classe Shapes
```

Figura 5.26 | Desenhando uma cascata de formas com base na escolha do usuário.

Classe ShapesTest

A Figura 5.27 é responsável por tratar a entrada do usuário e criar uma janela para exibir o desenho adequado com base na resposta do usuário. A linha 3 importa `JFrame` para tratar a exibição e a linha 4 importa `JOptionPane` para tratar a entrada.

As linhas 11–13 exibem um prompt para o usuário na forma de um diálogo de entrada e armazenam a resposta do usuário na variável `input`. A linha 15 utiliza o método `Integer` `parseInt` para converter a `String` inserida pelo usuário em um `int` e armazenar o resulta na variável `choice`. A linha 18 cria um objeto `Shapes`, com a escolha do usuário passada como um argumento para o construtor. As linhas 20–25 realizam as operações padrão que criam e configuram uma janela nesse estudo de caso — criam um quadro, configuram-no para encerrar o aplicativo quando fechado, adicionam o desenho ao quadro, configuram o tamanho do quadro e o tornam visível.

```
1  // Figura 5.27: ShapesTest.java
2  // Aplicativo de teste que exibe a classe Shapes.
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  public class ShapesTest
7  {
8     public static void main( String[] args )
9     {
10       // obtém a escolha do usuário
11       String input = JOptionPane.showInputDialog(
12          "Enter 1 to draw rectangles\n" +
13          "Enter 2 to draw ovals" );
14
15       int choice = Integer.parseInt( input ); // converte a entrada em int
16
17       // cria o painel com a entrada do usuário
18       Shapes panel = new Shapes( choice );
```

```
19
20          JFrame application = new JFrame(); // cria um novo JFrame
21
22          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
23          application.add( panel ); // adiciona o painel ao frame
24          application.setSize( 300, 300 ); // configura o tamanho desejado
25          application.setVisible( true ); // mostra o frame
26       } // fim de main
27    } // fim da classe ShapesTest
```

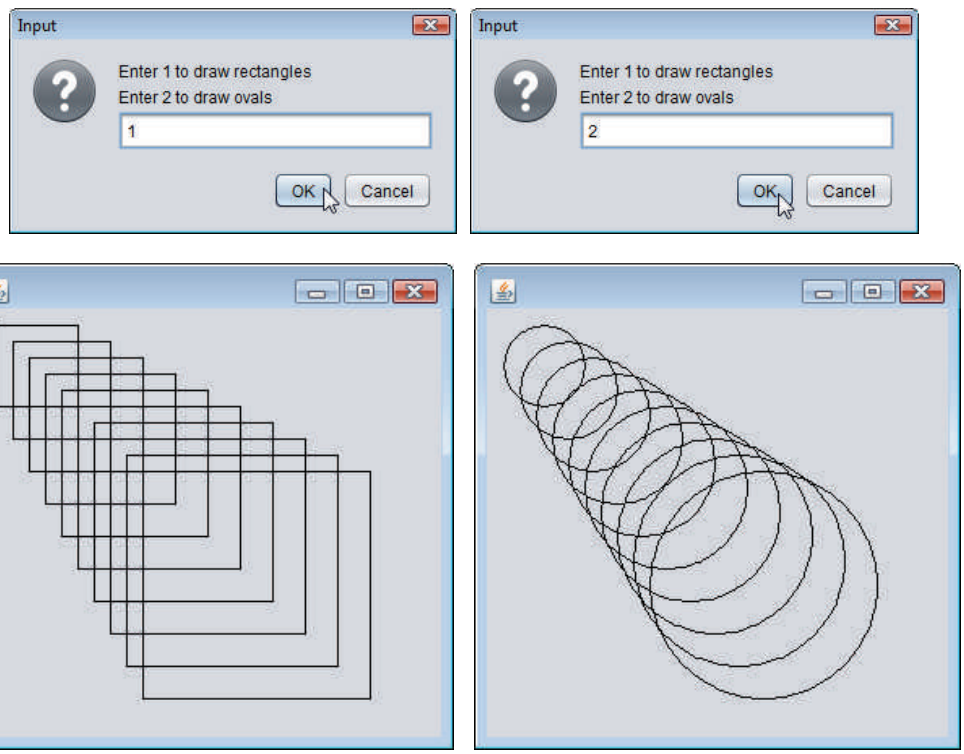


Figura 5.27 | Obtendo a entrada de usuário e criando um `JFrame` para exibir `Shapes`.

Exercícios do Estudo de caso sobre GUIs e imagens gráficas

5.1 Desenhe 12 círculos concêntricos no centro de um `JPanel` (Figura 5.28). O círculo mais interno deve ter um raio de 10 pixels e cada círculo sucessivo deve ter um raio 10 pixels maior que o anterior. Comece localizando o centro do `JPanel`. Para obter o canto superior esquerdo de um círculo, mova-se um raio para cima e um raio para a esquerda a partir do centro. A largura e a altura do retângulo delimitador têm o mesmo diâmetro do círculo (isto é, duas vezes o raio).

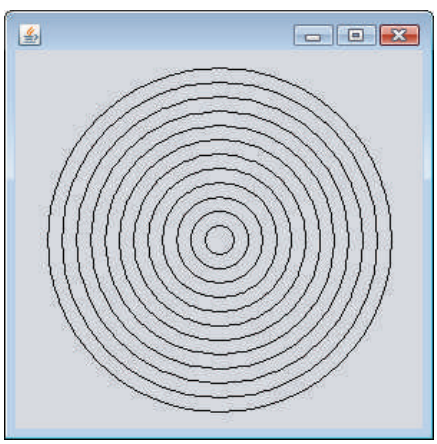


Figura 5.28 | Desenhando círculos concêntricos.

5.2 Modifique o Exercício 5.16 no final dos exercícios do capítulo para ler a entrada utilizando diálogos e exibir o gráfico de barras utilizando retângulos de comprimentos variados.

5.11 Conclusão

Neste capítulo, completamos nossa introdução às instruções de controle do Java, que permitem-lhe controlar o fluxo de execução em métodos. O Capítulo 4 discutiu as instruções `if`, `if...else` e `while` do Java. O capítulo atual demonstrou as instruções de controle do Java restantes — `for`, `do...while` e `switch`. Mostramos que qualquer algoritmo pode ser desenvolvido utilizando combinações da estrutura de sequência (isto é, instruções listadas na ordem em que devem ser executadas), os três tipos de instruções de seleção — `if`, `if...else` e `switch` — e os três tipos de instruções de repetição — `while`, `do...while` e `for`. Neste capítulo e no Capítulo 4, discutimos como você pode combinar esses blocos de construção para utilizar as comprovadas técnicas de construção de programa e solução de problemas. Este capítulo também introduziu operadores lógicos do Java, que permitem-lhe utilizar expressões condicionais mais complexas em instruções de controle.

No Capítulo 3, introduzimos os conceitos básicos de objetos, classes e métodos. O Capítulo 4 e o Capítulo 5 introduziram os tipos de instruções de controle que você pode utilizar para especificar a lógica do programa nos métodos. No Capítulo 6, examinamos os métodos em maior profundidade.

Resumo

Seção 5.2 Princípios básicos de repetição controlada por contador

- A repetição controlada por contador requer uma variável de controle (ou contador de loops), o valor inicial da variável de controle, o incremento (ou decremento) pelo qual a variável de controle é modificada a cada passagem pelo loop (também conhecido como cada iteração do loop) e a condição de continuação do loop que determina se um loop deve continuar.
- Você pode declarar uma variável e inicializá-la na mesma instrução.

Seção 5.3 Instrução de repetição `for`

- A instrução `while` pode ser utilizada para implementar qualquer loop controlado por contador.
- A instrução `for` especifica os detalhes da repetição controlada por contador em uma única linha do código.
- Quando a instrução `for` começa a executar, sua variável de controle é declarada e inicializada. Em seguida, o programa verifica a condição de continuação do loop. Se inicialmente a condição for verdadeira, o corpo será executado. Depois de executar o corpo do loop, a expressão de incremento é executada. Então o teste de continuação do loop é realizado novamente para determinar se o programa deve continuar com a próxima iteração do loop.
- O formato geral da instrução `for` é:

```
for ( inicialização; condiçãoDeContinuaçãoDoLoop; incremento )
   instrução
```

  onde a expressão inicialização nomeia a variável de controle do loop e fornece seu valor inicial, *loopContinuationCondition* é a condição que determina se o loop deve continuar executando e incremento modifica o valor da variável de controle, para que a condição de continuação do loop por fim torne-se falsa. Os dois ponto-e-vírgulas no cabeçalho `for` são necessários.
- A maioria das instruções `for` pode ser representada com instruções `while` equivalentes como a seguir:

```
inicialização;
while ( condiçãoDeContinuaçãoDoLoop )
{
   instrução
   incremento;
}
```

- Em geral, as instruções `for` são utilizadas para repetição controlada por contador e as instruções `while` são utilizadas para repetição controlada por sentinela.
- Se a expressão de *inicialização* no cabeçalho `for` declarar a variável de controle, a variável de controle só poderá ser utilizada nessa instrução `for` — ela não existirá fora da instrução `for`.
- As expressões em um cabeçalho `for` são opcionais. Se a *condiçãoDeContinuaçãoDoLoop* for omitida, o Java irá supor que ela sempre é verdadeira, criando assim um loop infinito. Você poderia omitir a expressão *inicialização* se a variável de controle for inicializada antes do loop. Você poderia omitir a expressão incremento se o incremento fosse calculado com instruções no corpo do loop ou se nenhum incremento fosse necessário.
- A expressão incremento em uma instrução `for` atua como se ela fosse uma instrução independente no fim do corpo de `for`.
- Uma instrução `for` pode contar para baixo utilizando um incremento negativo (isto é, um decremento).
- Se a condição de continuação do loop for inicialmente `false`, o programa não executará o corpo da instrução `for`. Em vez disso, a execução prossegue com a instrução seguinte ao `for`.

Seção 5.4 Exemplos com a estrutura `for`

- O Java trata as constantes de ponto flutuante como `1000.0` e `0.05` como tipo `double`. De maneira semelhante, o Java trata as constantes de número inteiro como `7` e `-22` como tipo `int`.
- O especificador de formato `%4s` gera saída para uma `String` em um tamanho de campo de 4 — isto é, `printf` exibe o valor com pelo menos 4 posições de caractere. Se o valor a ser enviado para a saída for menor do que a largura de 4 posições de caractere, o valor é alinhado à direita no campo por padrão. Se a largura tiver um valor maior do que 4 posições de caractere, o tamanho do campo é expandido para acomodar o número apropriado de caracteres. Para justificar o valor à esquerda, utilize um número inteiro negativo para especificar o tamanho do campo.
- `Math.pow(`*x*`, `*y*`)` calcula o valor de *x* elevado à *y*-ésima potência. O método recebe dois argumentos `double` e retorna um valor `double`.
- O flag de formatação vírgula (`,`) em um especificador de formato (por exemplo, `%,20.2f`) indica que um valor de ponto flutuante deve ser enviado para a saída com um separador de agrupamento. O separador real utilizado é específico à localidade do usuário (isto é, país). Nos Estados Unidos, o número terá vírgulas que separam cada três dígitos e um ponto decimal que separa a parte fracionária do número, como em 1,234.45.
- O `.` em um especificador de formato (por exemplo, `%,20.2f`) indica que o número inteiro à direita é a precisão do número.

Seção 5.5 Instrução de repetição `do...while`

- A instrução `do...while` é semelhante à instrução `while`. No `while`, o programa testa a condição de continuação do loop no início do loop, antes de executar seu corpo; se a condição for falsa, o corpo nunca será executado. A instrução `do...while` testa a condição de continuação do loop depois de executar o corpo do loop; portanto, o corpo sempre executa pelo menos uma vez.
- Não é necessário utilizar chaves na instrução de repetição `do...while` se houver apenas uma instrução no corpo. A maioria dos programadores inclui as chaves, para evitar confusão entre as instruções `while` e `do...while`.

Seção 5.6 A estrutura de seleção múltipla `switch`

- A instrução `switch` realiza ações diferentes com base nos possíveis valores de uma expressão inteira constante (isto é, um valor constante do tipo `byte`, `short`, `int` ou `char`, mas não `long`).
- O indicador de fim de arquivo é uma combinação de pressionamento de tecla dependente do sistema que termina a entrada de usuário. Nos sistemas UNIX/Linux/Mac OS X, o fim de arquivo é inserido digitando a sequência *<Ctrl> d* em uma linha separada. Essa notação significa pressionar simultaneamente a tecla Ctrl e a tecla d. Nos sistemas Windows, insira o fim de arquivo digitando *<Ctrl> z*.
- O método `Scanner` `hasNext` determina se há mais dados a inserir. Esse método retorna o valor `boolean` `true` se houver mais dados; do contrário, ele retorna `false`. Enquanto o indicador de fim do arquivo não tiver sido digitado, o método `hasNext` retornará `true`.
- A instrução `switch` consiste em um bloco que contém uma sequência de rótulos `case` e um caso `default` opcional.
- Quando o fluxo do controle alcança um `switch`, o programa avalia a expressão de controle do `switch` e compara seu valor com cada rótulo `case`. Se ocorrer uma correspondência, o programa executará as instruções para esse `case`.
- Listar casos consecutivamente sem instruções entre eles permite aos casos executar o mesmo conjunto de instruções.
- Cada valor que você deseja testar em um `switch` deve ser listado em um rótulo `case` separado.
- Cada `case` pode ter múltiplas instruções, e estas não devem ser colocadas entre colchetes.
- Sem as instruções `break`, toda vez que ocorre uma correspondência nas instruções `switch`, as instruções para esse caso e casos subsequentes são executadas até que uma instrução `break` ou o fim do `switch` seja encontrado.
- Se não ocorrer nenhuma correspondência entre o valor da expressão controladora e um rótulo `case`, o caso `default` opcional é executado. Se não ocorrer nenhuma correspondência e o `switch` não contiver um caso `default`, o controle de programa simplesmente continua com a primeira instrução depois do `switch`.

Seção 5.7 Instruções `break` e `continue`

- A instrução `break`, quando executada em um `while`, `for`, `do...while` ou `switch`, ocasiona a saída imediata dessa instrução. A execução continua com a primeira instrução depois da instrução de controle.
- A instrução `continue`, quando executada em um `while`, `for` ou `do...while`, pula as instruções restantes no corpo do loop e prossegue com a próxima iteração do loop. Nas instruções `while` e `do...while`, o programa avalia o teste de continuação do loop imediatamente. Em uma instrução `for`, a expressão incremento é executada, então o programa avalia o teste de continuação do loop.

Seção 5.8 Operadores lógicos

- As condições simples são expressas em termos dos operadores relacionais `>`, `<`, `>=` e `<=` e os operadores de igualdade `==` e `!=`, e cada expressão testa apenas uma condição.
- Os operadores lógicos permitem-lhe formar condições complexas combinando condições simples. Os operadores lógicos são `&&` (E condicional), `||` (OU condicional), `&` (E lógico booleano), `|` (OU inclusivo lógico booleano), `^` (OU exclusivo lógico booleano) e `!` (NÃO lógico).
- Para assegurar que duas condições são verdadeiras, utilize o operador `&&` (AND condicional). Se uma ou as duas condições simples forem falsas, a expressão inteira será falsa.
- Para assegurar que uma das duas *ou* ambas as condições são verdadeiras, utilize o operador `||` (OR condicional), que é avaliado como verdadeiro se uma das ou ambas as condições simples forem verdadeiras.
- Uma condição que usa os operadores `&&` ou `||` utiliza a avaliação em curto-circuito — elas só são avaliadas até que se conheça se a condição é verdadeira ou é falsa.

- Os operadores E lógico booleano (&) e OU inclusivo lógico booleano (|) funcionam de modo idêntico aos operadores && (E condicional) e | (OU condicional), mas sempre avaliam ambos os operandos.
- Uma condição simples que contém o operador OU exclusivo lógico booleano (^) é `true` se e somente se um de seus operandos for `true` e o outro for `true`. Se os dois operandos forem `true` ou ambos forem `false`, a condição inteira é `false`. Também é garantido que esse operador avaliará seus dois operandos.
- O operador unário ! (NÃO lógico) "inverte" o valor de uma condição.

Terminologia

^, OU lógico booleano exclusivo, operador, 138
-, (sinal de subtração), flag de formatação, 126
„ (vírgula), flag de formatação, 127
&&, E condicional, operador, 136
&, E lógico booleano, operador, 137
%b, especificador de formato, 138
||, OU condicional, operador, 137
|, OU inclusivo lógico booleano, operador, 137
!, NÃO lógico, operador, 138
alinhado à esquerda, 126
`break`, instrução, 132
cabeçalho da instrução de repetição `for`, 122
`case`, palavra-chave, 132
complemento lógico, !, operador, 138
`continue`, instrução, 134
curto-circuito, avaliação, 137
`default`, caso em um `switch`, 132
`do...while`, instrução de repetição, 127
`drawOval`, método da classe `Graphics`, 144
`drawRect`, método da classe `Graphics`, 144
E condicional, &&, operador, 136
efeito colateral, 137
E lógico booleano, &, operador, 137
empilhamento, regra, 142
erro por um (off-by-one), 122
escopo de variável, 123
expressão de controle da instrução `switch`, 132
expressão integral constante, 129
incremento de uma variável de controle, 120
largura de campo, 126
loop, condição de continuação, 120
`Math`, classe, 126
método auxiliar, 133
método utilitário, 133
negação lógica, !, 138
operadores lógicos, 136
OU condicional, ||, operador, 137
OU lógico booleano exclusivo, ^, operador, 138
OU lógico booleano inclusivo, |, operador, 137
regra de aninhamento, 142
retângulo delimitador, 144
rótulo em uma `switch`, 132
saída alinhada à direita, 126
separador de grupos (saída formatada), 127
sinal de subtração (–), flag de formatação, 126
`switch`, instrução de múltipla seleção, 129
tabela-verdade, 136
valor inicial da variável de controle, 120
variável constante, 134
variável de controle, 120
vírgula (,) flag de formatação, 127

Exercícios de autorrevisão

5.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Em geral, as instruções _____ são utilizadas para repetição controlada por contador e as instruções _____ são utilizadas para repetição controlada por sentinela.

b) A instrução `do...while` testa a condição de continuação do loop _____ de executar o corpo do loop; portanto, o corpo sempre executa pelo menos uma vez.

c) A instrução _____ seleciona entre múltiplas ações com base nos possíveis valores de uma variável ou expressão integrais.

d) A instrução _____, quando executada em uma instrução de repetição, pula as instruções restantes no corpo do loop e prossegue com a próxima iteração do loop.

e) O operador _____ pode ser usado para assegurar que duas condições são *ambas* verdadeiras antes de escolher certo caminho de execução.

f) Se a condição de continuação do loop em um cabeçalho `for` for inicialmente _____, o programa não executará o corpo da instrução `for`.

g) Os métodos que realizam as tarefas comuns e não exigem os objetos são chamados de métodos _____.

5.2 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) O caso `default` é requerido na instrução de seleção `switch`.

b) A instrução `break` é requerida no último caso de uma instrução de seleção `switch`.

c) A expressão `((x> y) && (a <b))` é verdadeira se `x > y` for verdadeiro ou `a <b` for verdadeiro.

d) Uma expressão contendo o operador `||` é verdadeira se um ou ambos de seus operandos forem verdadeiros.

e) O flag de formatação vírgula (`,`) em um especificador de formato (por exemplo, `%,20.2f`) indica que um valor deve ser enviado para a saída com um separador de milhares.

f) Para testar uma série de valores em uma instrução `switch`, utilize um hífen (–) entre os valores inicial e final da série em um rótulo `case`.

g) Listar casos consecutivamente sem instruções entre eles permite aos casos executar o mesmo conjunto de instruções.

5.3 Escreva uma instrução Java ou um conjunto de instruções Java para realizar cada uma das seguintes tarefas:

a) Some os inteiros ímpares entre 1 e 9 utilizando uma instrução `for`. Assuma que as variáveis de inteiro `sum` e `count` foram declaradas.

b) Calcule o valor de `2.5` elevado à potência de `3`, utilizando o método `pow`.

c) Imprima os inteiros de 1 a 20, utilizando um loop `while` e a variável contadora `i`. Assuma que a variável `i` foi declarada mas não foi inicializada. Imprima apenas cinco inteiros por linha. [*Dica:* utilize o cálculo `i % 5`. Quando o valor dessa expressão for 0, imprima um caractere de nova linha; caso contrário, imprima um caractere de tabulação. Assuma que esse código é um aplicativo. Utilize o método `System.out.println()` para imprimir o caractere de nova linha, e utilize o método `System.out.print('\t')` para imprimir o caractere de tabulação.]

d) Repita a parte (c), utilizando uma instrução `for`.

5.4 Localize o erro em cada um dos seguintes segmentos de código e explique como corrigi-los:

a)
```
i = 1;
while ( i <= 10 );
   i++;
}
```

b)
```
for ( k = 0.1; k != 1.0; k += 0.1 )
   System.out.println( k );
```

c)
```
switch ( n )
{
   case 1:
      System.out.println( "The number is 1" );
   case 2:
      System.out.println( "The number is 2" );
      break;
   default:
   System.out.println( "The number is not 1 or 2" );
      break;
}
```

d) O seguinte código deve imprimir os valores 1 a 10:
```
n = 1;
while ( n < 10 )
   System.out.println( n++ );
```

Respostas dos exercícios de autorrevisão

5.1 a) `for`, `while`. b) depois. c) `switch`. d) `continue`. e) `&&` (E condicional). f) `false`. g) `static`.

5.2 a) Falso. O caso `default` é opcional. Se nenhuma ação padrão for necessária, então não há necessidade de um caso `default`. b) Falso. A instrução `break` é utilizada para sair da instrução `switch`. A instrução `break` não é necessária para o último caso em uma instrução `switch`. c) Falso. As duas expressões relacionais devem ser verdadeiras para a expressão inteira ser verdadeira ao utilizar o operador `&&`. d) Verdadeiro. e) Verdadeiro. f) Falso. A instrução `switch` não fornece um mecanismo para testar intervalos de valores, então cada valor que deve ser testado deve ser listado em um rótulo `case` separado. g) Verdadeiro.

5.3 a)
```
sum = 0;

for ( count = 1; count <= 99; count += 2 )

sum += count;
```

b)
```
double result = Math.pow( 2.5, 3 );
```

c)
```
i = 1;

while ( i <= 20 )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );

   ++i;
}
```

d)
```
for ( i = 1; i <= 20; i++ )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );
}
```

5.4 a) Erro: o ponto-e-vírgula depois do cabeçalho `while` causa um loop infinito, e há uma chave esquerda ausente.
Correção: Substitua o ponto-e-vírgula por uma `{` ou remova o `;` e a `}`.

b) Erro: utilizar um número de ponto flutuante para controlar uma instrução `for` pode não funcionar, porque os números de ponto flutuante só são representados aproximadamente pela maioria dos computadores.
Correção: utilize um inteiro e realize o cálculo adequado a fim de obter os valores que você deseja:
```
for ( k = 1; k != 10; k++ )
   System.out.println( (double) k / 10 );
```

c) Erro: O código ausente é a instrução `break` nas instruções para o primeiro `case`.
Correção: Adicione uma instrução `break` ao fim das instruções para o primeiro `case`. Observe que essa omissão não é necessariamente um erro se você quiser que a instrução `case 2:` execute cada vez que a instrução `case 1:` executar.

d) Erro: Um operador relacional inadequado é utilizado na condição de continuação de `while`.
Correção: Utilize <= em vez de < ou altere `10` para `11`.

Exercícios

5.5 Descreva os quatro elementos básicos de repetição controlada por contador.

5.6 Compare e contraste as instruções de repetição `while` e `for`.

5.7 Discuta uma situação em que seria mais adequado utilizar uma instrução `do...while` do que uma instrução `while`. Explique por quê.

5.8 Compare e contraste as instruções `break` e `continue`.

5.9 Localize e corrija o(s) erro(s) em cada um dos seguintes segmentos de código:

a)
```
For ( i = 100, i >= 1, i++ )
   System.out.println( i );
```

b) O seguinte código deve imprimir se o inteiro `value` for par ou ímpar:
```
switch ( value % 2 )
{
   case 0:
      System.out.println( "Even integer" );

   case 1:
      System.out.println( "Odd integer" );
}
```

c) O código a seguir deve dar saída dos inteiros ímpares de 19 a 1:
```
for ( i = 19; i >= 1; i += 2 )
   System.out.println( i );
```

d) O código seguinte deve dar saída dos inteiros pares de 2 a 100:
```
counter = 2;

do
{
   System.out.println( counter );
   counter += 2;
} While ( counter < 100 );
```

5.10 O que o seguinte programa faz?

```
1  // Exercício 5.10: Printing.java
2  public class Printing
3  {
4     public static void main( String[] args )
5     {
6        for ( int i = 1; i <= 10; i++ )
7        {
8           for ( int j = 1; j <= 5; j++ )
9              System.out.print( '@' );
10
11          System.out.println();
12       } // fim do for externo
13    } // fim de main
14 } // fim da classe Printing
```

5.11 (***Localize o menor valor***) Escreva um aplicativo que localize o menor de vários números inteiros. Assuma que o primeiro valor lido especifica o número de valores a serem inseridos pelo usuário.

5.12 (***Calculando o produto de números inteiros ímpares***) Escreva um aplicativo que calcule o produto dos números inteiros ímpares de 1 a 15.

5.13 (***Fatoriais***) Fatoriais costumam ser utilizados em problemas de probabilidade. O fatorial de um inteiro positivo *n* (escrito como *n*! e pronunciado como "fatorial *de n*") é igual ao produto dos números inteiros positivos de 1 a *n*. Escreva um aplicativo que calcule os fatoriais de 1 a 20. Utilize o tipo `long`. Exiba os resultados em formato tabular. Que dificuldade poderia impedir você de calcular o fatorial de 100?

5.14 (***Programa de juros compostos modificado***) Modifique o aplicativo de juros compostos da Figura 5.6 para repetir os passos para taxas de juros de 5 %, 6 %, 7 %, 8 %, 9 % e 10 %. Utilize um loop `for` para variar a taxa de juros.

5.15 (***Programa de imprimir de triângulos***) Escreva um aplicativo que exibe os seguintes padrões separadamente, um embaixo do outro. Utilize loops `for` para gerar os padrões. Todos os asteriscos (*) devem ser impressos por uma única instrução na forma `System.out.print( '*' );` o que faz com que os asteriscos sejam impressos lado a lado. Uma instrução na forma `System.out.println();` pode ser utilizada para mover-se para a próxima linha. Uma instrução na forma `System.out.print( ' ' );` pode ser utilizada para exibir um espaço para os últimos dois

padrões. Não deve haver outras instruções de saída no programa. [*Dica:* os dois últimos padrões requerem que cada linha inicie com um número adequado de espaços em branco].

```
(a)         (b)                (c)         (d)
*           **********  **********          *
**          *********    *********         **
***         ********      ********        ***
****        *******        *******       ****
*****       ******          ******      *****
******      *****            *****     ******
*******     ****              ****    *******
********    ***                ***   ********
*********   **                  **  *********
**********  *                    * **********
```

5.16 **(*Programa de impressão de gráfico de barras*)** Uma aplicação interessante dos computadores é exibir diagramas e gráficos de barras. Escreva um aplicativo que leia cinco números entre 1 e 30. Para cada número que é lido, seu programa deve exibir o mesmo número de asteriscos adjacentes. Por exemplo, se seu programa lê o número 7, ele deve exibir `*******`. Exiba as barras dos asteriscos *depois* de ler os cinco números.

5.17 **(*Calculando vendas*)** Um varejista online vende cinco produtos cujos preços no varejo são como a seguir: produto 1, US$ 2,98; produto 2, US$ 4,50; produto 3, US$ 9,98; produto 4, US$ 4,49; e produto 5, US$ 6,87. Escreva um aplicativo que leia uma série de pares de números como segue:

a) número de produto.

b) quantidade vendida.

Seu programa deve utilizar uma instrução `switch` para determinar o preço de varejo de cada produto. Você deve calcular e exibir o valor de varejo total de todos os produtos vendidos. Utilize um loop controlado por sentinela para determinar quando o programa deve parar o loop e exibir os resultados finais.

5.18 **(*Programa de juros compostos modificado*)** Modifique o aplicativo na Figura 5.6 para utilizar apenas inteiros para calcular os juros compostos. [*Dica:* trate todas as quantidades monetárias como números inteiros em centavos. Então divida o resultado em suas partes dólar e centavos utilizando as operações divisão e resto, respectivamente. Insira uma vírgula entre as partes dólar e centavos.]

5.19 Suponha que `I = 1`, `j = 2`, `k = 3` e `m = 2`. O que cada uma das seguintes instruções imprime?

a) `System.out.println( i == 1 );`

b) `System.out.println( j == 3 );`

c) `System.out.println( ( i >= 1 ) && ( j < 4 ) );`

d) `System.out.println( ( m <= 99 ) & ( k < m ) );`

e) `System.out.println( ( j >= i ) || ( k == m ) );`

f) `System.out.println( ( k + m < j ) | ( 3 - j >= k ) );`

g) `System.out.println( !( k > m ) );`

5.20 **(*Calculando o valor de* π)** Calcule o valor de π a partir de uma série infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Imprima uma tabela que mostra o valor π aproximado calculando os 200.000 primeiros termos dessa série. Quantos termos você tem de utilizar antes de obter um valor que comece com 3.14159?

5.21 **(*Triplos de Pitágoras*)** Um triângulo retângulo pode ter lados cujos comprimentos são todos inteiros. O conjunto de três valores inteiros para os comprimentos dos lados de um triângulo retângulo é chamado triplos de Pitágoras. Os comprimentos dos três lados devem satisfazer a relação de que a soma dos quadrados de dois dos lados é igual ao quadrado da hipotenusa. Escreva um aplicativo para exibir uma tabela dos triplos de Pitágoras para `side1`, `side2` e a `hypotenuse`, todos não maiores que 500. Utilize um loop `for` triplamente aninhado que tenta todas as possibilidades. Esse é um método de computação de "força bruta". Você aprenderá nos cursos de ciência da computação mais avançados que para muitos problemas interessantes não há uma abordagem algorítmica conhecida além do uso de força bruta absoluta.

5.22 **(*Programa de impressão de triângulos modificado*)** Modifique o Exercício 5.15 para combinar seu código dos quatro triângulos de asteriscos separados de modo que todos os quatro padrões sejam impressos lado a lado. [Dica: faça uso inteligente de loops `for` aninhados.]

5.23 **(*Leis de Morgan*)** Neste capítulo, discutimos os operadores lógicos `&&`, `&`, `||`, `|`, `^` e `!`. As leis de Morgan às vezes podem tornar mais conveniente para nós expressar uma expressão lógica. Essas leis afirmam que a expressão `!(`*condition1* `&&` condition2`)` é logicamente equivalente à expressão `(!`condição1 `||` `!`condição2`)`. Além disso, a expressão `!(`*condição1*`||` condição2`)` é logicamente equivalente à expressão `(!`condição1`&&` `!`condição2`)`. Utilize as leis de Morgan para escrever expressões equivalentes para cada uma das expressões a seguir, então escreva um aplicativo para mostrar que tanto a expressão original como a nova expressão em cada caso produzem o mesmo valor.

a) `!( x < 5 ) && !( y >= 7 )`

b) `!( a == b ) || !( g != 5 )`

c) `!( ( x <= 8 ) && ( y > 4 ) )`

d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.24 (***Programa de impressão de losangos***) Escreva um aplicativo que imprime a seguinte forma de um losango. Você pode utilizar instruções de saída que imprimem um único asterisco (*), um único espaço ou um único caractere de nova linha. Maximize sua utilização de repetição (com instruções `for` aninhadas) e minimize o número de instruções de saída.

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

5.25 (***Programa de impressão de losangos modificado***) Modifique o aplicativo que você escreveu no Exercício 5.24 para ler um número ímpar no intervalo 1 a 19 para especificar o número de linhas no losango. Seu programa então deve exibir um losango do tamanho apropriado.

5.26 Uma crítica à instrução `break` e à instrução `continue` é que elas não são estruturadas. Na verdade, essas instruções sempre podem ser substituídas por instruções estruturadas, embora fazer isso possa ser difícil. Descreva de maneira geral como você removeria qualquer instrução `break` de um loop em um programa e a substituiria por alguma equivalente estruturada. [*Dica*: a instrução `break` sai de um loop do corpo do loop. A outra maneira de sair de um loop é falhando no teste de continuação do loop. Considere a possibilidade de utilizar no teste de continuação do loop um segundo teste que indica "saída prévia por causa de uma condição 'break'".] Utilize a técnica que você desenvolve aqui para remover a instrução `break` do aplicativo na Figura 5.12.

5.27 O que o seguinte segmento de programa faz?

```
for ( i = 1; i <= 5; i++ )
{
   for ( j = 1; j <= 3; j++ )
   {
      for ( k = 1; k <= 4; k++ )
         System.out.print( '*' );

      System.out.println();
   } // fim do for interno

   System.out.println();
} // fim do for externo
```

5.28 Descreva de maneira geral como você removeria qualquer instrução `continue` de um loop em um programa e a substituiria por alguma equivalente estruturada. Utilize a técnica que você desenvolve aqui para remover a instrução `continue` do programa na Figura 5.13.

5.29 (***A canção "The Twelve Days of Christmas"***) Escreva um aplicativo que utiliza instruções de repetição `switch` para imprimir a canção "The Twelve Days of Christmas". Uma instrução `switch` deve ser utilizada para imprimir o dia ("primeiro", "segundo" etc.). Uma instrução `switch` separada deve ser utilizada para imprimir o restante de cada verso. Visite o site web `en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)` para obter a letra da música.

Fazendo a diferença

5.30 (***Questionário sobre o aquecimento global***) A controversa questão do aquecimento global foi amplamente divulgada no filme "Uma verdade inconveniente", apresentando o ex-vice-presidente Al Gore. Gore e uma rede de cientistas da ONU, o Grupo Intergovernamental sobre Alterações Climáticas, dividiu o Prêmio Nobel da Paz de 2007 em reconhecimento aos "seus esforços para fomentar e disseminar melhor conhecimento sobre as mudanças climáticas feitas pelo homem". Pesquise online os dois lados da questão em relação ao aquecimento global (é recomendável pesquisar frases como "global warming skeptics" ["céticos do aquecimento global"]). Crie um questionário de múltipla escolha com cinco perguntas sobre o aquecimento global, cada pergunta tendo quatro possíveis respostas (numeradas 1–4). Seja objetivo e tente representar de uma maneira justa ambos os lados da questão. Em seguida, escreva um aplicativo que administre o questionário, calcule o número de respostas corretas (zero a cinco) e retorne uma mensagem ao usuário. Se o usuário responder corretamente cinco perguntas, imprima "Excelente"; se responder quatro, imprima "Muito bom"; se responder três ou menos, imprima "É o momento de aprimorar seu conhecimento sobre o aquecimento global" e inclua uma lista de alguns sites Web onde você encontrou os fatos.

5.31 (***Alternativas ao planejamento tributário; o "Imposto justo"***) Há muitas propostas para tornar a tributação mais justa. Verifique a iniciativa FairTax nos Estados Unidos em:

`www.fairtax.org/site/PageServer?pagename=calculator`

Pesquise como o FairTax proposto funciona. Uma sugestão é eliminar impostos de renda e a maioria dos outros impostos a favor de um imposto de consumo de 23% sobre todos os produtos e serviços que você compra. Alguns oponentes do FairTax questionam o percentual de 23% e afirmam que, por causa da maneira como o imposto é calculado, seria mais exato dizer que a taxa é 30% — verifique isso cuidadosamente. Escreva um programa que pede para o usuário inserir despesas nas várias categorias de despesas que ele tem (por exemplo, moradia, alimentação, vestuário, transporte, educação, assistência médica, férias) e então imprime o FairTax estimado que a pessoa pagaria.

A invenção mais importante do século XIX foi a invenção do método de invenção.
— **Alfred North Whitehead**

Chamem-me Ismael.
— **Herman Melville**

Quando você me chamar assim, sorria!
— **Owen Wister**

Responda-me em uma palavra.
— **William Shakespeare**

Chama o dia de ontem, faze que o tempo atrás retorne.
— **William Shakespeare**

Há um ponto em que os métodos se autodevoram.
— **Frantz Fanon**

6

Métodos: uma visão mais aprofundada

Objetivos

Neste capítulo, você aprenderá:

- Como métodos e campos `static` são associados a uma classe inteira em vez de instâncias específicas da classe.
- A utilizar métodos `Math` comuns disponíveis na Java API.
- A entender os mecanismos para passar informações entre métodos.
- Como o mecanismo de chamada/retorno de método é suportado pela pilha de chamadas de métodos e registros de ativação.
- Como pacotes agrupam classes relacionadas.
- Como utilizar a geração de números aleatórios para implementar aplicativos de jogos de azar.
- Como a visibilidade das declarações é limitada a regiões específicas dos programas.
- O que é a sobrecarga de método e como criar métodos sobrecarregados.

Sumário

6.1 Introdução
6.2 Módulos de programa em Java
6.3 Métodos `static`, campos `static` e classe `Math`
6.4 Declarando métodos com múltiplos parâmetros
6.5 Notas sobre a declaração e utilização de métodos
6.6 Pilha de chamadas de método e registros de ativação
6.7 Promoção e coerção de argumentos
6.8 Pacotes de Java API
6.9 Estudo de caso: geração de números aleatórios
6.9.1 Escalonamento e deslocamento generalizados de números aleatórios
6.9.2 Repetição de números aleatórios para teste e depuração
6.10 Estudo de caso: um jogo de azar; introdução a enumerações
6.11 Escopo das declarações
6.12 Sobrecarga de método
6.13 (Opcional) Estudo de caso de GUI e imagens gráficas: cores e formas preenchidas
6.14 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

6.1 Introdução

A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas e simples partes, ou **módulos**. Essa técnica se chama **dividir para conquistar**. Apresentamos métodos no Capítulo 3. Neste capítulo, estudaremos os métodos mais detalhadamente. Enfatizamos como declarar e utilizar métodos para facilitar o design, implementação, operação e manutenção de grandes programas.

Você verá que é possível que certos métodos, chamados métodos `static`, sejam chamados sem a necessidade de um objeto da classe para existir. Aprenderá a declarar um método com mais de um parâmetro. Também aprenderá como o Java é capaz de monitorar qual método está atualmente em execução, como variáveis locais dos métodos são mantidas na memória e como um método sabe para onde retornar depois de completar a execução.

Faremos uma breve digressão para examinarmos as técnicas de simulação com a geração de números aleatórios e desenvolveremos uma versão do jogo de dados de cassino chamado craps, que utiliza a maioria das técnicas de programação que você usou até agora neste livro. Além disso, você aprenderá como declarar valores que não podem mudar (isto é, constantes) nos seus programas.

Boa parte das classes que você utilizará ou criará ao desenvolver aplicativos terá mais de um método com o mesmo nome. Essa técnica, chamada sobrecarga, é utilizada para implementar os métodos que realizam tarefas semelhantes para argumentos de tipos diferentes ou para diferentes números de argumentos.

Continuaremos nossa discussão de métodos no Capítulo 18, "Recursão". A recursão fornece um modo intrigante de pensar nos métodos e algoritmos.

6.2 Módulos de programa em Java

Há três tipos de módulos em Java — métodos, classes e pacotes. Os programas Java são escritos combinando novos métodos e classes que você escreve com métodos e classes predefinidas disponíveis na **Java Application Programming Interface** (também conhecida como **Java API** ou **biblioteca de classes Java**) e em várias outras bibliotecas de classes. Em geral, classes relacionadas são agrupadas em pacotes de modo que possam ser importadas nos programas e reutilizadas.

Você aprenderá a agrupar suas próprias classes em pacotes no Capítulo 8. A Java API fornece uma rica coleção de classes predefinidas que contém métodos para realizar cálculos matemáticos comuns, manipulações de string, manipulações de caractere, operações de entrada/saída, operações de banco de dados, operações de rede, processamento de arquivo, verificação de erros e muitas outras tarefas úteis.

Observação de engenharia de software 6.1

Familiarize-se com a rica coleção de classes e métodos fornecidos pela Java API (`java.sun.com/javase/6/docs/api/`). Na Seção 6.8, apresentamos uma visão geral dos vários pacotes comuns. No Apêndice E, explicamos como navegar pela documentação da Java API. Não reinvente a roda. Quando possível, reutilize as classes e métodos na Java API. Isso reduz o tempo de desenvolvimento de programas e evita a introdução de erros.

Métodos (chamados de **funções** ou **procedimentos** em algumas linguagens) ajudam a modularizar um programa separando suas tarefas em unidades autocontidas. Você declarou métodos em todos os programas que escreveu até aqui. As instruções no corpo dos métodos são escritas apenas uma vez, permanecem ocultas de outros métodos e podem ser reutilizadas a partir de várias localizações em um programa.

Uma motivação para modularizar um programa em métodos é a abordagem "dividir e conquistar", que torna o desenvolvimento de programas mais gerenciável construindo programas a partir de peças mais simples e menores. Outra é a **capacidade de reutilização de software** — o uso de métodos existentes como blocos de construção para criar novos programas. Frequentemente, você pode criar programas principalmente a partir de métodos padronizados em vez de construir um código personalizado. Por exemplo, nos programas anteriores, não definimos como ler os valores dos dados a partir do teclado — o Java fornece essas capacidades na classe `Scanner`. Um terceiro motivo é evitar a repetição de código. Dividir um programa em métodos significativos torna o programa mais fácil de depurar e manter.

Observação de engenharia de software 6.2

Para promover a capacidade de reutilização de software, todos os métodos devem estar limitados à realização de uma única tarefa bem-definida e o nome do método deve expressar essa tarefa efetivamente.

Dica de prevenção de erro 6.1

Um método que realiza uma única tarefa é mais fácil de testar e depurar do que aquele que realiza muitas tarefas.

Observação de engenharia de software 6.3

Se você não puder escolher um nome conciso que expresse a tarefa de um método, seu método talvez tente realizar tarefas em demasia. Divida esse método em vários métodos menores.

Como você sabe, um método é invocado por uma chamada de método e, quando o método chamado completa sua tarefa, ele retorna um resultado ou simplesmente o controle ao chamador. Uma analogia a essa estrutura de programa é a forma hierárquica de gerenciamento (Figura 6.1). Um chefe (o chamador) solicita que um trabalhador (o método chamado) realize uma tarefa e informe (retorne) os resultados depois de completar a tarefa. O método chefe não tem conhecimento sobre como o método trabalhador realiza suas tarefas designadas. O trabalhador também pode chamar outros métodos trabalhadores, sem o que chefe saiba. Esse "ocultamento" dos detalhes de implementação promove a boa engenharia de software. A Figura 6.1 mostra o método `chefe` se comunicando com vários métodos trabalhadores de uma maneira hierárquica. O método `chefe` divide as responsabilidades entre os vários métodos `trabalhadores`. Observe que `trabalhador1` atua como um método "chefe" para `trabalhador4` e `trabalhador5`.

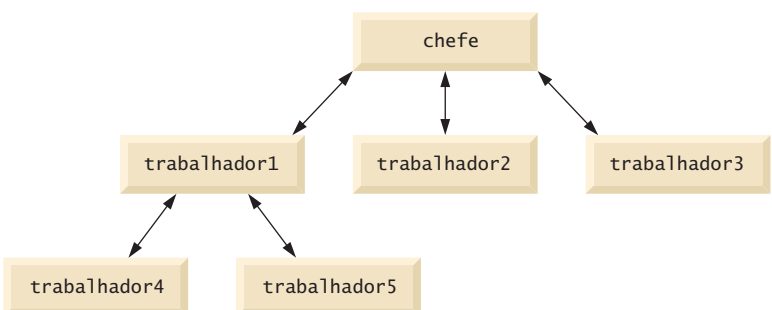


Figura 6.1 | Relacionamento hierárquico entre método trabalhador e método chefe.

6.3 Métodos `static`, campos `static` e classe `Math`

Embora a maioria dos métodos seja executada em resposta a chamadas de método em objetos específicos, isso nem sempre é o caso. Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto. Esse método se aplica à classe em que é declarado como um todo e é conhecido como método `static` ou **método de classe**. É comum que as classes contenham métodos `static` convenientes para realizar tarefas corriqueiras. Por exemplo, lembre-se de que utilizamos o método `static` `pow` da classe `Math` para elevar um valor a uma potência na Figura 5.6. Para declarar um método como `static`, coloque a palavra-chave `static` antes do tipo de retorno na declaração do método. Você pode chamar qualquer método `static` especificando o nome da classe em que o método é declarado, seguido por um ponto (`.`) e pelo nome do método, como em:

NomeDaClasse`.`*nomeDoMétodo*`(` *argumentos* `)`

Aqui, utilizamos vários métodos da classe Math para apresentar o conceito sobre os métodos static. A classe Math fornece uma coleção de métodos que permite realizar cálculos matemáticos comuns. Por exemplo, você pode calcular a raiz quadrada de 900.0 com a chamada do método static:

```
Math.sqrt( 900.0 )
```

A expressão anterior é avaliada como 30.0. O método sqrt aceita um argumento do tipo double e retorna um resultado do tipo double. Para gerar a saída do valor da chamada do método anterior na janela de comando, você poderia escrever a instrução

```
System.out.println( Math.sqrt( 900.0 ) );
```

Nessa instrução, o valor que sqrt retorna torna-se o argumento ao método println. Note que não houve necessidade de criar um objeto Math antes de chamar o método sqrt. Também observe que *todos* os métodos da classe Math são static — portanto, cada um é chamado precedendo o nome do método com o nome da classe Math e o ponto separador (.) .

Observação de engenharia de software 6.4

A classe Math faz parte do pacote java.lang, que é implicitamente importado pelo compilador, assim não é necessário importar a classe Math para utilizar seus métodos.

Os argumentos de método podem ser constantes, variáveis ou expressões. Se c = 13.0, d = 3.0 e f = 4.0, então a instrução

```
System.out.println( Math.sqrt( c + d * f ) );
```

calcula e imprime a raiz quadrada de 13.0 + 3.0 * 4.0 = 25.0 — a saber 5.0. A Figura 6.2 resume os vários métodos da classe Math. Na figura, *x* e *y* são do tipo double.

Método	Descrição	Exemplo
abs(*x*)	valor absoluto de *x*	abs(23.7) é 23.7 abs(0.0) é 0.0 abs(-23.7) é 23.7
ceil(*x*)	arredonda *x* para o menor inteiro não menor que *x*	ceil(9.2) é 10.0 ceil(-9.8) é -9.0
cos(*x*)	co-seno trigonométrico de *x* (*x* em radianos)	cos(0.0) é 1.0
exp(*x*)	método exponencial e^x	exp(1.0) é 2.71828 exp(2.0) é 7.38906
floor(*x*)	arredonda *x* para o maior inteiro não maior que *x*	floor(9.2) é 9.0 floor(-9.8) é -10.0
log(*x*)	logaritmo natural de *x* (base *e*)	log(Math.E) é 1.0 log(Math.E* Math.E) é 2.0
max(*x*,*y*)	maior valor de *x* e *y*	max(2.3, 12.7) é 12.7 max(-2.3, -12.7) é -2.3
min(*x*,*y*)	menor valor de *x* e *y*	min(2.3, 12.7) é 2.3 min(-2.3, -12.7) é -12.7
pow(*x*,*y*)	*x* elevado à potência de *y* (isto é, x^y)	pow(2.0, 7.0) é 128.0 pow(9.0, 0.5) é 3.0
sin(*x*)	seno trigonométrico de *x* (*x* em radianos)	sin(0.0) é 0.0
sqrt(*x*)	raiz quadrada de *x*	sqrt(900.0) é 30.0
tan(*x*)	tangente trigonométrica de *x* (*x* em radianos)	tan(0.0) é 0.0

Figura 6.2 | Métodos da classe Math.

Constantes da classe Math PI e E

A classe Math declara dois campos que representam constantes matemáticas comumente utilizadas — **Math.PI** e **Math.E**. A primeira, Math.PI, (3,141592653589793) é a relação da circunferência de um círculo com seu diâmetro. A constante Math.E (2,718281828459045)

é o valor da base para logaritmos naturais (calculados com o método `static` `Math` `log`). Esses campos são declarados na classe `Math` com os modificadores `public`, `final` e `static`. Torná-los `public` permite que você use esses campos nas suas próprias classes. Qualquer campo declarado com a palavra-chave **`final`** é constante — seu valor não pode alterar depois que o campo for inicializado. `PI` e `E` são declarados `final` porque os seus valores nunca mudam. Tornar esses campos `static` permite que eles sejam acessados pelo nome da classe `Math` e um ponto (`.`) separador, como ocorre com os métodos da classe `Math`. Lembre-se na Seção 3.5 de que quando cada objeto de uma classe mantém sua própria cópia de um atributo, o campo que representa o atributo também é conhecido como uma variável de instância — cada objeto (instância) da classe tem uma instância separada da variável na memória. Há campos em que cada objeto de uma classe não tem uma instância separada do campo. Esse é o caso dos campos `static`, também conhecidos como **variáveis de classe**. Quando objetos de uma classe que contém campos `static` são criados, todos os objetos dessa classe compartilham uma cópia dos campos da classe `static`. Juntas, as variáveis de classe (isto é, variáveis `static`) e as variáveis de instância representam os campos de uma classe. Examinaremos outros detalhes sobre os campos `static` na Seção 8.11.

Por que o método `main` é declarado `static`?

Por que o `main` deve ser declarado `static`? Ao executar a Java Virtual Machine (JVM) com o comando `java`, a JVM tenta invocar o método `main` da classe que você especifica — quando nenhum objeto da classe tiver sido criado. Declarar `main` como `static` permite que a JVM invoque `main` sem criar uma instância da classe. Ao executar seu aplicativo, você especifica o nome da classe como um argumento para o comando `java`, como em:

```
java NomeDaClasse argumento1 argumento2 ...
```

A JVM carrega a classe especificada pelo *NomeDaClasse* e utiliza esse nome de classe para invocar o método `main`. No comando anterior, *NomeDaClasse* é um **argumento de linha de comando** para a JVM que informa qual classe executar. Depois do *NomeDaClasse*, você também pode especificar uma lista de `Strings` (separadas por espaços) como argumentos de linha de comando que a JVM passará para seu aplicativo. Esses argumentos poderiam ser utilizados para especificar opções (por exemplo, um nome de arquivo) a fim de executar o aplicativo. Como você aprenderá no Capítulo 7, "Arrays e `ArrayLists`", seu aplicativo pode acessar esses argumentos de linha de comando e utilizá-los para personalizar o aplicativo.

6.4 Declarando métodos com múltiplos parâmetros

O aplicativo nas figuras 6.3–6.4 usa um método chamado `maximum` para determinar e retornar o maior dos três valores `double`. O método `main` da classe `MaximumFinderTest` (linhas 7–11 da Figura 6.4) cria o objeto `MaximumFinder` (linha 9) e chama seu método `determineMaximum` (linha 10) para produzir a saída do programa. No método `determineMaximum` (Figura 6.3), as linhas 14–18 solicitam que o usuário insira três valores `double` e lê a entrada fornecida pelo usuário. A linha 21 chama o método `maximum` (declarado nas linhas 28–41) para determinar o maior dos três valores que recebe como argumentos. Quando o método `maximum` retorna o resultado para a linha 21, o programa atribui o valor de retorno de `maximum` à variável local `result`. Em seguida, a linha 24 gera a saída do valor máximo. No final desta seção, discutiremos o uso do operador + na linha 24.

```
1  // Figura 6.3: MaximumFinder.java
2  // Método maximum declarado pelo programador com três parâmetros double.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtém três valores de ponto flutuante e localiza o valor máximo
8     public void determineMaximum()
9     {
10       // cria Scanner para entrada a partir da janela de comando
11       Scanner input = new Scanner( System.in );
12
13       // solicita e insere três valores de ponto flutuante
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // lê o primeiro double
17       double number2 = input.nextDouble(); // lê o segundo double
18       double number3 = input.nextDouble(); // lê o terceiro double
19
20       // determina o valor máximo
21       double result = maximum( number1, number2, number3 );
22
23       // exibe o valor máximo
24       System.out.println( "Maximum is: " + result );
25    } // fim do método determineMaximum
```

```
26
27       // retorna o máximo dos seus três parâmetros de double
28       public double maximum( double x, double y, double z )
29       {
30          double maximumValue = x; // supõe que x é o maior valor inicial
31
32          // determina se y é maior que maximumValue
33          if ( y > maximumValue )
34             maximumValue = y;
35
36          // determina se z é maior que maximumValue
37          if ( z > maximumValue )
38             maximumValue = z;
39
40          return maximumValue;
41       } // fim do método Maximum
42    } // fim da classe MaximumFinder
```

Figura 6.3 | O método `maximum` declarado pelo programador com três parâmetros `double`.

```
1  // Figura 6.4: MaximumFinderTest.java
2  // Aplicativo para testar a classe MaximumFinder.
3
4  public class MaximumFinderTest
5  {
6     // ponto de partida do aplicativo
7     public static void main( String[] args )
8     {
9        MaximumFinder maximumFinder = new MaximumFinder();
10       maximumFinder.determineMaximum();
11    } // fim de main
12 } // fim da classe MaximumFinderTest
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Figura 6.4 | Aplicativo para testar a classe `MaximumFinder`.

Considere a declaração de `maximum` (linhas 28–41). A linha 28 indica que o método retorna um valor `double`, que o nome do método é `maximum` e que o método requer três parâmetros `double` (`x`, `y` e `z`) para realizar sua tarefa. Observe que múltiplos parâmetros são especificados como uma lista separada por vírgula. Quando `maximum` é chamado a partir da linha 21, os parâmetros `x`, `y` e `z` são inicializados com os valores de argumentos `number1`, `number2` e `number3`, respectivamente. Deve haver um argumento na chamada de método para cada parâmetro (às vezes chamado **parâmetro formal**) na declaração de método. Além disso, cada argumento deve ser consistente com o tipo do parâmetro correspondente. Por exemplo, um parâmetro do tipo `double` pode receber valores como 7.35, 22 ou –0.03456, mas não `Strings` como `"hello"` nem valores `boolean` `true` ou `false`. A Seção 6.7 discute os tipos de argumentos que podem ser fornecidos em uma chamada de método para cada parâmetro de um tipo primitivo.

Para determinar o valor máximo, começamos com a suposição de que o parâmetro `x` contém o maior valor, assim a linha 30 declara a variável local `maximumValue` e a inicializa com o valor do parâmetro `x`. Naturalmente, é possível que o parâmetro `y` ou `z` contenha o maior valor real, portanto devemos comparar cada um desses valores com `maximumValue`. A instrução `if` nas linhas 33–34 determina se `y` é maior que `maximumValue`. Se for, a linha 34 atribui `y` a `maximumValue`. A instrução `if` nas linhas 37–38 determina se `z` é maior que `maximumValue`. Se for, a linha 38 atribui `z` a `maximumValue`. Nesse ponto, o maior dos três valores reside em `maximumValue`, de modo que a linha 40 retorna esse valor à linha 21. Quando o controle de programa retornar ao ponto no programa em que `maximum` foi chamado, os parâmetros de `maximum` `x`, `y` e `z` não existirão mais na memória.

Observação de engenharia de software 6.5

Métodos podem retornar no máximo um valor, mas o valor retornado poderia ser uma referência a um objeto que contém muitos valores.

Observação de engenharia de software 6.6

Variáveis devem ser declaradas como campos de uma classe somente se forem utilizadas em mais de um método da classe ou se o programa deve salvar seus valores entre chamadas aos métodos da classe.

Erro de programação comum 6.1

Declarar parâmetros de método do mesmo tipo como `float x, y` *em vez de* `float x, float y` *é um erro de sintaxe — um tipo é requerido para cada parâmetro na lista de parâmetros.*

Implementando o método `maximum` reutilizando o método `Math.max`

O corpo inteiro do nosso método maximum também poderia ser implementado com duas chamadas a `Math.max`, como a seguir:

```
return Math.max( x, Math.max( y, z ) );
```

A primeira chamada a `Math.max` especifica os argumentos `x` e `Math.max( y, z )`. Antes de qualquer chamada de método, seus argumentos devem ser avaliados para determinar seus valores. Se um argumento for uma chamada de método, a chamada de método deve ser realizada para determinar seu valor de retorno. Portanto, na instrução anterior, `Math.max( y, z )` é avaliada para determinar o valor máximo de `y` e `z`. O resultado é então passado como o segundo argumento para a outra chamada a `Math.max`, que retorna o maior dos seus dois argumentos. Esse é um bom exemplo da reutilização de software — encontramos o maior dos três valores reutilizando `Math.max`, que encontra o maior dos dois valores. Observe a concisão desse código comparado com as linhas 30–38 da Figura 6.3.

Montando strings com a concatenação de strings

O Java permite montar objetos `String` em strings maiores utilizando os operadores + ou +=. Isso é conhecido como **concatenação de strings**. Quando ambos os operandos do operador + são objetos `String`, o operador + cria um novo objeto `String` em que os caracteres do operando direito são colocados no fim daqueles no operando esquerdo — por exemplo, a expressão `"hello " + "there"` cria a `String` `"hello there"`.

Na linha 24 da Figura 6.3, a expressão `"Maximum is: " + result` utiliza o operador + com os operandos dos tipos `String` e `double`. Todos os objetos e valores primitivos em Java têm uma representação de `String`. Se um dos operandos do operador + for uma `String`, o outro é convertido em uma `String` e então os dois são concatenados. Na linha 24, o valor de `double` é convertido na sua representação `String` e colocado no final da `String` `"Maximum is: "`. Um ou mais zeros no final de um valor `double` são descartados quando o número é convertido em uma `String`, por exemplo, 9.3500 seria representado por 9.35

Para valores primitivos utilizados na concatenação de `String`, os valores primitivos são convertidos em `Strings`. Se uma `boolean` for concatenada com uma `String`, a `boolean` será convertida na `String` `"true"` ou `"false"`. Todos os objetos têm um método `toString` que retorna uma representação `String` do objeto. (Discutiremos `toString` em mais detalhes nos capítulos subsequentes.) Quando um objeto é concatenado com uma `String`, o método `toString` do objeto é chamado implicitamente para obter a representação de `String` do objeto.

Os programadores às vezes preferem dividir grandes literais de `String` em várias `Strings` menores e distribuí-las em diversas linhas de código para melhorar a legibilidade. Nesse caso, as `Strings` podem ser montadas utilizando a concatenação. Discutiremos os detalhes de `Strings` no Capítulo 16, "Strings, caracteres e expressões regulares".

Erro de programação comum 6.2

É um erro de sintaxe dividir um literal de `String` *em linhas. Se necessário, você pode dividir uma* `String` *em várias unidades menores e utilizar concatenação para formar a* `String` *desejada.*

Erro de programação comum 6.3

Confundir o operador + utilizado para concatenação de string com o operador + utilizado para adição pode levar a resultados estranhos. O Java avalia os operandos de um operador da esquerda para a direita. Por exemplo, suponha que a variável inteira `y` *tem o valor 5, a expressão* `"y + 2 = " + y + 2` *resulta na string* `"y + 2 = 52"`*, não em* `"y + 2 = 7"`*, porque o primeiro valor de* `y` *(5) é concatenado na string* `"y + 2 = "`*, então o valor 2 é concatenado na nova e maior string* `"y + 2 = 5"`*. A expressão* `"y + 2 = " + (y + 2)` *produz o resultado desejado* `"y + 2 = 7"`*.*

6.5 Notas sobre a declaração e utilização de métodos

Há três maneiras de chamar um método:

1. Utilizar um nome de método sozinho para chamar outro método da mesma classe — como `maximum(number1, number2, number3)` na linha 21 da Figura 6.3.
2. Utilizar uma variável que contém uma referência a um objeto, seguido por um ponto (`.`) e o nome de método para chamar um método do objeto referido — como a chamada de método na linha 10 da Figura 6.4, `maximumFinder.determineMaximum()`, que chama um método da classe `MaximumFinder` do método `main` de `MaximumFinderTest`.
3. Utilizar o nome de classe e um ponto (`.`) para chamar um método `static` de uma classe — como `Math.sqrt(900.0)` na Seção 6.3.

Um método `static` pode chamar qualquer método da mesma classe diretamente (isto é, utilizar o nome do método por si só) e manipular alguns dos campos da classe diretamente. Por outro lado, um método `static` só pode chamar outros métodos `static` da mesma classe diretamente e só pode manipular campos `static` na mesma classe diretamente. Para acessar os membros não `static` da classe, um método `static` deve utilizar uma referência a um objeto da classe. Lembre-se de que métodos `static` se relacionam a uma classe como um todo, enquanto métodos não `static` são associados a uma instância de classe específica (objeto) e podem manipular as variáveis de instância do objeto. Muitos objetos de uma classe, cada um com suas próprias cópias das variáveis de instância, podem existir ao mesmo tempo. Suponha que um método `static` fosse invocar um método não `static` diretamente. Como o método saberia quais variáveis de instância do objeto devem ser manipuladas? O que aconteceria se nenhum objeto da classe existisse quando o método não `static` fosse invocado? Evidentemente, isso seria problemático. Assim, o Java não permite que um método `static` acesse diretamente membros não `static` da mesma classe.

Há três maneiras de retornar o controle à instrução que chama um método. Se o método não retornar um resultado, o controle retornará quando o fluxo do programa alcançar a chave direita de fechamento do método ou quando a instrução

```
return;
```

for executada. Se o método retornar um resultado, a instrução

```
return expressão;
```

avalia a *expressão* e então retorna o resultado ao chamador.

Erro de programação comum 6.4

Declarar um método fora do corpo de uma declaração de classe ou dentro do corpo de um outro método é um erro de sintaxe.

Erro de programação comum 6.5

Omitir o tipo do valor de retorno, possivelmente `void`, em uma declaração de método é um erro de sintaxe.

Erro de programação comum 6.6

Colocar um ponto-e-vírgula após o parêntese direito que envolve a lista de parâmetros de uma declaração de método é um erro de sintaxe.

Erro de programação comum 6.7

Redeclarar um parâmetro como uma variável local no corpo do método é um erro de compilação.

Erro de programação comum 6.8

Esquecer de retornar um valor em um método que deve retornar um valor é um erro de compilação. Se um tipo de retorno além de `void` for especificado, o método deverá conter uma instrução `return` que retorne um valor consistente com o tipo de retorno do método. Retornar um valor de um método cujo tipo de retorno foi declarado como `void` é um erro de compilação.

6.6 Pilha de chamadas de método e registros de ativação

Para entender como o Java realiza chamadas de método, precisamos primeiro considerar uma estrutura de dados (isto é, a coleção de itens de dados relacionados) conhecida como **pilha**. Você pode pensar em uma pilha como análoga a uma pilha de pratos. Quando um prato é colocado na pilha, normalmente ele é colocado na parte superior (conhecido como **inserir** o prato na pilha). De maneira semelhante,

quando um prato é removido da pilha, ele é sempre removido da parte superior (conhecido como **retirar** o prato da pilha). As pilhas são conhecidas como estruturas de dados do tipo **último a entrar e primeiro a sair** (***Last-In, First-Out*** — LIFO) — o último item inserido na pilha é o primeiro item removido da pilha.

Quando um programa chama um método, o método chamado deve saber como retornar ao seu chamador, assim o endereço de retorno do método de chamada é colocado na **pilha de execução de programas** (às vezes chamada **pilha de chamadas de método**). Se uma série de chamadas de método ocorre, os sucessivos endereços de retorno são empilhados na ordem primeiro a entrar, primeiro a sair, de modo que cada método possa retornar para seu chamador.

A pilha de execução de programas também contém a memória para as variáveis locais utilizadas em cada invocação de um método durante a execução do programa. Esses dados, armazenados como uma parte da pilha de execução de programas, são conhecidos como **registro de ativação** ou **quadro de pilha** da chamada de método. Quando uma chamada de método é feita, o registro de ativação dessa chamada de método é inserido na pilha de execução de programas. Quando o método retorna ao seu chamador, o registro de ativação dessa chamada de método é retirado da pilha e essas variáveis locais não são mais conhecidas para o programa. Se uma variável local que armazena uma referência a um objeto for a única variável no programa com uma referência a esse objeto, quando o registro de ativação que contém essa variável local for retirado da pilha, o objeto não pode mais ser acessado pelo programa e consequentemente será excluído da memória pela JVM durante a "coleta de lixo". Discutiremos a coleta de lixo na Seção 8.10.

Naturalmente, a quantidade de memória em um computador é finita, portanto somente certa quantidade de memória pode ser utilizada para armazenar os registros de ativação na pilha de execução de programas. Se mais chamadas de método ocorrerem além do limite de seus registros de ativação armazenados na pilha de execução de programas, ocorrerá um erro conhecido como **estouro de pilha**.

6.7 Promoção e coerção de argumentos

Um outro recurso importante das chamadas de método é a **promoção de argumentos** — converter um valor do argumento, se possível, no tipo que o método espera receber no seu parâmetro correspondente. Por exemplo, um programa pode chamar o método `Math sqrt` com um argumento inteiro mesmo que o método espere receber um argumento `double` (mas, como veremos mais adiante, não vice-versa). A instrução

```
System.out.println( Math.sqrt( 4 ) );
```

avalia corretamente `Math.sqrt( 4 )` e imprime o valor `2.0`. A lista de parâmetros da declaração do método faz com que o Java converta o valor `int 4` no valor `double 4.0` antes de passar o valor para o método `sqrt`. Tentar essas conversões pode levar a erros de compilação se as **regras de promoção** do Java não forem satisfeitas. As regras de promoção especificam quais conversões são autorizadas — isto é, quais conversões podem ser realizadas sem perda de dados. No exemplo `sqrt` acima, um `int` é convertido em um `double` sem alterar seu valor. Mas converter um `double` em um `int` trunca a parte fracionária do valor `double` — portanto, parte do valor é perdida. Converter tipos inteiros grandes em tipos inteiros pequenos (por exemplo, `long` em `int` ou `int` em `short`) também pode resultar em valores alterados.

As regras de promoção se aplicam a expressões que contenham valores de dois ou mais tipos primitivos e a valores de tipo primitivo passados como argumentos para os métodos. Cada valor é promovido para o tipo "mais alto" na expressão. Na verdade, a expressão utiliza uma cópia temporária de cada valor — os tipos dos valores originais permanecem inalterados. A Figura 6.5 lista os tipos primitivos e os tipos para os quais cada um pode ser promovido. Observe que as promoções válidas para um dado tipo sempre são para um tipo mais alto na tabela. Por exemplo, um `int` pode ser promovido aos tipos `long`, `float` e `double` mais altos.

Tipo	Promoções válidas
`double`	None
`float`	`double`
`long`	`float` ou `double`
`int`	`long`, `float` ou `double`
`char`	`int`, `long`, `float` ou `double`
`short`	`int`, `long`, `float` ou `double` (mas não `char`)
`byte`	`short`, `int`, `long`, `float` ou `double` (mas não `char`)
`boolean`	Nenhuma (os valores `boolean` não são considerados como números em Java)

Figura 6.5 | Promoções permitidas para tipos primitivos.

Converter valores em tipos mais baixos na tabela da Figura 6.5 resultará em diferentes valores se o tipo mais baixo não puder representar o valor do tipo mais alto (por exemplo, o valor `int` 2000000 não pode ser representado como um `short` e qualquer número de ponto flutuante com dígitos depois do seu ponto de fração decimal não pode ser representado em um tipo inteiro como `long`, `int` ou `short`).

Portanto, nos casos em que as informações podem ser perdidas devido à conversão, o compilador Java requer que você utilize um operador de coerção (introduzido na Seção 4.9) para forçar explicitamente que a conversão ocorra — do contrário, ocorre um erro de compilação. Isso permite que você "assuma controle" do compilador. Você essencialmente diz, "Sei que essa conversão poderia causar perda das informações, mas, aqui, para meus propósitos, isso não é um problema". Suponha que o método `square` calcule o quadrado de um inteiro e assim requeira um argumento `int`. Para chamarmos `square` com um argumento `double` chamado `doubleValue`, deveríamos escrever a chamada de método como:

```
square( (int) doubleValue )
```

Essa chamada de método faz uma coerção explícita (converte) de uma *cópia* do valor da variável `doubleValue` em um inteiro para uso no método `square`. Assim, se o valor do `doubleValue` for `4.5`, o método receberá o valor `4` e retornará `16`, não `20.25`.

Erro de programação comum 6.9

Converter um valor de tipo primitivo em um outro tipo primitivo pode alterar o valor se o novo tipo não for uma promoção válida. Por exemplo, converter um valor de ponto flutuante em um valor inteiro pode introduzir erros de truncamento (perda da parte fracionária) no resultado.

6.8 Pacotes da Java API

Como vimos, o Java contém muitas classes predefinidas que são agrupadas em categorias de classes relacionadas chamadas pacotes. Em conjunto, elas são conhecidos como a Java API (Java Application Programming Interface) ou biblioteca de classes Java. Uma grande capacidade do Java são as milhares de classes da Java API. Alguns pacotes-chave da Java API são descritos na Figura 6.6, que representa só uma pequena parte dos componentes reutilizáveis na Java API.

Pacote	Descrição
`java.applet`	O **Java Applet Package** contém uma classe e várias interfaces exigidas para criar applets Java — programas que executam nos navegadores da Web. Os applets são discutidos no Capítulo 23, "Applets e Java Web Start"; as interfaces são discutidas no Capítulo 10, "Programação orientada a objetos: polimorfismo".
`java.awt`	O **Java Abstract Window Toolkit Package** contém as classes e interfaces exigidas para criar e manipular GUIs em versões anteriores do Java. Nas versões atuais do Java, os componentes GUI Swing dos pacotes `javax.swing` são geralmente usados em seu lugar. (Alguns elementos do pacote `java.awt` são discutidos no Capítulo 14, "Componentes GUI: Parte 1", Capítulo 15, "Imagens gráficas e Java2D™" e Capítulo 25, "Componentes GUI: Parte 2".)
`java.awt.event`	O **Java Abstract Window Toolkit Event Package** contém classes e interfaces que permitem o tratamento de eventos para componentes GUI tanto nos pacotes `java.awt` como `javax.swing`. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e o Capítulo 25, "Componentes GUI: Parte 2".)
`java.awt.geom`	O **Java 2D Shapes Package** contém classes e interfaces para trabalhar com as avançadas capacidades gráficas bidimensionais do Java. (Consulte o Capítulo 15, "Imagens gráficas e Java 2D™".)
`java.io`	O **Java Input/Output Package** contém classes e interfaces que permitem aos programas gerar entrada e saída de dados. (Consulte o Capítulo 17, "Arquivos, fluxos e serialização de objetos".)
`java.lang`	O **Java Language Package** contém classes e interfaces (discutidas em todo o livro) que são exigidas por muitos programas Java. Esse pacote é importado pelo compilador em todos os programas.
`java.net`	O **Java Networking Package** contém classes e interfaces que permitem aos programas comunicar-se via redes de computadores, como a Internet. (Consulte o Capítulo 27, "Redes".)
`java.sql`	O **JDBC Package** contém classes e interfaces para trabalhar com bancos de dados. (Consulte o Capítulo 28, "Acesso a bancos de dados com o JDBC".)
`java.text`	O **Java Text Package** contém classes e interfaces que permitem aos programas manipular números, datas, caracteres e strings. O pacote fornece recursos de internacionalização que permitem a um programa ser personalizado para localidades (por exemplo, um programa pode exibir strings em diferentes idiomas com base no país do usuário).
`java.util`	O **Java Utilities Package** contém classes utilitárias e interfaces que permitem ações como manipulações de data e hora, processamento de número aleatório (classe `Random`) e o armazenamento e o processamento de grandes quantidades de dados. (Consulte o Capítulo 20, "Coleções genéricas".)

Pacote	Descrição
`java.util.concurrent`	O **Java Concurrency Package** contém classes utilitárias e interfaces para implementar programas que podem realizar múltiplas tarefas paralelamente. (Consulte o Capítulo 26, "Multithreading".)
`javax.media`	O **Java Media Framework Package** contém classes e interfaces para trabalhar com capacidades multimídia do Java. (Consulte o Capítulo 24, "Multimídia: Applets e aplicativos".)
`javax.swing`	O **Java Swing GUI Components Package** contém classes e interfaces para componentes GUI Swing do Java que fornecem suporte para GUIs portáveis. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e no Capítulo 25: "Componentes GUI: Parte 2".)
`javax.swing.event`	O **Java Swing Event Package** contém classes e interfaces que permitem o tratamento de eventos (por exemplo, responder a cliques de botão) para componentes GUI do pacote `javax.swing`. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e no Capítulo 25, "Componentes GUI: Parte 2".)
`javax.xml.ws`	O **JAX-WS Package** contém classes e interfaces para trabalhar com serviços da Web no Java. (Consulte o Capítulo 31, "Serviços da Web".)

Figura 6.6 | Pacotes da Java API (um subconjunto).

O conjunto de pacotes disponíveis no Java SE 6 é bastante grande. Além dos pacotes resumidos na Figura 6.6, o Java SE 6 inclui pacotes para imagens gráficas complexas, interfaces gráficas com o usuário avançadas, impressão, rede avançada, segurança, processamento de banco de dados, multimídia, acessibilidade (para pessoas com deficiências), programação concorrente, criptografia, processamento de XML e muitas outras capacidades. Para uma visão geral dos pacotes no Java SE 6, visite:

```
java.sun.com/javase/6/docs/api/overview-summary.html
```

Muitos outros pacotes também estão disponíveis para download em `java.sun.com`.

Você pode localizar informações adicionais sobre os métodos de uma classe Java predefinida na documentação da Java API em `java.sun.com/javase/6/docs/api/`. Ao visitar esse site, clique no link **Index** para ver uma listagem alfabética de todas as classes e métodos da Java API. Localize o nome da classe e clique no seu link para ver a descrição on-line dessa classe. Clique no link **METHOD** para ver uma tabela dos métodos da classe. Cada método `static` será listado com a palavra "`static`" precedendo o tipo de retorno do método.

6.9 Estudo de caso: geração de números aleatórios

Agora faremos uma divertida digressão para discutir um tipo popular de aplicativo de programação — a simulação de jogos. Nesta seção, e na seção a seguir, desenvolveremos um programa bem estruturado de jogos com múltiplos métodos. Esse programa utiliza a maioria das instruções de controle apresentadas até aqui neste livro e introduz vários novos conceitos de programação.

Há algo no ar de um cassino que anima as pessoas — dos altos apostadores nas mesas de mogno cobertas de feltro dos jogos de dados aos pequenos apostadores nos caça-níqueis. Trata-se do **elemento chance**, a possibilidade de que a sorte converta uma pequenas quantia de dinheiro em uma fortuna. O elemento chance pode ser introduzido em um programa via um objeto da classe **`Random`** (pacote `java.util`) ou via o método `static` `random` da classe `Math`. Os objetos da classe `Random` podem produzir valores aleatórios `boolean`, `byte`, `float`, `double`, `int`, `long` e gaussianos, enquanto o método `Math` `random` pode produzir somente os valores `double` no intervalo `0,0` $\leq x <$ `1,0`, onde x é o valor retornado pelo método `random`. Nos próximos vários exemplos, utilizaremos objetos da classe `Random` para produzir valores aleatórios.

Um novo objeto gerador de números aleatórios pode ser criado como a seguir:

```
Random randomNumbers = new Random();
```

Ele pode ser usado para gerar valores `boolean`, `byte`, `float`, `double`, `int`, `long` e gaussianos aleatórios — aqui, discutiremos apenas os valores `int` aleatórios. Para mais informações sobre a classe `Random`, veja `java.sun.com/javase/6/docs/api/java/util/Random.html`.

Considere a seguinte instrução:

```
int randomValue = randomNumbers.nextInt();
```

O método `Random` **`nextInt`** gera um valor `int` aleatório no intervalo de –2,147,483,648 a +2,147,483,647, inclusive. Se ele realmente produzir valores aleatoriamente, então cada valor no intervalo deve ter uma chance (ou probabilidade) igual de ser escolhido toda vez que `nextInt` é chamado. Os números, na verdade, são **números pseudoaleatórios** — uma sequência de valores produzidos por um cálculo matemático complexo. O cálculo utiliza o horário do dia atual (que, naturalmente, muda constantemente) para **semear** o gerador de números aleatórios de tal maneira que cada execução de um programa fornece uma sequência diferente de valores aleatórios.

O intervalo de valores produzido diretamente pelo método `nextInt` costuma diferir do intervalo de valores requerido em um aplicativo Java particular. Por exemplo, um programa que simula lançamento de moeda talvez requeira somente 0 para "caras" e 1 para "coroas". Um programa que simula o lançamento de um dado de seis faces exigiria inteiros aleatórios no intervalo de 1 a 6. Um programa que prevê aleatoriamente o próximo tipo de espaçonave (entre quatro possibilidades) que voará pelo horizonte em um videogame exigiria inteiros aleatórios no intervalo de 1 a 4. Para esses casos, a classe `Random` fornece uma outra versão do método `nextInt` que recebe um argumento `int` e retorna um valor a partir de 0, mas sem incluí-lo, até o valor do argumento. Por exemplo, para o lançamento de moedas, a seguinte instrução retorna 0 ou 1.

```
int randomValue = randomNumbers.nextInt( 2 );
```

Lançando um dado de seis faces

Para demonstrar números aleatórios, vamos desenvolver um programa que simula 20 lançamentos de um dado de seis faces e exibe o valor de cada lançamento. Iniciamos utilizando `nextInt` para produzir os valores aleatórios no intervalo de 0 a 5, como a seguir:

```
face = randomNumbers.nextInt( 6 );
```

O argumento 6 — chamado de **fator de escala** — representa o número de valores únicos que `nextInt` deve produzir (nesse caso seis — 0, 1, 2, 3, 4 e 5). Essa manipulação é chamada **escalonar** o intervalo de valores produzido pelo método `Random nextInt`.

Um dado com seis faces tem os números de 1 a 6 nas suas faces, não de 0 a 5. Assim, **deslocamos** o intervalo dos números produzidos adicionando um **valor de deslocamento** — nesse caso 1 — para nosso resultado anterior, como em:

```
face = 1 + randomNumbers.nextInt( 6 );
```

O valor de deslocamento (1) especifica o primeiro valor no intervalo desejado de inteiros aleatórios. A instrução anterior atribui `face` a um inteiro aleatório no intervalo de 1 a 6.

A Figura 6.7 mostra duas saídas de exemplo que confirmam o fato de que os resultados do cálculo anterior são inteiros no intervalo de 1 a 6 e que cada execução do programa pode produzir uma sequência diferente de números aleatórios. A linha 3 importa a classe `Random` do pacote `java.util`. A linha 9 cria o objeto `Random randomNumbers` para produzir valores aleatórios. A linha 16 executa 20 vezes em um loop para lançar o dado. A instrução `if` (linhas 21–22) no loop inicia uma nova linha de saída depois de cada cinco números.

```
1  // Figura 6.7: RandomIntegers.java
2  // Inteiros aleatórios deslocados e escalonados.
3  import java.util.Random; // o programa utiliza a classe Random
4
5  public class RandomIntegers
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // gerador de número aleatório
10       int face; // armazena cada inteiro aleatório gerado
11
12       // faz o loop 20 vezes
13       for ( int counter = 1; counter <= 20; counter++ )
14       {
15          // seleciona o inteiro aleatório entre 1 a 6
16          face = 1 + randomNumbers.nextInt( 6 );
17
18          System.out.printf( "%d  ", face ); // exibe o valor gerado
19
20          // se o contador for divisível por 5, inicia uma nova linha de saída
21          if ( counter % 5 == 0 )
22             System.out.println();
23       } // for final
24    } // fim de main
25 } // fim da classe RandomIntegers
```

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Figura 6.7 | Inteiros aleatórios deslocados e escalonados.

Lançando um dado de seis faces 6.000 vezes

Para mostrar que os números produzidos por `nextInt` ocorrem com probabilidade aproximadamente igual, vamos simular 6.000 lançamentos de um dado com o aplicativo da Figura 6.8. Todo número inteiro de 1 a 6 deve aparecer aproximadamente 1.000 vezes.

```
1  // Figura 6.8: RollDie.java
2  // Rola um dado de seis lados 6 mil vezes.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // gerador de número
10                                            // aleatório
11       int frequency1 = 0; // mantém a contagem de 1s lançados
12       int frequency2 = 0; // contagem de 2s lançados
13       int frequency3 = 0; // contagem de 3s lançados
14       int frequency4 = 0; // contagem de 4s lançados
15       int frequency5 = 0; // contagem de 5s lançados
16       int frequency6 = 0; // contagem de 6s lançados
17
18       int face; // armazena o valor lançado mais recentemente
19
20       // soma 6.000 lançamentos de um dado
21       for ( int roll = 1; roll <= 6000; roll++ )
22       {
23          face = 1 + randomNumbers.nextInt( 6 ); // número entre 1 a 6
24
25          // define o valor de lançamento de 1 a 6 e incrementa o contador apropriado
26          switch ( face )
27          {
28             case 1:
29                ++frequency1; // incrementa o contador de 1s
30                break;
31             case 2:
32                ++frequency2; // incrementa o contador de 2s
33                break;
34             case 3:
35                ++frequency3; // incrementa o contador de 3s
36                break;
37             case 4:
38                ++frequency4; // incrementa o contador de 4s
39                break;
40             case 5:
41                ++frequency5; // incrementa o contador de 5s
42                break;
43             case 6:
44                ++frequency6; // incrementa o contador de 6s
45                break; // opcional no final do switch
46          } // fim do switch
47       } // for final
48
49       System.out.println( "Face\tFrequency" ); // cabeçalhos de saída
50       System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51          frequency1, frequency2, frequency3, frequency4,
52          frequency5, frequency6 );
53    } // fim de main
54 } // fim da classe RollDie
```

```
Face    Frequency
1       982
2       1001
3       1015
4       1005
5       1009
6       988
```

```
Face    Frequency
1       1029
2       994
3       1017
4       1007
5       972
6       981
```

Figura 6.8 | Rolando um dado de seis lados 6.000 vezes.

Como as saídas de exemplo mostram, escalonar e deslocar os valores produzidos por `nextInt` permite que o programa simule o lançamento de um dado de seis faces. O aplicativo utiliza as instruções de controle aninhadas (a `switch` é aninhada dentro da `for`) para determinar o número de vezes que a face do dado aparecerá. A instrução `for` (linhas 21–47) itera 6.000 vezes. Durante cada iteração, a linha 23 produz um valor aleatório de 1 a 6. Esse valor é então utilizado como a expressão de controle (linha 26) da instrução `switch` (linhas 26–46). Com base no valor de `face`, a instrução `switch` incrementa uma das seis variáveis de contador durante cada iteração do loop. Ao estudarmos arrays no Capítulo 7, mostraremos uma maneira elegante de substituir toda a instrução `switch` nesse programa por uma única instrução! Observe que essa instrução `switch` não tem nenhum caso `default`, porque temos um `case` para cada possível valor do dado que a expressão na linha 23 produziria. Execute o programa e observe os resultados. Como você verá, toda vez que executar esse programa, ele produzirá resultados diferentes.

6.9.1 Escalonamento e deslocamento generalizados de números aleatórios

Anteriormente, simulamos o lançamento de um dado de seis lados com a instrução:

```
face = 1 + randomNumbers.nextInt( 6 );
```

Essa instrução sempre atribui à variável `face` um inteiro no intervalo $1 \leq face \leq 6$. A largura desse intervalo (isto é, o número de inteiros consecutivos no intervalo) é 6 e o número inicial no intervalo é 1. Consultando a instrução anterior, vemos que a largura do intervalo é determinada pelo número 6 que é passado como um argumento para o método `Random` `nextInt` e o número inicial do intervalo é o número 1 que é adicionado a `randomNumberGenerator.nextInt(6)`. Podemos generalizar esse resultado como

```
number = valorDoDeslocamento + randomNumbers.nextInt( fatorDeEscalonamento );
```

onde *valorDoDeslocamento* especifica o primeiro número no intervalo desejado de inteiros consecutivos e *fatorDeEscalonamento* especifica quantos números estão no intervalo.

Também é possível escolher inteiros aleatoriamente a partir de conjuntos de valores além dos intervalos de inteiros consecutivos. Por exemplo, para obter um valor aleatório na sequência 2, 5, 8, 11 e 14, você poderia utilizar a instrução:

```
number = 2 + 3 * randomNumbers.nextInt( 5 );
```

Nesse caso, `randomNumberGenerator.nextInt(5)` produz os valores do intervalo de 0 a 4. Cada valor produzido é multiplicado por 3 para produzir um número na sequência 0, 3, 6, 9 e 12. Adicionamos 2 a esse valor para deslocar o intervalo de valores e obter um valor da sequência 2, 5, 8, 11 e 14. Podemos generalizar esse resultado como:

```
number = valorDeDeslocamento + diferençaEntreValores * randomNumbers.nextInt( fatorDeEscalonamento );
```

onde *valorDeDeslocamento* especifica o primeiro número no intervalo desejado de valores, *diferençaEntreValores* representa a diferença constante entre números consecutivos na sequência e *fatorDeEscalonamento* especifica quantos números estão no intervalo.

6.9.2 Repetição de números aleatórios para teste e depuração

Os métodos da classe `Random` geram, na verdade, números pseudoaleatórios com base em cálculos matemáticos complexos — a sequência de números parece ser aleatória. O cálculo que produz os números usa a hora do dia como um **valor de semente** para alterar o ponto inicial da sequência. Cada novo objeto `Random` semeia a si mesmo com um valor baseado no relógio do sistema do computador no momento em que o objeto é criado, permitindo que cada execução de um programa produza uma sequência diferente de números aleatórios.

Ao depurar um aplicativo, às vezes é útil repetir a mesma sequência exata de números pseudoaleatórios durante cada execução do programa. Essa repetitividade permite provar que seu aplicativo funciona de acordo com uma sequência específica de números aleatórios antes de você testar o programa com diferentes sequências de números aleatórios. Se a repetitividade for importante, você poderá criar um objeto `Random` como a seguir:

```
Random randomNumbers = new Random( seedValue );
```

O argumento `seedValue` (do tipo `long`) semeia o cálculo de números aleatórios. Se o mesmo `seedValue` for utilizado todas as vezes, o objeto `Random` produz a mesma sequência de números. Você pode configurar a semente de um objeto `Random` em qualquer momento durante a execução do programa chamando o método `set` do objeto, como em:

```
randomNumbers.set( seedValue );
```

Dica de prevenção de erro 6.2

Enquanto estiver desenvolvendo um programa, crie o objeto `Random` com um valor específico de semente para produzir uma sequência repetível de números toda vez que o programa for executado. Se ocorrer um erro de lógica, corrija-o e teste o programa novamente com o mesmo valor de semente — isso permite reconstruir a mesma sequência de números que causou o erro. Depois que os erros de lógica foram removidos, crie o objeto `Random` sem utilizar um valor de semente, fazendo com que o objeto `Random` gere uma nova sequência de números aleatórios toda vez que o programa é executado.

6.10 Estudo de caso: um jogo de azar; introdução a enumerações

Um jogo popular de azar é um jogo de dados conhecido como craps, que é jogado em cassinos e nas ruas de todo o mundo. As regras do jogo são simples e diretas:

Você lança dois dados. Cada dado tem seis faces que contêm um, dois, três, quatro, cinco e seis pontos, respectivamente. Depois que os dados pararam de rolar, a soma dos pontos nas faces viradas para cima é calculada. Se a soma for 7 ou 11 no primeiro lance, você ganha. Se a soma for 2, 3 ou 12 no primeiro lance (chamado "craps"), você perde (isto é, a "casa" ganha). Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se sua "pontuação". Para ganhar, você deve continuar a rolar os dados até "fazer sua pontuação" (isto é, obter um valor igual à sua pontuação). Você perde se obtiver um 7 antes de fazer sua pontuação.

O aplicativo nas figuras 6.9 e 6.10 simula o craps, utilizando métodos para implementar a lógica do jogo. No método `main` da classe `CrapsTest` (Figura 6.10), a linha 8 cria um objeto da classe `Craps` (Figura 6.9) e a linha 9 chama seu método `play` para começar o jogo. O método `play` (Figura 6.9, linhas 21–65) chama o método `rollDice` (Figura 6.9, linhas 68–81), conforme necessário, para lançar os dados e calcular a soma. As saídas de exemplo (Figura 6.10) demonstram a vitória e a perda na primeira rolagem e em uma rolagem subsequente.

```
1  // Figura 6.9: Craps.java
2  // A classe Craps simula o jogo de dados craps.
3  import java.util.Random;
4
5  public class Craps
6  {
7     // cria gerador de números aleatórios para uso no método rollDice
8     private static final Random randomNumbers = new Random();
9
10    // enumeração com constantes que representam o status do jogo
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constantes que representam lançamentos comuns dos dados
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
20    // joga uma partida de craps
21    public void play()
22    {
```

```
23          int myPoint = 0; // pontos se não ganhar ou perder na 1a. rolagem
24          Status gameStatus; // pode conter CONTINUE, WON ou LOST
25
26          int sumOfDice = rollDice(); // primeira rolagem dos dados
27
28          // determina o status do jogo e a pontuação com base no primeiro lançamento
29          switch ( sumOfDice )
30          {
31             case SEVEN: // ganha com 7 no primeiro lançamento
32             case YO_LEVEN: // ganha com 11 no primeiro lançamento
33                gameStatus = Status.WON;
34                break;
35             case SNAKE_EYES: // perde com 2 no primeiro lançamento
36             case TREY: // perde com 3 no primeiro lançamento
37             case BOX_CARS: // perde com 12 no primeiro lançamento
38                gameStatus = Status.LOST;
39                break;
40             default: // não ganhou nem perdeu, então registra a pontuação
41                gameStatus = Status.CONTINUE; // jogo não terminou
42                myPoint = sumOfDice; // informa a pontuação
43                System.out.printf( "Point is %d\n", myPoint );
44                break; // opcional no final do switch
45          } // fim do switch
46
47          // enquanto o jogo não estiver completo
48          while (gameStatus == Status.CONTINUE ) // nem WON nem LOST
49          {
50             sumOfDice = rollDice(); // lança os dados novamente
51
52             // determina o status do jogo
53             if ( sumOfDice == myPoint ) // vitória por pontuação
54                gameStatus = Status.WON;
55             else
56                if ( sumOfDice == SEVEN ) // perde obtendo 7 antes de atingir a pontuação
57                   gameStatus = Status.LOST;
58          } // fim do while
59
60          // exibe uma mensagem ganhou ou perdeu
61          if (gameStatus == Status.WON )
62             System.out.println( "Player wins" );
63          else
64             System.out.println( "Player loses" );
65       } // fim do método play
66
67       // lança os dados, calcula a soma e exibe os resultados
68       public int rollDice()
69       {
70          // seleciona valores aleatórios do dado
71          int die1 = 1 + randomNumbers.nextInt( 6 ); // Primeiro lançamento do dado
72          int die2 = 1 + randomNumbers.nextInt( 6 ); // Segundo lançamento do dado
73
74          int sum = die1 + die2; // soma dos valores dos dados
75
76          // exibe os resultados desse lançamento
77          System.out.printf( "Player rolled %d + %d = %d\n",
78             die1, die2, sum );
79
80          return sum; // retorna a soma dos dados
81       } // fim do método rollDice
82    } // fim da classe Craps
```

Figura 6.9 | A classe `Craps` simula o jogo de dados craps.

```
1  // Figura 6.10: CrapsTest.java
2  // Aplicativo para testar a classe Craps.
3
4  public class CrapsTest
5  {
6     public static void main( String[] args )
7     {
8        Craps game = new Craps();
9        game.play(); // joga um jogo de craps
10    } // fim de main
11 } // fim da classe CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Figura 6.10 | Aplicativo para testar a classe `Craps`.

Método `rollDice`

Nas regras desse jogo, o jogador deve lançar dois dados no primeiro lançamento e deve fazer o mesmo em todos os lançamentos subsequentes. Declaramos o método `rollDice` (Figura 6.9, linhas 68–81) para lançar o dado, calcular e imprimir sua soma. O método `rollDice` é declarado uma vez, mas é chamado a partir de dois lugares (linhas 26–50) no método `play`, que contém a lógica para um jogo de craps completo. O método `rollDice` não recebe nenhum argumento, então tem uma lista vazia de parâmetro. Toda vez que é chamado, `rollDice` retorna a soma dos dados, assim o tipo de retorno `int` é indicado no cabeçalho do método (linha 68). Embora as linhas 71–72 pareçam idênticas (exceto quanto aos nomes dos dados), elas não produzem necessariamente o mesmo resultado. Cada uma dessas instruções produz um valor aleatório no intervalo de 1–6. Observe que `randomNumbers` (utilizado nas linhas 71–72) não é declarado no método. Em vez disso, ele é declarado como uma variável de instância `private static final` da classe e inicializado na linha 8. Isso permite criar um objeto `Random` que é reutilizado a cada chamada a `rollDice`. Uma vez que esse objeto é `static`, ele também seria compartilhado por todas as instâncias da classe `Craps` no programa.

Variáveis locais do método `play`

O jogo é razoavelmente complexo. O jogador pode ganhar ou perder na primeira rolagem ou em qualquer rolagem subsequente. O método `play` (linhas 21–65) utiliza a variável local `myPoint` (linha 23) para armazenar a "pontuação" se o jogador não ganhar nem perder no primeiro lançamento, a variável local `gameStatus` (linha 24) para monitorar o status geral do jogo e a variável local `sumOfDice` (linha 26) para manter a soma dos dados do lançamento mais recente. Observe que `myPoint` é inicializada como `0` a fim de assegurar que o aplicativo compilará. Se você não inicializar `myPoint`, o compilador emitirá um erro, porque `myPoint` não recebe um valor em cada `case` da instrução `switch`, e assim o programa pode tentar utilizar `myPoint` antes de receber um valor. Em vez disso, um valor *é* atribuído a `gameStatus` em *todo* `case` da instrução `switch` — assim, garante-se que essa variável será inicializada antes de ser utilizada e que, portanto, não precisa ser inicializada.

Tipo `enum Status`

A variável local `gameStatus` (linha 24) é declarada como um novo tipo chamado `Status` (declarada na linha 11). O tipo `Status` é um membro `private` da classe `Craps`, porque `Status` será utilizado somente nessa classe. `Status` é um tipo chamado de **enumeração**, que, em sua forma mais simples, declara um conjunto de constantes representadas por identificadores. Uma enumeração é um tipo especial de classe que é introduzida pela palavra-chave **`enum`** e um nome do tipo (nesse caso, `Status`). Como com as classes, as chaves delimitam o corpo de uma declaração `enum`. Dentro das chaves há uma lista separada por vírgulas de **constantes de enumeração**, cada uma representando um valor único. Os identificadores em uma `enum` devem ser únicos. (Você aprenderá mais sobre enumerações no Capítulo 8.)

Boa prática de programação 6.1

Utilize somente letras maiúsculas nos nomes das constantes de enumeração. Isso destaca as constantes e nos lembra de que as constantes de enumeração não são variáveis.

Variáveis do tipo `Status` podem receber somente as três constantes declaradas na enumeração (linha 11) ou ocorrerá um erro de compilação. Quando se ganha o jogo, o programa configura a variável local `gameStatus` como `Status.WON` (linhas 33 e 54). Quando se perde o jogo, o programa configura a variável local `gameStatus` como `Status.LOST` (linhas 38 e 57). Caso contrário, o programa configura a variável local `gameStatus` como `Status.CONTINUE` (linha 41) para indicar que o jogo não terminou e que os dados devem ser lançados novamente.

Boa prática de programação 6.2

Usar constantes de enumeração (como `Status.WON`, `Status.LOST` e `Status.CONTINUE`) em vez de valores literais (como 0, 1 e 2) torna os programas mais fáceis de ler e manter.

Lógica do método `play`

A linha 26 do método `play` chama `rollDice`, que seleciona dois valores aleatórios entre 1 e 6, exibe os valores do primeiro dado, do segundo dado e a soma dos dados, e retorna a soma. Em seguida, o método `play` insere a instrução `switch` nas linhas 29–45, que utilizam o valor `sumOfDice` na linha 26 para determinar se o jogo foi ganho ou perdido, ou se deve continuar com um outro lançamento. As somas dos dados que resultariam em uma vitória ou perda no primeiro lançamento são declaradas como constantes `public static final int` nas linhas 14–18. Estas são utilizadas nos `case`s da instrução `switch`. Os nomes dos identificadores utilizam a terminologia de cassino para essas somas. Observe que essas constantes, como ocorre com constantes `enum`, são declaradas, por convenção, com todas as letras maiúsculas, fazendo-as se destacar no programa. As linhas 31–34 determinam se o jogador ganhou no primeiro lançamento com `SEVEN` (7) ou `YO_LEVEN` (11). As linhas 35–39 determinam se o jogador perdeu no primeiro lançamento com `SNAKE_EYES` (2), `TREY` (3) ou `BOX_CARS` (12). Depois do primeiro lançamento, se o jogo não tiver acabado, a opção `default` (linhas 40–44) configura `gameStatus` como `Status.CONTINUE`, salva `sumOfDice` em `myPoint` e exibe a pontuação.

Se ainda estivermos tentando "fazer nossa melhor pontuação" (isto é, o jogo continua a partir de um lançamento anterior), as linhas 48–58 serem executadas. A linha 50 lança os dados novamente. Se `sumOfDice` coincidir com `myPoint` (linha 53), a linha 54 configura `gameStatus` como `Status.WON`, o loop termina porque o jogo está completo. Se `sumOfDice` for `SEVEN` (linha 56), a linha 57 configura `gameStatus` como `Status.LOST`, e o loop termina porque o jogo está completo. Quando o jogo é concluído, as linhas 61–64 exibem uma mensagem que indica se o jogador ganhou ou perdeu e o programa termina.

Note a utilização dos vários mecanismos de controle de programa que já discutimos. A classe `Craps` em colaboração com a classe `CrapsTest` usa três métodos — `main`, `play` (chamado a partir `main`) e `rollDice` (chamado duas vezes a partir de `play`) — e as instruções `switch`, `while`, `if...else` e instruções de controle `if` aninhadas. Observe também o uso de múltiplos rótulos `case` na instrução `switch` para executar as mesmas instruções para somas de `SEVEN` e `YO_LEVEN` (linhas 31–32) e para somas de `SNAKE_EYES`, `TREY` e `BOX_CARS` (linhas 35–37).

Por que algumas constantes não são definidas como constantes `enum`

Você poderia questionar por que declaramos as somas dos dados como constantes `public final static int` em vez de constantes `enum`. A resposta reside no fato de que o programa deve comparar a variável `int sumOfDice` (linha 26) com essas constantes para determinar o resultado de cada lançamento. Suponha que fôssemos declarar `enum Sum` contendo constantes (por exemplo, `Sum.SNAKE_EYES`) que representam as cinco somas utilizadas no jogo e, então, utilizar essas constantes nos `case`s da instrução `switch` (linhas 29–45). Fazer isso evitaria o uso de `sumOfDice` como a expressão de controle da instrução `switch`, pois o Java não permite que um `int` seja comparado com uma constante de enumeração. Para alcançar a mesma funcionalidade do programa atual, teríamos de usar uma variável `currentSum` do tipo `Sum` como a expressão de controle do `switch`. Infelizmente, o Java não fornece uma maneira fácil de converter um valor `int` em uma constante `enum` particular. Converter um `int` em uma constante `enum` poderia ser feito com uma instrução `switch` separada. Obviamente, isso seria trabalhoso e não aprimoraria a legibilidade do programa (anulando assim o objetivo do uso de uma `enum`).

6.11 Escopo das declarações

Você viu declarações de várias entidades Java como classes, métodos, variáveis e parâmetros. As declarações introduzem nomes que podem ser utilizados para referenciar essas entidades Java. O **escopo** de uma declaração é a parte do programa que pode referenciar a entidade declarada pelo seu nome. Diz-se que essa entidade está "no escopo" para essa parte do programa. Esta seção introduz várias questões importantes de escopo. (Para mais informações sobre escopo, consulte *Java Language Specification, Section 6.3: Scope of a Declaration*, em `java.sun.com/docs/books/jls/third_edition/html/names.html#103228`.)

As regras básicas de escopo são estas:

1. o escopo de uma declaração de parâmetro é o corpo do método em que a declaração aparece;
2. o escopo de uma declaração de variável local é do ponto em que a declaração aparece até o final desse bloco;
3. o escopo de uma declaração de variável local que aparece na seção de inicialização do cabeçalho de uma instrução `for` é o corpo da instrução `for` e as outras expressões no cabeçalho;
4. o escopo de um método ou campo é o corpo inteiro da classe. Isso permite que métodos não `static` de uma classe utilizem os campos e outros métodos da classe.

Qualquer bloco pode conter declarações de variável. Se uma variável local ou um parâmetro em um método tiver o mesmo nome de um campo da classe, o campo permanece "oculto" até que o bloco termine a execução — isso é chamado **sombreamento**. No Capítulo 8, discutiremos como acessar campos "sombreados".

Erro de programação comum 6.10

Um erro de compilação ocorre quando uma variável local é declarada mais de uma vez em um método.

Dica de prevenção de erro 6.3

Utilize nomes diferentes para campos e variáveis locais para ajudar a evitar erros de lógica sutis que ocorrem quando um método é chamado e uma variável local do método sombreia um campo da classe.

O aplicativo nas figuras 6.11 e 6.12 demonstra as questões de escopo para campos e variáveis locais. Quando o aplicativo inicia a execução, o método `main` da classe `ScopeTest` (Figura 6.12, linhas 7–11) cria um objeto da classe `Scope` (linha 9) e chama o método `begin` do objeto (linha 11) para produzir a saída do programa (mostrada na Figura 6.12).

```
1  // Figura 6.11: Scope.java
2  // A classe Scope demonstra os escopos de campo e de variável local.
3
4  public class Scope
5  {
6     // campo acessível para todos os métodos dessa classe
7     private int x = 1;
8
9     // método begin cria e inicializa a variável local x
10    // e chama os métodos useLocalVariable e useField
11    public void begin()
12    {
13       int x = 5; // variável local x do método sombreia o campo x
14
15       System.out.printf( "local x in method begin is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable tem uma variável local x
18       useField(); // useField utiliza o campo x da classe Scope
19       useLocalVariable(); // useLocalVariable reinicializa a variável local x
20       useField(); // campo x da classe Scope retém seu valor
21
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // fim do método begin
24
25    // cria e inicializa a variável local x durante cada chamada
26    public void useLocalVariable()
27    {
28       int x = 25; // inicializa toda vez que useLocalVariable é chamado
29
30       System.out.printf(
31          "\nlocal x on entering method useLocalVariable is %d\n", x );
```

```
32          ++x; // modifica a variável local x desse método
33          System.out.printf(
34             "local x before exiting method useLocalVariable is %d\n", x );
35       } // fim do método useLocalVariable
36
37       // modifica o campo x da classe Scope durante cada chamada
38       public void useField()
39       {
40          System.out.printf(
41             "\nfield x on entering method useField is %d\n", x );
42          x *= 10; // modifica o campo x da classe Scope
43          System.out.printf(
44             "field x before exiting method useField is %d\n", x );
45       } // fim do método useField
46    } // fim da classe Scope
```

Figura 6.11 | A classe Scope demonstrando os escopos de um campo e de variáveis locais.

```
1  // Figura 6.12: ScopeTest.java
2  // Aplicativo para testar a classe Scope.
3
4  public class ScopeTest
5  {
6     // ponto de partida do aplicativo
7     public static void main( String[] args )
8     {
9        Scope testScope = new Scope();
10       testScope.begin();
11    } // fim de main
12 } // fim da classe ScopeTest
```

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Figura 6.12 | Aplicativo para testar a classe Scope.

Na classe Scope, a linha 7 declara e inicializa o campo x como 1. Esse campo permanece sombreado (oculto) em qualquer bloco (ou método) que declara uma variável local chamada x. O método begin (linhas 11–23) declara uma variável local x (linha 13) e a inicializa para 5. O valor dessa variável local é gerado para mostrar que o campo x (cujo valor é 1) permanece sombreado no método begin. O programa declara outros dois métodos — useLocalVariable (linhas 26–35) e useField (linhas 38–45) — que não recebem argumentos e não retornam resultados. O método begin chama cada método duas vezes (linhas 17–20). O método useLocalVariable declara a variável local x (linha 28). Quando useLocalVariable é chamado pela primeira vez (linha 17), ele cria a variável local x e a inicializa como 25 (linha 28), gera a saída do valor de x (linhas 30–31), incrementa x (linha 32) e gera a saída do valor de x novamente (linhas 33–34). Quando uselLocalVariable é chamado uma segunda vez (linha 19), ele recria a variável local x e a reinicializa como 25, assim, a saída de cada chamada useLocalVariable é idêntica.

O método useField não declara nenhuma variável local. Portanto, quando ele se refere a x, é o campo x (linha 7) da classe que é utilizado. Ao ser chamado pela primeira vez (linha 18), o método useField gera saída do valor (1) do campo x (linhas 40–41), multiplica o campo x por 10 (linha 42) e gera a saída do valor (10) do campo x novamente (linhas 43–44) antes de retornar. A próxima vez que o método useField for chamado (linha 20), o campo conterá seu valor modificado (10), o método gerará saída de 10 e, então, 100. Por fim, no método begin, o programa gera saída do valor da variável local x novamente (linha 22) para mostrar que nenhum método chama a variável local x do begin modificado, pois todos os métodos se referiram às variáveis identificadas como x nos outros escopos.

6.12 Sobrecarga de método

Os métodos com o mesmo nome podem ser declarados na mesma classe, contanto que tenham diferentes conjuntos de parâmetros (determinado pelo número, tipos e ordem dos parâmetros) — isso é chamado **sobrecarga de método**. Quando um método sobrecarregado é chamado, o compilador Java seleciona o método adequado examinando o número, os tipos e a ordem dos argumentos na chamada. A sobrecarga de métodos é comumente utilizada para criar vários métodos com o mesmo nome que realizam as mesmas tarefas, ou tarefas semelhantes, mas sobre tipos diferentes ou números diferentes de argumentos. Por exemplo, os métodos `Math` `abs`, `min` e `max` (resumidos na Seção 6.3) são sobrecarregados com quatro versões:

1. uma com dois parâmetros `double`;
2. uma com dois parâmetros `float`;
3. uma com dois parâmetros `int`;
4. uma com dois parâmetros `long`.

Nosso próximo exemplo demonstra como declarar e invocar métodos sobrecarregados. Apresentaremos exemplos de construtores sobrecarregados no Capítulo 8.

Declarando métodos sobrecarregados

Na nossa classe `MethodOverload` (Figura 6.13), incluímos duas versões sobrecarregadas de um método chamado `square` — uma que calcula o quadrado de um `int` (e retorna um `int`) e outro que calcula o quadrado de um `double` (e retorna um `double`). Embora esses métodos tenham o mesmo nome e listas e corpos semelhantes de parâmetros, pense neles simplesmente como métodos *diferentes*. Talvez ajude pensar nos nomes dos métodos como as "`square` de `int`" e "`square` de `double`", respectivamente. Quando o aplicativo inicia a execução, o método `main` da classe `MethodOverloadTest` (Figura 6.14, linhas 6–10) cria um objeto da classe `MethodOverload` (linha 8) e chama o método `testOverloadedMethods` do objeto (linha 9) para gerar a saída (Figura 6.14).

```
1  // Figura 6.13: MethodOverload.java
2  // Declarações de métodos sobrecarregados.
3
4  public class MethodOverload
5  {
6     // teste de métodos square sobrecarregados
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // fim do método testOverloadedMethods
12
13    // método square com argumento int
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // fim do método square com argumento int
20
21    // método square com argumento double
22    public double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // fim do método square com argumento double
28 } // fim da classe MethodOverload
```

Figura 6.13 | Declarações de métodos sobrecarregados.

```
1  // Figura 6.14: MethodOverloadTest.java
2  // Aplicativo para testar a classe MethodOverload.
3
4  public class MethodOverloadTest
5  {
6     public static void main( String[] args )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // fim de main
11 } // fim da classe MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Figura 6.14 | Aplicativo para testar a classe `MethodOverload`.

Na Figura 6.13, a linha 9 invoca o método `square` com o argumento `7`. Valores literais inteiros são tratados como um tipo `int`, assim a chamada de método na linha 9 invoca a versão de `square` nas linhas 14–19 que especifica um parâmetro `int`. De maneira semelhante, a linha 10 invoca o método `square` com o argumento `7,5`. Valores de ponto flutuante literais são tratados como um tipo `double`, dessa forma a chamada de método na linha 10 invoca a versão de `square` nas linhas 22–27 que especifica um parâmetro `double`. Cada método primeiro gera a saída de uma linha de texto para provar que o método adequado foi chamado em cada caso. Observe que os valores nas linhas 10–24 são exibidos com o especificador de formato `%f` e que não especificamos uma precisão em nenhum caso. Por padrão, valores de ponto flutuante são exibidos com seis dígitos de precisão se a precisão não for especificada no especificador de formato.

Distinguindo entre métodos sobrecarregados

O compilador distingue os métodos sobrecarregados pelas suas **assinaturas** — uma combinação do nome e número do método, tipos e ordem dos seus parâmetros. Se o compilador examinasse somente os nomes do método durante a compilação, o código na Figura 6.13 seria ambíguo — o compilador não saberia distinguir entre os dois métodos `square` (linhas 14–19 e 22–27). Internamente, o compilador utiliza nomes de método mais longos que incluem o nome original do método, os tipos de cada parâmetro e a ordem exata dos parâmetros para determinar se os métodos em uma classe são únicos nessa classe.

Por exemplo, na Figura 6.13, o compilador utilizaria o nome lógico "`square` de `int`" para o método `square` que especifica um parâmetro `int` e "`square` de `double`" para o método `square` que especifica um parâmetro `double` (os nomes reais que o compilador utiliza são mais confusos). Se a declaração do `method1` iniciar como

```
void method1( int a, float b )
```

o compilador, então, poderia utilizar o nome lógico "`method1` de `int` e `float`". Se os parâmetros forem especificados como

```
void method1( float a, int b )
```

o compilador, então, poderia utilizar o nome lógico "`method1` de `float` e `int`". Observe que a ordem dos tipos de parâmetros é importante, pois o compilador considera os dois cabeçalhos do `method1` anterior como sendo distintos.

Tipos de retorno dos métodos sobrecarregados

Na discussão sobre os nomes lógicos dos métodos utilizados pelo compilador, não mencionamos os tipos de retorno dos métodos. *As chamadas de método não podem ser distinguidas por tipo de retorno*. Se você tivesse sobrecarregado métodos que se diferenciassem apenas por seus tipos de retorno e chamasse um dos métodos em uma instrução autônoma como em:

```
square( 2 );
```

o compilador não seria capaz de determinar a versão do método a chamar, porque o valor de retorno é ignorado. A Figura 6.15 ilustra os erros de compilador gerados quando dois métodos têm a mesma assinatura e diferentes tipos de retorno. Métodos sobrecarregados podem ter diferentes tipos de retorno se os métodos tiverem diferentes listas de parâmetro. Além disso, métodos sobrecarregados não precisam ter o mesmo número de parâmetros.

Erro de programação comum 6.11

Declarar métodos sobrecarregados com listas de parâmetros idênticas é um erro de compilação independentemente de os tipos de retorno serem diferentes.

```
1  // Figura 6.15: MethodOverloadError.java
2  // Métodos sobrecarregados com assinaturas idênticas
3  // resultam em erros de compilação, mesmo que os tipos de retorno sejam diferentes.
4
5  public class MethodOverloadError
6  {
7     // declaração do método square com argumento int
8     public int square( int x )
9     {
10       return x * x;
11    }
12
13    // segunda declaração do método square com argumento int resulta
14    // em erros de compilação mesmo que os tipos de retorno sejam diferentes
15    public double square( int y )
16    {
17       return y * y;
18    }
19 } // fim da classe MethodOverloadError
```

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Figura 6.15 | Declarações de métodos sobrecarregados com assinaturas idênticas causam erros de compilação, mesmo que os tipos de retorno sejam diferentes.

6.13 (Opcional) Estudo de caso de GUI e imagens gráficas: cores e formas preenchidas

Embora você possa criar muitos designs interessantes apenas com linhas e formas básicas, a classe `Graphics` fornece várias outras capacidades. Os próximos dois recursos que introduziremos são cores e formas preenchidas. Acrescentar cor enriquece os desenhos que um usuário vê na tela do computador. As formas podem ser preenchidas com cores sólidas.

As cores exibidas na tela dos computadores são definidas pelos componentes vermelho, verde e azul (chamados de **valores RGB**) que têm valores inteiros de 0 a 255. Quanto mais alto o valor de uma cor componente, mais rica será a tonalidade na cor final. O Java utiliza a classe **`Color`** do pacote `java.awt` para representar cores que usam valores RGB. Por conveniência, a classe `Color` contém 13 objetos `static Color` predefinidos (`java.sun.com/javase/6/docs/api/java/awt/Color.html`) — `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` e `YELLOW`. Cada objeto pode ser acessado por meio do nome da classe e um ponto (`.`) como em `Color.RED`. A classe `Color` também contém um construtor na forma:

```
public Color( int r, int g, int b )
```

portanto, é possível criar cores personalizadas especificando os valores componentes vermelho, verde e azul.

Os métodos **`fillRect`** e **`fillOval`** da classe `Graphics` desenham ovais e retângulos preenchidos, respectivamente. Esses métodos têm os mesmos parâmetros que `drawRect` e `drawOval`; os dois primeiros são as coordenadas do canto superior esquerdo da forma, enquanto os dois seguintes determinam a largura e a altura. O exemplo nas figuras 6.16 e 6.17 demonstra cores e formas preenchidas desenhando e exibindo um rosto amarelo sorridente na tela.

```
1  // Figura 6.16: DrawSmiley.java
2  // Demonstra formas preenchidas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // desenha o rosto
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // desenha os olhos
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // desenha a boca
23       g.fillOval( 50, 110, 120, 60 );
24
25       // "retoca" a boca para criar um sorriso
26       g.setColor( Color.YELLOW );
27       g.fillRect( 50, 110, 120, 30 );
28       g.fillOval( 50, 120, 120, 40 );
29    } // fim do método paintComponent
30 } // fim da classe DrawSmiley
```

Figura 6.16 | Desenhando um rosto sorridente com cores e formas preenchidas.

As instruções `import` nas linhas 3–5 da Figura 6.16 importam as classes `Color`, `Graphics` e `JPanel`. A classe `DrawSmiley` (linhas 7–30) utiliza a classe `Color` para especificar as cores do desenho, e utiliza a classe `Graphics` para desenhar. Mais uma vez, a classe `JPanel` fornece a área em que desenhamos. A linha 14 no método `paintComponent` utiliza o método `Graphics` **setColor** para configurar a cor atual de desenho como `Color.YELLOW`. O método `setColor` requer um argumento, `Color` para configurar a cor de desenho. Nesse caso, utilizamos o objeto predefinido `Color.YELLOW`. A linha 15 desenha um círculo com um diâmetro de 200 para representar o rosto — se os argumentos de largura e altura forem idênticos, o método `fillOval` desenhará um círculo. Em seguida, a linha 18 configura a cor como `Color.Black` e as linhas 19–20 desenham os olhos. A linha 23 desenha a boca como uma oval, mas isso não é bem o que nós queremos. Para criar um rosto feliz, vamos "retocar" a boca. A linha 26 configura a cor como `Color.YELLOW`, portanto quaisquer formas que desenhamos serão combinadas com o rosto. A linha 27 desenha um retângulo com metade da altura da boca. Isso "apaga" a metade superior da boca, deixando somente a metade inferior. Para criar um sorriso melhor, a linha 28 desenha uma outra oval para cobrir levemente a parte superior da boca. A classe `DrawSmileyTest` (Figura 6.17) cria e exibe um `JFrame` contendo o desenho. Quando o `JFrame` é exibido, o sistema chama o método `paintComponent` para desenhar o rosto sorridente.

```
1  // Figura 6.17: DrawSmileyTest.java
2  // Aplicativo de teste que exibe um rosto sorridente.
3  import javax.swing.JFrame;
4
5  public class DrawSmileyTest
6  {
7     public static void main( String[] args )
8     {
9        DrawSmiley panel = new DrawSmiley();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 230, 250 );
15       application.setVisible( true );
16    } // fim de main
17 } // fim da classe DrawSmileyTest
```

Figura 6.17 | Criando `JFrame` para exibir um rosto sorridente.

Exercícios do Estudo de caso sobre GUIs e imagens gráficas

6.1 Utilizando o método `fillOval`, desenhe um alvo que alterna entre duas cores aleatórias, como na Figura 6.18. Utilize o construtor `Color( int r, int g, int b)` com argumentos aleatórios para gerar cores aleatórias.

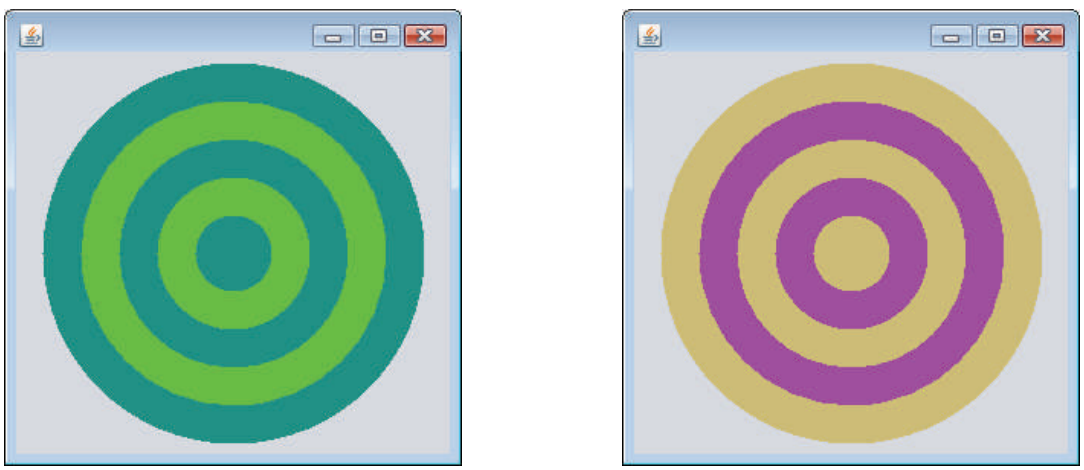


Figura 6.18 | Um alvo com duas cores aleatórias alternativas.

6.2 Crie um programa que desenhe 10 formas preenchidas aleatórias com cores aleatórias, posições e tamanhos (Figura 6.19). O método `paintComponent` deve conter um loop que itera 10 vezes. Em cada iteração, o loop deve determinar se deve ser desenhado um retângulo ou uma oval preenchida, criar uma cor aleatória e escolher as coordenadas e dimensões aleatoriamente. As coordenadas devem ser escolhidas com base na largura e altura do painel. O comprimento dos lados deve estar limitado à metade da largura ou altura da janela.

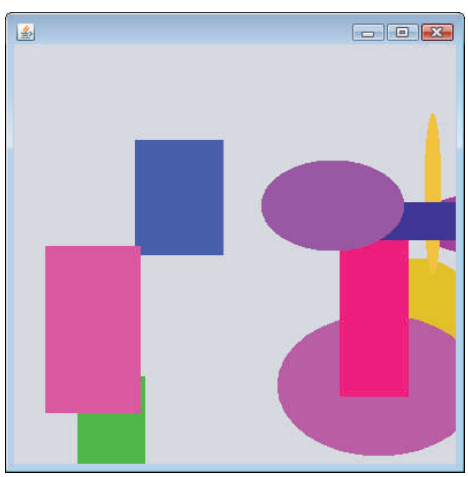


Figura 6.19 | Formas geradas aleatoriamente.

6.14 Conclusão

Neste capítulo, você aprendeu mais sobre declarações de método. Você também aprendeu a diferença entre métodos não `static` e `static` e como chamar métodos `static` precedendo o nome do método com o nome da classe em que ele aparece e com o separador de ponto (`.`). Você aprendeu a utilizar os operadores + e += para realizar concatenações de strings. Discutimos como a pilha de chamadas de método e os registros de ativação monitoram os métodos que foram chamados e para onde cada método deve retornar quando ele completa sua tarefa. Também discutimos as regras de promoção do Java para converter implicitamente entre tipos primitivos e como realizar conversões explícitas com operadores de coerção. Em seguida, você aprendeu sobre alguns dos pacotes mais utilizados na Java API.

Você observou como declarar constantes identificadas utilizando tanto tipos `enum` como variáveis `public static final`. Você utilizou a classe `Random` para gerar números aleatórios para simulações. Você também aprendeu o escopo dos campos e variáveis locais em uma classe. Por fim, você aprendeu que múltiplos métodos em uma classe podem ser sobrecarregados fornecendo ao método o mesmo nome e assinaturas diferentes. Esses métodos podem ser utilizados para realizar as mesmas tarefas, ou tarefas semelhantes, utilizando tipos diferentes ou números diferentes de parâmetros.

No Capítulo 7, você aprenderá a manter listas e tabelas de dados em arrays. Você verá uma implementação mais elegante do aplicativo que lança um dado 6.000 vezes e duas versões aprimoradas do nosso estudo de caso `GradeBook` que você estudou nos capítulos 3–5. Você também aprenderá a acessar os argumentos de linha de comando do aplicativo que são passados para o método `main` quando um aplicativo inicia a execução.

Resumo

Seção 6.1 Introdução

- A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas e simples partes, ou módulos. Essa técnica se chama dividir para conquistar.

Seção 6.2 Módulos de programa em Java

- Os métodos são declarados dentro de classes. Em geral, as classes são agrupadas em pacotes para que possam ser importadas e reutilizadas.
- Os métodos permitem modularizar um programa separando suas tarefas em unidades autocontidas. As instruções em um método são escritas somente uma vez e permanecem ocultas de outros métodos.
- Utilizar os métodos existentes como blocos de construção para criar novos programas é uma forma de reutilização de softwares que permite evitar a repetição de código dentro de um programa.

Seção 6.3 Métodos `static`, campos `static` e classe `Math`

- Uma chamada de método especifica o nome do método a ser chamado e fornece os argumentos que o método chamado requer para realizar sua tarefa. Quando a chamada de método é concluída, o método retorna um resultado, ou simplesmente o controle, ao seu chamador.
- Uma classe pode conter métodos `static` para realizar tarefas comuns que não exigem um objeto da classe. Quaisquer dados que um método `static` poderia requer para realizar suas tarefas podem ser enviados ao método como argumentos em uma chamada de método. Um método `static` é chamado especificando o nome da classe em que o método é declarado seguido por um ponto (`.`) e pelo nome do método, como em:

  NomeDaClasse`.`*nomeDoMétodo*`(` *argumentos* `)`
- A classe `Math` fornece os métodos `static` para realizar cálculos matemáticos comuns.
- A constante `Math.PI` (3.141592653589793) é a relação entre a circunferência de um círculo e seu diâmetro. A constante `Math.E` (2.718281828459045) é o valor base para logaritmos naturais (calculados com o método `static` `Math` `log`).
- `Math.PI` e `Math.E` são declaradas com os modificadores `public`, `final` e `static`. Torná-los `public` permite que você use esses campos nas suas próprias classes. Um campo declarado com a palavra-chave `final` é constante — seu valor não pode ser alterado depois de ele ser inicializado. Tanto `PI` como `E` são declarados `final` porque seus valores nunca mudam. Tornar esses campos `static` permite que eles sejam acessados pelo nome da classe `Math` e um ponto (`.`) separador, como ocorre com os métodos da classe `Math`.
- Todos os objetos de uma classe compartilham uma cópia dos campos `static` da classe. As variáveis de classe e as variáveis de instância representam os campos de uma classe.
- Quando você executa a Java Virtual Machine (JVM) com o comando `java`, a JVM carrega a classe que você especifica e utiliza esse nome de classe para invocar o método `main`. Você pode especificar argumentos de linha de comando adicionais que a JVM passará para seu aplicativo.
- Você pode colocar um método `main` em cada classe que você declara — somente o método `main` na classe que você utiliza para executar o aplicativo será chamado pelo comando `java`.

Seção 6.4 Declarando métodos com múltiplos parâmetros

- Quando um método é chamado, o programa faz uma cópia dos valores de argumento do método e os atribui aos parâmetros correspondentes do método. Quando o controle do programa retorna ao ponto no programa em que método foi chamado, os parâmetros do método são removidos da memória.
- Um método pode retornar no máximo um valor, mas o valor retornado poderia ser uma referência a um objeto que contém muitos valores.
- Variáveis devem ser declaradas como campos de uma classe somente se forem utilizadas em mais de um método da classe ou se o programa salvar seus valores entre chamadas aos métodos da classe.

Seção 6.5 Notas sobre a declaração e utilização de métodos

- Há três maneiras de chamar um método — usar o próprio nome de um método para chamar um outro método da mesma classe; utilizar uma variável que contém uma referência a um objeto, seguido por um ponto (`.`) e nome do método para chamar um método do objeto referenciado; e utilizar o nome da classe e um ponto (`.`) para chamar um método `static` de uma classe.
- Há três maneiras de retornar o controle a uma instrução que chama um método. Se o método não retornar um resultado, o controle retornará quando o fluxo do programa alcançar a chave direita de fechamento do método ou quando a instrução:

  ```
  return;
  ```

  for executada. Se o método retornar um resultado, a instrução:

  ```
  return expressão;
  ```

  avalia a *expressão* e então imediatamente retorna o valor resultante ao chamador.
- Se um método tiver mais de um parâmetro, os parâmetros serão especificados como uma lista separada por vírgulas. Deve haver um argumento na chamada de método para cada parâmetro na declaração do método. Além disso, cada argumento deve ser consistente com o tipo do parâmetro correspondente. Se um método não aceitar argumentos, a lista de parâmetros estará vazia.
- `String`s podem ser concatenadas com o operador `+`, o que posiciona os caracteres do operando direito no final daqueles no operando esquerdo.
- Todos os objetos e valores primitivos em Java têm uma representação de `String`. Quando um objeto é concatenado com uma `String`, o objeto é convertido em uma `String` e então as duas `String`s são concatenadas.
- Se um `boolean` for concatenado com uma `String`, a palavra `"true"` ou `"false"` é utilizada para representar o valor `boolean`.
- Todos os objetos em Java têm um método especial chamado `toString` que retorna uma representação `String` do conteúdo do objeto. Quando um objeto é concatenado com uma `String`, a JVM chama implicitamente o método `toString` do objeto a fim de obter a representação string do objeto.
- Os programadores às vezes dividem grandes literais `String` em várias `String`s menores e as colocam em múltiplas linhas de código para melhorar a legibilidade, depois remontam as `String`s utilizando concatenação.

Seção 6.6 Pilha de chamadas de método e registros de ativação

- As pilhas são conhecidas como estruturas de dados do tipo último a entrar e primeiro a sair (*last-in, first-out — LIFO*) — o último item inserido na pilha é o primeiro item que é removido da pilha.
- Um método chamado deve saber como retornar ao seu chamador, portanto o endereço de retorno do método de chamada é colocado na pilha de execução de programas quando o método é chamado. Se uma série de chamadas de método ocorrer, os sucessivos endereços de retorno são empilhados na ordem "último a entrar, primeiro a sair" de modo que o último método a executar será o primeiro a retornar ao seu chamador.
- A pilha de execução de programas contém a memória para as variáveis locais utilizadas em cada invocação de método durante a execução de um programa. Esses dados são conhecidos como registro de ativação ou quadro de pilha da chamada de método. Quando uma chamada de método é feita, o registro de ativação dessa chamada de método é inserido na pilha de execução de programas. Quando o método retorna ao seu chamador, sua chamada do registro de ativação é retirada da pilha e as variáveis locais não são mais conhecidas para o programa. Se essa variável local armazena a única referência a um objeto, o objeto não pode mais ser acessado pelo programa e acabará sendo excluído da memória durante a "coleta de lixo".
- A quantidade de memória em um computador é finita, portanto somente certa quantidade de memória pode ser utilizada para armazenar registros de ativação na pilha de execução do programa. Se houver um número maior de chamadas de método do que seus registros de ativação podem armazenar na pilha de execução do programa, ocorrerá um erro conhecido como estouro de pilha. O aplicativo compilará corretamente, mas sua execução causa um estouro de pilha.

Seção 6.7 Promoção e coerção de argumentos

- A promoção de argumento converte o valor de um argumento para o tipo que o método espera receber no parâmetro correspondente.
- Regras de promoção se aplicam a expressões que contenham valores de dois ou mais tipos primitivos e a valores de tipo primitivo passados como argumentos para os métodos. Cada valor é promovido para o tipo "mais alto" na expressão. Em casos em que as informações podem ser perdidas devido à conversão, o compilador Java exige que você utilize um operador de coerção para forçar explicitamente que a conversão ocorra.

Seção 6.9 Estudo de caso: geração de números aleatórios

- Os objetos da classe `Random` (pacote `java.util`) podem produzir valores `int`, `long`, `float` ou `double` aleatórios. O método `Math` `random` pode produzir valores `double` no intervalo $0.0 \leq x < 1.0$, onde x é o valor retornado pelo método `random`.
- O método `Random` `nextInt` gera um valor aleatório `int` no intervalo entre –2.147.483.648 e +2.147.483.647. Os valores retornados por `nextInt` são na verdade números pseudoaleatórios — uma sequência de valores produzida por um cálculo matemático complexo. Esse cálculo utiliza a hora do dia atual para semear o gerador de números aleatórios de uma maneira que cada execução de um programa fornece uma sequência diferente de valores aleatórios.
- A classe `Random` fornece uma outra versão do método `nextInt` que recebe um argumento `int` e retorna um valor a partir de 0, mas sem incluí-lo, até o valor do argumento.
- Os números aleatórios em um intervalo podem ser gerados com:

  ```
  number = valorDeDeslocamento + randomNumbers.nextInt( fatorDeEscalonamento );
  ```

  onde *valorDeDeslocamento* especifica o primeiro número no intervalo desejado de inteiros consecutivos e *fatorDeEscalonamento* especifica quantos números estão no intervalo.

- Os números aleatórios podem ser escolhidos a partir de intervalos de inteiro não consecutivos, como em:

```
number = valorDeDeslocamento +
   diferençaEntreValores * randomNumbers.nextInt( fatorDeEscalonamento );
```

  onde *valorDeDeslocamento* especifica o primeiro número no intervalo de valores, *diferençaEntreValores* representa a diferença entre números consecutivos na sequência e *fatorDeEscalonamento* especifica quantos números estão no intervalo.
- Para fins de depuração, às vezes é útil repetir a sequência de números pseudoaleatórios durante cada execução de programa. Para fazer isso, passe um valor inteiro `long` para o construtor de objeto `Random`. Utilizar a mesma semente toda vez que o programa executado produz a mesma sequência de números.

Seção 6.10 Estudo de caso: um jogo de azar; introdução a enumerações

- Uma enumeração é introduzida pela palavra-chave `enum` e um nome de tipo. Como com qualquer classe, as chaves (`{` e `}`) delimitam o corpo de uma declaração `enum`. Dentro das chaves há uma lista separada por vírgulas de constantes de enumeração, cada uma representando um valor único. Os identificadores em uma `enum` devem ser únicos. Pode-se atribuir variáveis de um tipo `enum` somente a constantes do tipo `enum`.
- Constantes também podem ser declaradas como variáveis `public final static`. Essas constantes, por convenção, são declaradas com todas as letras maiúsculas fazendo com que elas se destaquem no programa.

Seção 6.11 Escopo das declarações

- O escopo é a parte do programa em que uma entidade, como uma variável ou um método, pode ser referida pelo seu nome. Diz-se que essa entidade está "no escopo" para essa parte do programa.
- O escopo de uma declaração de parâmetro é o corpo do método em que a declaração aparece.
- O escopo de uma declaração de variável local é do ponto em que a declaração aparece até o final desse bloco.
- O escopo de uma declaração de variável local que aparece na seção de inicialização do cabeçalho de uma instrução `for` é o corpo da instrução `for` e as outras expressões no cabeçalho.
- O escopo de um método ou campo de uma classe é o corpo inteiro da classe. Isso permite que os métodos da classe utilizem nomes simples para chamar os outros métodos da classe e acessem os campos da classe.
- Qualquer bloco pode conter declarações de variável. Se uma variável local ou um parâmetro em um método tiver o mesmo nome de um campo, o campo permanece "sombreado" até que o bloco termine a execução.

Seção 6.12 Sobrecarga de método

- O Java permite que vários métodos com o mesmo nome sejam declarados em uma classe, contanto que os métodos tenham conjuntos diferentes de parâmetros (determinados pelo número, ordem e tipos dos parâmetros). Essa técnica é chamada sobrecarga de método.
- Métodos sobrecarregados são distinguidos por suas assinaturas — combinações dos nomes dos métodos e números, tipos e ordem dos seus parâmetros. Os métodos não podem ser distinguidos por tipo de retorno.

Terminologia

Abstract Window Toolkit Event Package, 163
assinatura de um método, 175
biblioteca de classes Java, 155
capacidade de reutilização de software, 156
colocar em uma pilha, 161
`Color`, classe, 176
concatenação, 160
concatenação de string, 160
constante de enumeração, 171
deslocar (números aleatórios), 165
dividir para conquistar, abordagem, 155
elemento chance, 164
`enum`, palavra-chave, 171
enumeração, 171
escopo de uma declaração, 172
estouro de pilha, 162
excluir uma pilha, 162
`fillOval`, método da classe `Graphics`, 176
`fillRect`, método da classe `Graphics`, 176
função, 156
interface de programas aplicativos (*applications programming interface* — API), 155
Java Abstract Window Toolkit (AWT), pacote, 163
Java Abstract Window Toolkit Event, pacote, 163
Java API, 155
Java Applet Package, 163
Java Application Programming Interface (Java API), 155
Java Concurrency Package, 164
Java Input/Output Package, 163
Java Language Package, 163
Java Media Framework Package, 164
Java Networking Package, 163
Java Swing Event Package, 164
Java Swing GUI Components Package, 164
Java Text Package, 163
Java Utilities Package, 163
JAX-WS, pacote, 164
JDBC, pacote, 163
LIFO (*Last-In, First-Out*), 162
linha de comando, argumento, 158
`Math.E`, 157
`Math.PI`, 157
método de classe, 156
módulo, 155
`nextInt`, método da classe `Random`, 164
número pseudoaleatório, 164
pacotes da Java API, 163
parâmetro formal, 159
pilha, 161
pilha de chamadas de método, 162
pilha de execução do programa, 162
procedimento, 156
promoção de argumentos, 162
quadro de pilha, 162
`Random`, classe, 164
registro de ativação, 162
regras de promoção para tipos primitivos, 162
RGB, valores, 176
semear números aleatórios, 164
`setColor`, método da classe `Graphics`, 177
sobrecarga de método, 174
sobrecarregar um método, 174
sombrear um campo, 172
último a entrar, primeiro a sair (*Last-In, First-Out* — LIFO), 162
valor de deslocamento, 165
variável de classe, 158

Exercícios de autorrevisão

6.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um método é invocado com um(a) _____.

b) Uma variável conhecida somente dentro do método em que é declarada chama-se _____.

c) A instrução _____ em um método chamado pode ser usada para passar o valor de uma expressão de volta para o método de chamada.

d) A palavra-chave _____ indica que um método não retorna um valor.

e) Os dados podem ser adicionados ou removidos somente do(a) _____ de uma pilha.

f) As pilhas são conhecidas como _____ estruturas de dados — o último item colocado (inserido) na pilha é o primeiro item retirado (removido) da pilha.

g) As três maneiras de retornar controle de um método chamado a um chamador são _____, _____ e _____.

h) Um objeto da classe _____ produz números aleatórios.

i) A pilha de execução de programas contém a memória criada para variáveis locais a cada invocação de método durante a execução de um programa. Esses dados, armazenados como uma parte da pilha de execução de programas, são conhecidos como _____ ou _____ da chamada de método.

j) Se houver mais chamadas de método do que pode ser armazenado na pilha de execução de programas, um erro conhecido como _____ ocorrerá.

k) O _____ de uma declaração é a parte de um programa que pode referenciar a entidade na declaração pelo nome.

l) É possível ter diversos métodos com o mesmo nome que operam, separadamente, sobre diferentes tipos ou números de argumentos. Esse recurso é chamado de método _____.

m) A pilha de execução de programas também é chamada pilha de _____.

6.2 Para a classe `Craps` na Figura 6.9, declare o escopo de cada uma das seguintes entidades:

a) a variável `randomNumbers`.

b) a variável `die1`.

c) o método `rollDice`.

d) o método `play`.

e) a variável `sumOfDice`.

6.3 Escreva um aplicativo que teste se os exemplos de chamadas de método da classe `Math` mostrada na Figura 6.2 realmente produz os resultados indicados.

6.4 Forneça o cabeçalho de método para cada um dos métodos a seguir.

a) O método `hypotenuse`, que aceita dois argumentos de ponto flutuante de precisão dupla `side1` e `side2` e retorna um resultado de ponto flutuante de dupla precisão.

b) O método `smallest`, que recebe três inteiros `x`, `y` e `z` e retorna um inteiro.

c) O método `instructions`, que não aceita nenhum argumento e não retorna um valor. [*Nota:* esses métodos são comumente utilizados para exibição de instruções para o usuário.]

d) O método `intToFloat`, que aceita um argumento inteiro `number` e retorna um resultado de ponto flutuante.

6.5 Encontre o erro em cada um dos seguintes segmentos de programa. Explique como corrigir o erro.

a)
```
void g()
{
   System.out.println( "Inside method g" );

   void h()
   {
      System.out.println( "Inside method h" );
   }
}
```

b)
```
int sum( int x, int y )
{
   int result;
   result = x + y;
}
```

c)
```
void f( float a );
{
   float a;
   System.out.println( a );
}
```

d)
```
void product()
{
   int a = 6, b = 5, c = 4, result;
   result = a * b * c;
   System.out.printf( "Result is %d\n", result );
   return result;
}
```

6.6 Escreva um aplicativo Java completo para solicitar ao usuário o raio (do tipo `double`) de uma esfera e chame o método `sphereVolume` para calcular e exibir o volume da esfera. Utilize a seguinte instrução para calcular o volume:
```
double volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 )
```

Respostas dos exercícios de autorrevisão

6.1 a) chamada de método. b) variável local. c) `return`. d) `void`. e) parte superior. f) último a entrar, primeiro a sair (LIFO). g) `return;` ou `return` expressão; ou encontrar a chave direita de fechamento de um método. h) `Random`. i) registro de ativação, quadro de pilha. j) estouro de pilha. k) escopo. l) método sobrecarregado. m) chamada de método.

6.2 a) corpo de classe. b) bloco que define o corpo do método `rollDice`. c) corpo de classe. d) corpo de classe. e) bloco que define o corpo do método `play`.

6.3 A seguinte solução demonstra os métodos da classe `Math` na Figura 6.2:

```
1  // Exercícios 6.3: MathTest.java
2  // Testando os métodos da classe Math
3
4  public class MathTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "Math.abs( 23.7 ) = %f\n", Math.abs( 23.7 ) );
9        System.out.printf( "Math.abs( 0.0 ) = %f\n", Math.abs( 0.0 ) );
10       System.out.printf( "Math.abs( -23.7 ) = %f\n", Math.abs( -23.7 ) );
11       System.out.printf( "Math.ceil( 9.2 ) = %f\n", Math.ceil( 9.2 ) );
12       System.out.printf( "Math.ceil( -9.8 ) = %f\n", Math.ceil( -9.8 ) );
13       System.out.printf( "Math.cos( 0.0 ) = %f\n", Math.cos( 0.0 ) );
14       System.out.printf( "Math.exp( 1.0 ) = %f\n", Math.exp( 1.0 ) );
15       System.out.printf( "Math.exp( 2.0 ) = %f\n", Math.exp( 2.0 ) );
16       System.out.printf( "Math.floor( 9.2 ) = %f\n", Math.floor( 9.2 ) );
17       System.out.printf( "Math.floor( -9.8 ) = %f\n",
18          Math.floor( -9.8 ) );
19       System.out.printf( "Math.log( Math.E ) = %f\n",
20          Math.log( Math.E ) );
21       System.out.printf( "Math.log( Math.E * Math.E ) = %f\n",
22          Math.log( Math.E * Math.E ) );
23       System.out.printf( "Math.max( 2.3, 12.7 ) = %f\n",
24          Math.max( 2.3, 12.7 ) );
25       System.out.printf( "Math.max( -2.3, -12.7 ) = %f\n",
26          Math.max( -2.3, -12.7 ) );
27       System.out.printf( "Math.min( 2.3, 12.7 ) = %f\n",
28          Math.min( 2.3, 12.7 ) );
29       System.out.printf( "Math.min( -2.3, -12.7 ) = %f\n",
30          Math.min( -2.3, -12.7 ) );
31       System.out.printf( "Math.pow( 2.0, 7.0 ) = %f\n",
32          Math.pow( 2.0, 7.0 ) );
33       System.out.printf( "Math.pow( 9.0, 0.5 ) = %f\n",
34          Math.pow( 9.0, 0.5 ) );
35       System.out.printf( "Math.sin( 0.0 ) = %f\n", Math.sin( 0.0 ) );
36       System.out.printf( "Math.sqrt( 900.0 ) = %f\n",
37          Math.sqrt( 900.0 ) );
38       System.out.printf( "Math.tan( 0.0 ) = %f\n", Math.tan( 0.0 ) );
39    } // fim de main
40 } // fim da classe MathTest
```

```
Math.abs( 23.7 ) = 23.700000
Math.abs( 0.0 ) = 0.000000
Math.abs( -23.7 ) = 23.700000
Math.ceil( 9.2 ) = 10.000000
Math.ceil( -9.8 ) = -9.000000
Math.cos( 0.0 ) = 1.000000
Math.exp( 1.0 ) = 2.718282
Math.exp( 2.0 ) = 7.389056
Math.floor( 9.2 ) = 9.000000
Math.floor( -9.8 ) = -10.000000
Math.log( Math.E ) = 1.000000
Math.log( Math.E * Math.E ) = 2.000000
Math.max( 2.3, 12.7 ) = 12.700000
Math.max( -2.3, -12.7 ) = -2.300000
Math.min( 2.3, 12.7 ) = 2.300000
Math.min( -2.3, -12.7 ) = -12.700000
Math.pow( 2.0, 7.0 ) = 128.000000
Math.pow( 9.0, 0.5 ) = 3.000000
Math.sin( 0.0 ) = 0.000000
Math.sqrt( 900.0 ) = 30.000000
Math.tan( 0.0 ) = 0.000000
```

6.4 a) `double hypotenuse( double side1, double side2 )`

b) `int smallest( int x, int y, int z )`

c) `void instructions()`

d) `float intToFloat( int number )`

6.5 a) Erro: o método `h` é declarado dentro do método `g`.

Correção: mova a declaração de `h` para fora da declaração de `g`.

b) Erro: o método supostamente deve retornar um inteiro, mas não o faz.

Correção: exclua a variável `result` e coloque a instrução

```
return x + y;
```

no método ou adicione a seguinte instrução no fim do corpo do método:

```
return result;
```

c) Erro: o ponto-e-vírgula após o parêntese direito da lista de parâmetros está incorreto e o parâmetro `a` não deve ser novamente declarado no método.

Correção: exclua o ponto-e-vírgula após o parêntese direito da lista de parâmetros e exclua a declaração `float a;`.

d) Erro: o método retorna um valor quando supostamente não deveria.

Correção: altere o tipo de retorno de `void` para `int`.

6.6 A solução a seguir calcula o volume de uma esfera, utilizando o raio inserido pelo usuário:

```
1  // Exercícios 6.6: Sphere.java
2  // Calcula o volume de uma esfera.
3  import java.util.Scanner;
4
5  public class Sphere
6  {
7     // obtém o raio a parti do usuário e exibe o volume da esfera
8     public void determineSphereVolume()
9     {
10       Scanner input = new Scanner( System.in );
11
12       System.out.print( "Enter radius of sphere: " );
13       double radius = input.nextDouble();
14
15       System.out.printf( "Volume is %f\n", sphereVolume( radius ) );
16    } // fim do método determineSphereVolume
17
18 // calcula e retorna volume de esfera
19    public double sphereVolume( double radius )
20    {
21       double volume = ( 4.0 / 3.0 ) * Math.PI * Math.pow( radius, 3 );
22       return volume;
23    } // fim do método sphereVolume
24 } // fim da classe Sphere
```

```
1  // Exercícios 6.6: SphereTest.java
2  // Calcula o volume de uma esfera.
3
```

```
4  public class SphereTest
5  {
6     // ponto de partida do aplicativo
7     public static void main( String[] args )
8     {
9        Sphere mySphere = new Sphere();
10       mySphere.determineSphereVolume();
11    } // fim de main
12 } // fim da classe SphereTest
```

```
Enter radius of sphere: 4
Volume is 268.082573
```

Exercícios

6.7 Qual é o valor de x depois que cada uma das seguintes instruções é executada?

a) `x = Math.abs( 7.5 );`

b) `x = Math.floor( 7.5 );`

c) `x = Math.abs( 0.0 );`

d) `x = Math.ceil( 0.0 );`

e) `x = Math.abs( -6.4 );`

f) `x = Math.ceil( -6.4 );`

g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

6.8 (***Taxa de estacionamento***) Um estacionamento cobra uma taxa mínima de R$ 2,00 para estacionar por até três horas. Um adicional de R$ 0,50 por hora *não necessariamente inteira* é cobrado após as três primeiras horas. A carga máxima para qualquer dado período de 24 horas é R$ 10,00. Assuma que nenhum carro fica estacionado por mais de 24 horas por vez. Escreva um aplicativo que calcule e exiba as taxas de estacionamento de cada cliente que estacionou nessa garagem ontem. Você deve inserir as horas de estacionamento para cada cliente. O programa deve exibir a cobrança para o cliente atual e calcular e exibir o total dos recibos de ontem. Ele deve utilizar o método `calculateCharges` para determinar a carga para cada cliente.

6.9 (***Arredondando números***) `Math.floor` pode ser utilizado para arredondar valores ao número inteiro mais próximo — p. ex.:

```
y = Math.floor( x + 0.5 );
```

arredondará o número x para o inteiro mais próximo e atribuirá o resultado a y. Escreva um aplicativo que lê valores `double` e utiliza a instrução anterior para arredondar cada um dos números para o inteiro mais próximo. Para cada número processado, exiba ambos os números, o original e o arredondado.

6.10 (***Arredondando números***) Para arredondar números em casas decimais específicas, utilize uma instrução como

```
y = Math.floor( x * 10 + 0.5 ) / 10;
```

que arredonda x para a casa decimal (isto é, a primeira posição à direita do ponto de fração decimal), ou

```
y = Math.floor( x * 100 + 0.5 ) / 100;
```

que arredonda x para a casa centesimal (isto é, a segunda posição à direita do ponto de fração decimal). Escreva um aplicativo que defina quatro métodos para arredondar um número x de várias maneiras:

a) `roundToInteger( number )`

b) `roundToTenths( number )`

c) `roundToHundredths( number )`

d) `roundToThousandths( number )`

Para cada leitura de valor, seu programa deve exibir o valor original, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

6.11 Responda cada uma das seguintes perguntas:

a) O que significa escolher números "aleatoriamente"?

b) Por que o método `nextInt` da classe `Random` é útil para simular jogos de azar?

c) Por que frequentemente é necessário escalonar ou deslocar os valores produzidos por um objeto `Random`?

d) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?

6.12 Escreva instruções que atribuem inteiros aleatórios à variável n nos seguintes intervalos:

a) $1 \leq n \leq 2$.

b) $1 \leq n \leq 100$.

c) $0 \leq n \leq 9$.

d) $1000 \leq n \leq 1112$.

e) $-1 \leq n \leq 1$.

f) $-3 \leq n \leq 11$.

6.13 Para cada um dos seguintes conjuntos de inteiros, escreva uma única instrução que exiba um número aleatoriamente a partir do conjunto.

a) 2, 4, 6, 8, 10.

b) 3, 5, 7, 9, 11.

c) 6, 10, 14, 18, 22.

6.14 (***Exponenciação***) Escreva um método `integerPower(base, exponent)` que retorne o valor de $base^{expoente}$

Por exemplo, `integerPower(3, 4)` calcula 3^4 (ou `3 * 3 * 3 * 3`). Assuma que `exponent` é um inteiro não zero, positivo e `base` é um inteiro. O método `integerPower` deve utilizar uma instrução `for` ou `while` para controlar o cálculo. Não utilize métodos da classe `Math`. Incorpore esse método a um aplicativo que lê os valores inteiros para `base` e `exponent` e realiza o cálculo com o método `integerPower`.

6.15 (***Cálculos da hipotenusa***) Defina um método `hypotenuse` que calcula a hipotenusa de um triângulo retângulo quando os comprimentos dos outros dois lados são dados. O método deve tomar dois argumentos do tipo `double` e retornar a hipotenusa como um `double`. Incorpore esse método a um aplicativo que lê valores para `side1` e `side2` e realiza o cálculo com o método `hypotenuse`. Utilize os métodos `Math pow` e `sqrt` para determinar o tamanho da hipotenusa de cada um dos triângulos na Figura 6.20. [*Nota:* a classe `Math` também fornece o método `hypot` para realizar esse cálculo.]

Triângulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Figura 6.20 | Valores para os lados dos triângulos no Exercício 6.15.

6.16 (***Múltiplos***) Escreva um método `isMultiple` que determina um par de inteiros se o segundo inteiro for um múltiplo do primeiro. O método deve aceitar dois argumentos inteiros e retornar `true` se o segundo for um múltiplo do primeiro e `false` caso contrário. [*Dica:* utilize o operador de módulo]. Incorpore esse método a um aplicativo que insere uma série de pares inteiros (um par por vez) e determina se o segundo valor em cada par é um múltiplo do primeiro.

6.17 (***Par ou ímpar***) Escreva um método `isEven` que utiliza o operador de resto (%) para determinar se um inteiro é par. O método deve levar um argumento inteiro e retornar `true` se o número inteiro for par, e `false`, caso contrário. Incorpore esse método a um aplicativo que insere uma sequência de inteiros (um por vez) e determina se cada um é par ou ímpar.

6.18 (***Exibindo um quadrado de asteriscos***) Escreva um método `squareOfAsterisks` que exibe um quadrado sólido (o mesmo número de linhas e colunas) de asteriscos cujo lado é especificado no parâmetro inteiro `side`. Por exemplo, se `side` for `4`, o método deverá exibir:

```
****
****
****
****
```

Incorpore esse método a um aplicativo que lê um valor inteiro para `side` a partir da entrada fornecida pelo usuário e gera saída dos asteriscos com o método `squareOfAsterisks`.

6.19 (***Exibindo um quadrado de qualquer caractere***) Modifique o método criado no Exercício 6.18 para receber um segundo parâmetro do tipo `char` chamado `fillCharacter`. Forme o quadrado utilizando o char fornecido como um argumento. Portanto, se `side` for `5` e `fillCharacter` for `#`, o método deve exibir:

```
#####
#####
#####
#####
#####
```

Utilize a seguinte instrução (em que `input` é um objeto `Scanner`) para ler um caractere do usuário no teclado:

```
char fill = input.next().charAt( 0 );
```

6.20 (***Área de círculo***) Escreva um aplicativo que solicite ao usuário o raio de um círculo e utilize um método chamado `circleArea` para calcular a área do círculo.

6.21 (***Separando dígitos***) Escreva métodos que realizam cada uma das seguintes tarefas:

a) Calcule a parte inteiro do quociente quando o inteiro `a` é dividido pelo inteiro `b`.

b) Calcule o resto inteiro quando o inteiro `a` é dividido pelo inteiro `b`.

c) Utilize os métodos desenvolvidos nas partes (a) e (b) para escrever um método `displayDigits` que recebe um inteiro entre `1` e `99999` e o exibe como uma sequência de dígitos, separando cada par de dígitos por dois espaços. Por exemplo, o inteiro `4562` deve aparecer como:

```
4  5  6  2
```

Incorpore os métodos em um aplicativo que insere um número inteiro e chama `displayDigits` passando para o método o número inteiro inserido. Exiba os resultados.

6.22 **(*Conversões de temperatura*)** Implemente os seguintes métodos inteiros:

a) O método `celsius` retorna o equivalente em Celsius de uma temperatura em Fahrenheit utilizando o cálculo:

```
celsius = 5.0 / 9.0 * ( fahrenheit - 32 );
```

b) O método `fahrenheit` retorna o equivalente em Fahrenheit de uma temperatura em Celsius utilizando o cálculo:

```
fahrenheit = 9.0 / 5.0 * celsius + 32;
```

c) Utilize os métodos nas partes (a) e (b) para escrever um aplicativo que permite ao usuário inserir uma temperatura em Fahrenheit e exibir o equivalente em Celsius ou inserir uma temperatura em Celsius e exibir o equivalente em Fahrenheit.

6.23 **(*Encontrar o mínimo*)** Escreva um método `minimum3` que retorna o menor de três números de ponto flutuante. Utilize o método `Math.min` para implementar `minimum3`. Incorpore o método a um aplicativo que lê três valores do usuário, determina o menor valor e exibe o resultado.

6.24 **(*Números perfeitos*)** Dizemos que um número inteiro é um *número perfeito* se a soma de seus fatores, incluindo 1 (mas não o próprio número), for igual ao número. Por exemplo, 6 é um número perfeito porque 6 = 1 + 2 + 3. Escreva um método `isPerfect` que determina se o parâmetro `number` é um número perfeito. Utilize esse método em um applet que exibe todos os números perfeitos entre 1 e 1.000. Exiba os fatores de cada número perfeito confirmando que o número é de fato perfeito. Desafie o poder de computação do seu computador testando números bem maiores que 1.000. Exiba os resultados.

6.25 **(*Números primos*)** Um número inteiro positivo é *primo* se for divisível apenas por 1 e por ele mesmo. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não o são. O número 1, por definição, não é primo.

a) Escreva um método que determina se um número é primo.

b) Utilize esse método em um aplicativo que determina e exibe todos os números primos menores que 10.000. Quantos números até 10.000 você precisa testar a fim de assegurar que encontrou todos os primos?

c) Inicialmente, você poderia pensar que $n/2$ é o limite superior que você deve testar para ver se um número é primo, mas você precisa ir apenas até a raiz quadrada de n. Reescreva o programa e execute-o de ambas as maneiras.

6.26 **(*Dígitos invertidos*)** Escreva um método que aceita um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7.631, o método deve retornar 1.367. Incorpore o método a um aplicativo que lê um valor a partir da entrada fornecida pelo usuário e exibe o resultado.

6.27 **(*Máximo divisor comum*)** O *máximo divisor comum* (*MCD*) de dois inteiros é o maior inteiro que é divisível por cada um dos dois números. Escreva um método `mcd` que retorna o máximo divisor comum de dois inteiros. [*Dica:* você pode querer utilizar o algoritmo de Euclides. Você pode localizar informações sobre o algoritmo em `en.wikipedia.org/wiki/Euclidean_algorithm`.] Incorpore o método a um aplicativo que lê dois valores do usuário e exibe o resultado.

6.28 Escreva um método `qualityPoints` que insere a média de um aluno e retorna `4` se a média do aluno estiver entre 90 e 100, `3` se a média estiver entre 80 e 89, `2` se a média estiver entre 70 e 79, `1` se a média estiver entre 60 e 69 e `0` se a média for mais baixa que 60. Incorpore o método a um aplicativo que lê do usuário um valor e exibe o resultado.

6.29 **(*Lançamento de uma moeda*)** Escreva um aplicativo que simula o lançamento de uma moeda. Deixe o programa lançar uma moeda toda vez que o usuário escolher a opção "`Toss Coin`" no menu. Conte o número de vezes que cada lado da moeda aparece. Exiba os resultados. O programa deve chamar um método `flip` separado que não aceita argumentos e retorna um valor a partir de um `Coin enum` (`HEADS` e `TAILS`). [*Nota:* se o programa simular, de maneira realista, o arremesso de moeda, cada lado da moeda deve aparecer aproximadamente metade das vezes.]

6.30 **(*Adivinhe o número*)** Escreva um aplicativo que execute "adivinhe o número" como mostrado a seguir: Seu programa escolhe o número a ser adivinhado selecionando um inteiro aleatório no intervalo de 1 a 1.000. O aplicativo exibe o prompt `Guess a number between 1 and 1000 [Adivinhe um número entre 1 e 1.000]`. O jogador insere uma primeira suposição. Se o palpite do jogador estiver incorreto, seu programa deve exibir `Too high. Try again [Muito alto. Tente novamente]` ou `Too low. Try again [Muito baixo. Tente novamente]` para ajudar o jogador a "zerar" mediante uma resposta correta. O programa deve solicitar ao usuário o próximo palpite. Quando o usuário insere a resposta correta, exiba `Congratulations. You guessed the number. [Parabéns, você advinhou o número.]` e permitir que o usuário escolha se quer jogar novamente. [*Nota:* a técnica de adivinhação empregada nesse problema é semelhante à de uma pesquisa binária, discutida no Capítulo 19.

6.31 **(*Modificação de adivinhe o número*)** Modifique o programa do Exercício 6.30 para contar o número de adivinhações que o jogador faz. Se o número for 10 ou menos, exiba `Either you know the secret or you got lucky! [Você sabe o segredo ou tem muita sorte!]` se o jogador adivinhar o número em 10 tentativas, exiba `Aha! You know the secret! [Aha! Você sabe o segredo!]` se o jogador fizer mais que 10 adivinhações, exiba `You should be able to do better! [Você deve ser capaz de fazer melhor.]` Por que esse jogo não deve precisar de mais que 10 suposições? Bem, com cada "boa adivinhação" o jogador deve ser capaz de eliminar a metade dos números, depois a metade dos números restantes, e assim por diante.

6.32 **(*Distância entre pontos*)** Escreva um método `distance` para calcular a distância entre dois pontos ($x1$, $y1$) e ($x2$, $y2$). Todos os números e valores de retorno devem ser do tipo `double`. Incorpore esse método a um aplicativo que permite que o usuário insira as coordenadas de pontos.

6.33 **(*Modificação do jogo Craps*)** Modifique o programa de jogo de dados da Figura 6.9 para permitir apostas. Inicialize a variável `bankBalance` como 1.000 dólares. Peça para o jogador inserir um `wager`. Verifique se `wager` é menor que ou igual a `bankBalance` e, se não for, faça o usuário reinserir `wager` até um `wager` válido ser inserido. Depois de um `wager` correto foi inserido, execute um jogo de dados. Se o jogador ganhar,

aumente `bankBalance` por `wager` e exiba o novo `bankBalance`. Se o jogador perder, diminua `bankBalance` por `wager`, exiba o novo `bankBalance`, verifique se `bankBalance` tornou-se zero e, se isso tiver ocorrido, exiba a mensagem `"Sorry. You busted!"` [`"Desculpe, mas você faliu!"`]. À medida que o jogo se desenvolver, exiba várias mensagens para criar uma "conversa", como `"Oh, you're going for broke, huh?"` [`"Xi, parece que você vai quebrar, hein?"`] ou `"Aw c'mon, take a chance!"` [`"Ei, vamos lá, dê uma chance para sua sorte"`] ou `"You're up big. Now's the time to cash in your chips!"` [`Você está montado na grana. Agora é hora de trocar essas fichas e embolsar o dinheiro!"`]. Implemente a "conversa" como um método separado que escolhe aleatoriamente a string a ser exibida.

6.34 (***Tabela de números binários octal e hexadecimal***) Escreva um aplicativo que exibe uma tabela de equivalentes binários octal e hexadecimal dos números decimais no intervalo 1 a 256. Se não estiver familiarizado com esses sistemas de numeração, leia primeiro o Apêndice H.

Fazendo a diferença

À medida que o preço dos computadores cai, torna-se viável para cada estudante, apesar da circunstância econômica, ter um computador e usá-lo na escola. Isso cria grandes oportunidades para aprimorar a experiência educativa de todos os estudantes em todo o mundo conforme sugerido pelos cinco exercícios a seguir. [*Nota:* verifique iniciativas como One Laptop Per Child Project (`www.laptop.org`). Pesquise também laptops "verdes" — quais são as principais características amigáveis ao meio ambiente desses dispositivos? Consulte a Electronic Product Environmental Assessment Tool (`www.epeat.net`) que pode ajudá-lo a avaliar o grau de responsabilidade ambiental ("greenness") de computadores desktop, notebooks e monitores para ajudá-lo a decidir que produtos comprar.]

6.35 (***Instrução auxiliada por computador***) O uso de computadores na educação é referido como *instrução auxiliada por computador* (*computer-assisted instruction — CAI*). Escreva um programa que ajudará um aluno da escola elementar a aprender multiplicação. Utilize um objeto `Random` para produzir dois inteiros positivos de um algarismo. O programa deve então fazer ao usuário uma pergunta, como:

```
How much is 6 times 7?
```

O aluno insere então a resposta. Em seguida, o programa verifica a resposta do aluno. Se estiver correta, exibe a mensagem `"Very good!"` e faz outra pergunta de multiplicação. Se a resposta estiver errada, exibe a mensagem `"No. Please try again."` e deixa que o aluno tente a mesma pergunta repetidamente até o aluno por fim responder corretamente. Um método separado deve ser utilizado para gerar cada nova pergunta. Esse método deve ser chamado uma vez quando a aplicação inicia a execução e toda vez o usuário responde a pergunta corretamente.

6.36 (***Instrução auxiliada por computador: reduzindo a fadiga do aluno***) Um problema em ambientes CAI é a fadiga do aluno. Isso pode ser reduzido variando-se as respostas do computador para prender a atenção do aluno. Modifique o programa do Exercício 6.35 para que vários comentários sejam exibidos para cada resposta como mostrado a seguir:

Respostas possíveis para uma resposta correta:

```
Very good! [Parabéns!]

Excellent! [Excelente!]

Nice work! [Bom trabalho!]

Keep up the good work! [Continue trabalhando bem assim!]
```

Respostas possíveis para uma resposta incorreta:

```
No. Please try again. [Não. Tente de novo.]

Wrong. Try once more. [Errado. Tente mais uma vez.]

Don't give up! [Não desista!]

No. Keep trying. [Não. Continue tentando.]
```

Utilize a geração de números aleatórios para escolher um número de 1 a 4 que será utilizado para selecionar uma de quatro respostas adequadas a cada resposta correta ou incorreta. Utilize uma instrução `switch` para emitir as respostas.

6.37 (***Instrução auxiliada por computador: monitorando o desempenho do aluno***) Sistemas mais sofisticados de instruções auxiliadas por computador monitoram o desempenho do aluno durante um período de tempo. A decisão sobre um novo tópico frequentemente é baseada no sucesso do aluno com tópicos prévios. Modifique o programa de Exercício 6.36 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois que o aluno digitar 10 respostas, seu programa deve calcular a porcentagem das que estão corretas. Se a porcentagem for menor que 75%, exiba `"Peça ajuda extra ao seu professor."` e, então, reinicialize o programa para que outro estudante possa tentá-lo. Se a porcentagem for 75% ou maior, exiba `"Parabéns, você está pronto para avançar para o próximo nível!"` e, então, reinicialize o programa para que outro estudante possa tentá-lo.

6.38 (***Instrução auxiliada por computador: níveis de dificuldade***) Os exercícios 6.35 a 6.37 desenvolveram um programa de instrução auxiliada por computador para ajudar a ensinar multiplicação a um aluno do ensino básico. Modifique o programa para permitir que o usuário insira um nível de dificuldade. Em um nível de dificuldade 1, o programa deve utilizar apenas números de um único dígito nos problemas; em um nível de dificuldade 2, os números com dois dígitos, e assim por diante.

6.39 (***Instrução auxiliada por computador: variando os tipos de problemas***) Modifique o programa do Exercício 6.38 para permitir ao usuário selecionar um tipo de problema de aritmética a ser estudado. Uma opção de `1` significa apenas problemas de adição, `2` significa apenas problemas de subtração, `3` significa apenas problemas de multiplicação, `4` significa apenas problemas de divisão e `5` significa uma combinação aleatória de todos esses tipos.

Vai pois agora, escreve isso numa
tábua perante eles, registra-o num livro.
— Isaías 30:8

Ir além do limite é tão incerto quanto não alcançar o objetivo.
— Confúcio

Comece pelo começo, … e vá até ao fim: então, pare.
— Lewis Carroll

7

Arrays e ArrayLists

Objetivos

Neste capítulo, você aprenderá:

- O que são arrays.
- A utilizar arrays para armazenar dados e recuperá-los de listas e tabelas de valores.
- A declarar arrays, inicializar arrays e referir-se a elementos específicos dos arrays.
- A utilizar a instrução `for` aprimorada para iterar por arrays.
- A passar arrays para métodos.
- A declarar e manipular arrays multidimensionais.
- A escrever métodos que utilizam listas de argumentos de comprimento variável.
- A ler argumentos da linha de comando em um programa.

Sumário

7.1 Introdução
7.2 Arrays
7.3 Declarando e criando arrays
7.4 Exemplos que utilizam arrays
7.5 Estudo de caso: simulação de embaralhamento e distribuição de cartas
7.6 A estrutura for aprimorada
7.7 Passando arrays para métodos
7.8 Estudo de caso: classe GradeBook utilizando um array para armazenar notas
7.9 Arrays multidimensionais
7.10 Estudo de caso: classe GradeBook utilizando um array bidimensional
7.11 Listas de argumentos de comprimento variável
7.12 Utilizando argumentos de linha de comando
7.13 Classe Arrays
7.14 Introdução a coleções e classe ArrayList
7.15 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando arcos
7.16 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Seção especial: criando seu próprio computador | Fazendo a diferença

7.1 Introdução

Este capítulo introduz **estruturas de dados** — coleções de itens de dados relacionados. **Arrays** são estruturas de dados consistindo em itens de dados do mesmo tipo relacionados. Arrays tornam conveniente processar grupos relacionados de valores. O tamanho dos arrays permanece o mesmo depois de serem criados, embora uma variável de array possa ser reatribuída de tal maneira que ela referencie um novo array de um tamanho diferente. Estudaremos as estruturas de dados a fundo nos capítulos 20–22.

Depois de discutir como os arrays são declarados, criados e inicializados, apresentamos exemplos práticos que demonstram manipulações de array comuns. Também apresentamos um estudo de caso que examina como os arrays podem ajudar a simular as ações de embaralhar e distribuir cartas em um aplicativo de jogo de cartas. Introduzimos a instrução `for` aprimorada do Java, que permite a um programa acessar os dados em um array mais facilmente que a instrução `for` controlada por contador apresentada na Seção 5.3. Aprimoramos o estudo de caso do `GradeBook` dos capítulos 3–5. Especificamente, utilizamos arrays para permitir que a classe mantenha um conjunto de notas escolares na memória e analise as notas dos alunos a partir de múltiplos exames. Mostramos como utilizar listas de argumento de tamanho variável para criar métodos que podem ser chamados com números variados de argumentos e demonstramos como processar argumentos de linha de comando no método `main`. Em seguida, apresentamos algumas manipulações de array comuns com métodos `static` da classe `Arrays` do pacote `java.util`.

Embora comumente utilizados, arrays têm capacidades limitadas. Por exemplo, você precisa especificar o tamanho de um array e, se em tempo de execução quiser modificá-lo, você deverá fazer isso manualmente criando um novo array. No final deste capítulo, introduziremos uma das estruturas de dados pré-construídas do Java a partir das classes de coleção de API do Java. Estas oferecem melhores capacidades do que arrays tradicionais. Elas são reutilizáveis, confiáveis, poderosas e eficientes e foram cuidadosamente projetadas e testadas para assegurar a qualidade e o desempenho. Focalizamos a coleção `ArrayList`. `ArrayLists` são semelhantes a array, mas fornecem funcionalidades adicionais, como **redimensionamento dinâmico** — o aumento automático do tamanho dos arrays em tempo de execução para acomodar elementos adicionais.

7.2 Arrays

Um array é um grupo de variáveis (chamados **elementos** ou **componentes**) que contém valores todos do mesmo tipo. Os arrays são objetos, portanto, considerados tipos por referência. Como você logo verá, o que em geral consideramos um array é, na verdade, uma referência a um objeto array na memória. Os elementos de um array podem ser tipos primitivos ou tipos por referência (inclusive arrays, como veremos na Seção 7.9). Para referenciar um elemento particular em um array, especificamos o nome da referência para o array e o número de posição do elemento no array. O número de posição do elemento é chamado de **índice** ou **subscrito** do elemento.

A Figura 7.1 mostra uma representação lógica de um array de inteiro chamado `c`. Esse array contém 12 *elementos*. Um programa refere-se a qualquer um desses elementos com uma **expressão de acesso ao array** que inclui o nome do array seguido pelo índice do elemento particular entre **colchetes (`[]`)**. O primeiro elemento em cada array tem **índice zero** e às vezes é chamado de **zero-ésimo elemento**. Assim, os elementos do array `c` são `c[0]`, `c[1]`, `c[2]` etc. O índice mais alto no array `c` é 11, que é 1 menor que 12 — o número de elementos no array. Nomes de array seguem as mesmas convenções que outros nomes de variável.

Um índice deve ser um inteiro não negativo. Um programa pode utilizar uma expressão como um índice. Por exemplo, se assumimos que a variável `a` é `5` e a variável `b`, `6`, então a instrução:

```
c[ a + b ] += 2;
```

adiciona 2 ao elemento do array `c[11]`. Observe que um nome de array indexado é uma expressão de acesso ao array. Essas expressões podem ser utilizadas no lado esquerdo de uma atribuição para colocar um novo valor em um elemento de array.

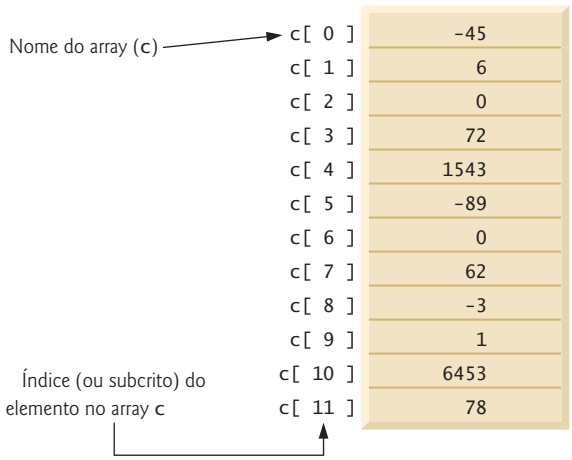


Figura 7.1 | Um array de 12 elementos.

Erro comum de programação 7.1

Um índice precisa ser um valor `int` ou um valor de um tipo que possa ser promovido a `int` — a saber, `byte`, `short` ou `char`, mas não `long`; caso contrário, ocorrerá um erro de compilação.

Vamos examinar o array `c` na Figura 7.1 mais atentamente. O **nome** do array é `c`. Cada objeto de array conhece seu próprio tamanho e armazena-o em uma **variável de instância `length`**. A expressão array `c.length` acessa campo `length` do `c` para determinar o comprimento do array. Observe que embora a variável de instância `length` de um array seja `public`, ela não pode ser alterada porque é uma variável `final`. Esse array com 12 elementos são conhecidos `c[0]`, `c[1]`, `c[2]`, ..., `c[11]`. O valor de `c[0]` é `-45`, o valor de `c[1]` é `6`, o valor de `c[2]` é `0`, o valor de `c[7]` é `62` e o valor de `c[11]` é `78`. Se calculássemos a soma dos valores contidos nos primeiros três elementos de array `c` e armazenássemos o resultado na variável `sum`, escreveríamos:

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

Para dividir o valor de `c[6]` por 2 e atribuir o resultado à variável `x`, escreveríamos:

```
x = c[ 6 ] / 2;
```

7.3 Declarando e criando arrays

Os objetos de array ocupam espaço na memória. Como os outros objetos, os arrays são criados com a palavra-chave `new`. Para criar um objeto de array, especifique o tipo dos elementos do array e o número de elementos como parte de uma **expressão de criação de array** que utiliza a palavra-chave `new`. Tal expressão retorna uma referência que pode ser armazenada em uma variável de array. A declaração e a expressão de criação de arrays a seguir criam um objeto de array que contém 12 elementos `int` e armazenam a referência do array na variável `c` do array:

```
int[] c = new int[ 12 ];
```

Essa expressão pode ser usada para criar o array da Figura 7.1. Essa tarefa também pode ser realizada em dois passos como a seguir:

```
int[] c; // declara a variável de array

c = new int[ 12 ]; // cria o array; atribui à variável de array
```

Na declaração, os colchetes que seguem o tipo indicam que `c` é uma variável que referenciará um array (isto é, a variável armazenará uma referência de array). Na instrução de atribuição, a variável de array `c` recebe a referência para um novo array de 12 elementos `int`. Quando um array é criado, cada elemento do array recebe um valor padrão — zero para os elementos numéricos de tipo primitivo, `false`

para elementos `boolean` e `null` para referências. Como veremos mais adiante, você pode fornecer valores de elemento iniciais não padrão ao criar um array.

Erro comum de programação 7.2

Em uma declaração de array, especificar o número de elementos entre os colchetes da declaração (por exemplo, `int[12] c;`) é um erro de sintaxe.

Um programa pode criar vários arrays em uma única declaração. A declaração seguinte reserva 100 elementos para `b` e 27 elementos para `x`:

```
String[] b = new String[ 100 ], x = new String[ 27 ];
```

Quando o tipo do array e colchetes são combinados no início da declaração, todos os identificadores na declaração são variáveis de array. Nesse caso, as variáveis `b` e `x` referem-se a arrays `String`. Para maior legibilidade, preferimos declarar apenas uma variável por declaração. A declaração anterior é equivalente a:

```
String[] b = new String[ 100 ]; // cria array b

String[] x = new String[ 27 ]; // cria array x
```

Boa prática de programação 7.1

Para legibilidade, declare apenas uma variável por declaração. Mantenha cada declaração em uma linha separada e inclua um comentário que descreva a variável sendo declarada.

Quando somente uma variável é declarada em cada declaração, os colchetes podem ser colocados depois do tipo ou depois do nome da variável de array, como em:

```
String b[] = new String[ 100 ]; // cria array b

String x[] = new String[ 27 ]; // cria array x
```

Um programa pode declarar arrays de qualquer tipo. Cada elemento de um array do tipo primitivo contém um valor do tipo de elemento declarado do array. De maneira semelhante, em um array de um tipo por referência, todo elemento é uma referência a um objeto do tipo de elemento declarado do array. Por exemplo, todo elemento de um array `int` é um valor `int` e todo elemento de um array `String` é uma referência a um objeto `String`.

Erro comum de programação 7.3

Declarar múltiplas variáveis de array em uma única declaração pode levar a erros sutis. Considere a declaração `int [] a, b, c;`. Se `a`, `b` e `c` devem ser declarados como variáveis de array, então essa declaração é correta — colocar os colchetes logo depois do tipo indica que todos os identificadores na declaração são variáveis de array. Entretanto, se apenas `a` destina-se a ser uma variável de array, e `b` e `c` variáveis `int` individuais, então essa declaração é incorreta — a declaração `int a[], b, c;` alcançaria o resultado desejado.

7.4 Exemplos que utilizam arrays

Esta seção apresenta vários exemplos que demonstram a declaração, criação e inicialização de arrays e a manipulação de elementos de array.

Criando e inicializando um array

O aplicativo da Figura 7.2 utiliza a palavra-chave `new` para criar um array de 10 elementos `int`, que são inicialmente zero (o padrão para variáveis `int`).

```
1  // Figura 7.2: InitArray.java
2  // Inicializando os elementos de um array como valores padrão de zero.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
```

```
8          int[] array; // declara o array identificado
9
10         array = new int[ 10 ]; // cria o objeto do array
11
12         System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
13
14         // gera saída do valor de cada elemento do array
15         for ( int counter = 0; counter < array.length; counter++ )
16            System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17      } // fim de main
18   } // fim da classe InitArray
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Figura 7.2 | Inicializando os elementos de um array com valores zero padrão.

A linha 8 declara `array` — uma referência capaz de referenciar um array de elementos `int`. A linha 10 cria o objeto de array e atribui sua referência à variável `array`. A linha 12 gera saída dos títulos de coluna. A primeira coluna contém o índice (0–9) de cada elemento de array e a segunda coluna contém o valor padrão (0) de cada elemento de array.

A instrução `for` nas linhas 15 e 16 gera saída do número de índice (representado por `counter`) e valor de cada elemento de array (representado por `array[counter ]`). Observe que a variável de controle de loop `counter` inicialmente é `0` — os valores de índice iniciam em 0, então utilizar a contagem baseada em zero permite ao loop acessar cada elemento do array. A condição de continuação do loop `for` utiliza a expressão `array.length` (linha 15) para determinar o comprimento do array. Nesse exemplo, o tamanho do array é 10, portanto o loop continua a executar desde que o valor da variável de controle `counter` seja menor que 10. O valor de índice mais alto de um array de 10 elementos é 9, assim utilizar operador menor que na condição de continuação do loop garante que o loop não tentará acessar um elemento depois do final do array (isto é, durante a iteração final do loop, `counter` é 9). Logo veremos o que o Java faz quando ele encontra esse índice fora do intervalo no tempo de execução.

Utilizar um inicializador de array

Você pode criar um array e inicializar seus elementos com um **inicializador de array** — uma lista separada por vírgula de expressões (chamada **lista inicializadora**) colocada entre chaves. Nesse caso, o comprimento do array é determinado pelo número de elementos na lista inicializadora. Por exemplo:

```
int[] n = { 10, 20, 30, 40, 50 };
```

cria um array de cinco elementos com valores de índice `0`–`4`. O elemento `n[0]` é inicializado em `10`, `n[1]` é inicializado em `20` etc. Quando o compilador encontrar uma declaração de array que inclua uma lista de inicializador, ele conta o número de inicializadores na lista para determinar o tamanho do array; depois, configura a operação `new` apropriada "nos bastidores".

O aplicativo na Figura 7.3 inicializa um array de inteiro com 10 valores (linha 9) e exibe o array em formato tabular. O código para exibir os elementos do array (linhas 14–15) é idêntico ao que aparece na Figura 7.2 (linhas 15–16).

```
1  // Figura 7.3: InitArray.java
2  // Inicializando os elementos de um array com um inicializador de array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // lista de inicializador especifica o valor de cada elemento
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
```

```
12
13          // gera saída do valor de cada elemento do array
14          for ( int counter = 0; counter < array.length; counter++ )
15             System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16       } // fim de main
17    } // fim da classe InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Figura 7.3 | Inicializando os elementos de um array com um inicializador de array.

Calculando os valores a armazenar em um array

O aplicativo na Figura 7.4 cria um array de 10 elementos e atribui a cada elemento um dos inteiros pares de 2 a 20 (2, 4, 6, ..., 20). Em seguida, o aplicativo exibe o array em formato tabular. A instrução `for` nas linhas 12–13 calcula o valor de um elemento do array multiplicando o valor atual da variável de controle `counter` por 2 e adicionando 2.

```
1  // Figura 7.4: InitArray.java
2  // Calculando valores a serem colocados em elementos de um array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        final int ARRAY_LENGTH = 10; // declara constante
9        int[] array = new int[ ARRAY_LENGTH ]; // cria o array
10
11       // calcula o valor de cada elemento do array
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
16
17       // gera a saída do valor de cada elemento do array
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20    } // fim de main
21 } // fim da classe InitArray
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Figura 7.4 | Calculando os valores a serem colocados nos elementos de um array.

A linha 8 utiliza o modificador `final` para declarar a variável constante `ARRAY_LENGTH`, com o valor `10`. Variáveis constantes devem ser inicializadas antes de serem utilizadas e não podem ser modificadas depois. Se você tentar modificar uma variável `final` depois que ela é inicializada na declaração (como na linha 8), o compilador emite uma mensagem de erro como:

```
cannot assign a value to final variable nomeDaVariável
```

Se uma tentativa de acessar o valor de uma variável `final` for feita antes de ela ser inicializada, o compilador emite uma mensagem de erro como:

```
variable nomeDaVariável might not have been initialized
```

Boa prática de programação 7.2

Variáveis constantes também são chamadas ***constantes nomeadas****. Frequentemente, elas tornam os programas mais legíveis que os programas que utilizam valores literais (por exemplo, `10`) — uma constante identificada como `ARRAY_LENGTH` indica claramente seu propósito, enquanto um valor literal poderia ter diferentes significados com base em seu contexto.*

Erro comum de programação 7.4

Atribuir um valor a uma variável constante depois de ela ter sido inicializada é um erro de compilação.

Erro comum de programação 7.5

Tentar utilizar uma constante antes de ela ser inicializada é um erro de compilação.

Somando os elementos de um array

Frequentemente, os elementos de um array representam uma série de valores a ser utilizados em um cálculo. Se, por exemplo, elas representarem notas de exames, um professor talvez queira somar os elementos do array e utilizar essa soma a fim de calcular a média da turma para o exame. Os exemplos de `GradeBook` nas figuras 7.14 e 7.18 utilizam essa técnica.

A Figura 7.5 soma os valores contidos em um array de 10 elementos inteiros. O programa declara, cria e inicializa o array na linha 8. A instrução `for` realiza os cálculos. [*Nota:* os valores fornecidos como inicializadores de array costumam ser lidos em um programa em vez de especificados em uma lista de inicializador. Por exemplo, um aplicativo poderia inserir os valores de um usuário ou de um arquivo em disco (como discutido no Capítulo 17, "Arquivos, fluxos e serialização de objetos"). Fazer um programa ler uma entrada de dados (em vez de "codificar manualmente" esses dados no programa) torna o programa mais reutilizável, porque ele pode trabalhar com diferentes conjuntos de dados.]

```
1  // Figura 7.5: SumArray.java
2  // Calculando a soma dos elementos de um array.
3
4  public class SumArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // adiciona o valor de cada elemento ao total
12       for ( int counter = 0; counter < array.length; counter++ )
13          total += array[ counter ];
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // fim de main
17 } // fim da classe SumArray
```

```
Total of array elements: 849
```

Figura 7.5 | Calculando a soma dos elementos de um array.

Utilizando gráficos de barras para exibir dados de array graficamente

Muitos programas apresentam dados graficamente aos usuários. Por exemplo, os valores numéricos são frequentemente exibidos como barras em um gráfico de barras. Nesse gráfico, as barras mais longas representam os valores numéricos proporcionalmente maiores. Uma maneira simples de exibir os dados numéricos graficamente é utilizar um gráfico de barras que mostra cada valor numérico como uma barra de asteriscos (*).

Os professores frequentemente gostam de examinar a distribuição de notas de um exame. Um professor poderia representar em gráficos o número de notas em cada uma de várias categorias para visualizar a distribuição de notas. Suponha que as notas em um exame foram 87, 68, 94, 100, 83, 78, 85, 91, 76 e 87. Observe que elas incluem uma nota 100, duas notas no intervalo 90, quatro notas no intervalo 80, duas notas no intervalo 70, uma nota no intervalo 60 e nenhuma nota abaixo de 60. Nosso próximo aplicativo (Figura 7.6) armazena esses dados de distribuição de notas em um array de 11 elementos, cada um correspondendo a uma categoria das notas. Por exemplo, `array[0]` indica o número de notas no intervalo 0–9, `array[7]` o número de notas no intervalo 70–79 e `array[10]` o número de notas 100. A classe `GradeBooks` mais adiante no capítulo (figuras 7.14 e 7.18) contém o código que calcula essas frequências de notas com base no conjunto de notas. Por enquanto, criamos manualmente o array com as frequências das notas dadas.

O aplicativo lê os números a partir do array e representa as informações graficamente como um gráfico de barras. Ele exibe cada intervalo de notas seguido por uma barra de asteriscos indicando o número de notas nesse intervalo. Para rotular cada barra, as linhas 16–20 geram saída de um intervalo de notas (por exemplo, `"70-79: "`) com base no valor atual de `counter`. Quando `counter` for `10`, a linha 17 gera saída de `100` com uma largura de campo de 5, seguida por um dois-pontos e um espaço, para alinhar o rótulo `"100: "` com os outros rótulos de barra. A instrução `for` aninhada (linhas 23–24) gera saída das barras. Observe a condição de continuação do loop na linha 23 (`stars < array[counter] counter ]`). Toda vez que o programa alcançar o `for` interno, o loop conta de `0` até `array[counter] counter ]`, utilizando assim um valor em `array` para determinar o número de asteriscos a serem exibidos. Nesse exemplo, nenhum aluno recebeu uma nota abaixo de 60, portanto `array[0]`–`array[5]` contém zeros, e nenhum asterisco é exibido ao lado dos seis primeiros intervalos de notas. Na linha 19, o especificador de formato `%02d` indica que um valor `int` deve ser formatado como um campo de dois dígitos. O **flag 0** no especificador de formato exibe um `0` à esquerda para valores com menos dígitos do que a largura do campo (`2`).

```
1  // Figura 7.6: BarChart.java
2  // programa de impressão de gráfico de barras.
3
4  public class BarChart
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
9
10       System.out.println( "Grade distribution:" );
11
12       // para cada elemento de array, gera saída de uma barra do gráfico
13       for ( int counter = 0; counter < array.length; counter++ )
14       {
15          // imprime rótulo de barra ( "00-09: ", ..., "90-99: ", "100: " )
16          if ( counter == 10 )
17             System.out.printf( "%5d: ", 100 );
18          else
19             System.out.printf( "%02d-%02d: ",
20                counter * 10, counter * 10 + 9  );
21
22          // imprime a barra de asteriscos
23          for ( int stars = 0; stars < array[ counter ]; stars++ )
24             System.out.print( "*" );
25
26          System.out.println(); // inicia uma nova linha de saída
27       } // fim do for externo
28    } // fim de main
29 } // fim da classe BarChart
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
```

```
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Figura 7.6 | Programa de impressão de gráfico de barras.

Utilizando os elementos de um array como contadores

Às vezes, os programas utilizam as variáveis de contador para resumir dados, como os resultados de uma pesquisa. Na Figura 6.8, utilizamos os contadores separados em nosso programa de lançamento de dados para monitorar o número de ocorrências de cada face de um dado de seis faces quando o programa lançou o dado 6.000 vezes. Uma versão de array desse aplicativo é mostrada na Figura 7.7.

```
1  // Figura 7.7: RollDie.java
2  // Programa de jogo de dados utilizando arrays em vez de switch
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // gerador de número aleatório
10       int[] frequency = new int[ 7 ]; // array de contadores de frequência
11
12       // lança o dados 6000 vezes; utiliza o valor do dado como índice de frequência
13       for ( int roll = 1; roll <= 6000; roll++ )
14          ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];
15
16       System.out.printf( "%s%10s\n", "Face", "Frequency" );
17
18       // gera saída do valor de cada elemento do array
19       for ( int face = 1; face < frequency.length; face++ )
20          System.out.printf( "%4d%10d\n", face, frequency[ face ] );
21    } // fim de main
22 } // fim da classe RollDie
```

```
Face Frequency
   1       988
   2       963
   3      1018
   4      1041
   5       978
   6      1012
```

Figura 7.7 | Programa de jogo de dados utilizando arrays em vez de `switch`.

A Figura 7.7 utiliza o array `frequency` (linha 10) para contar as ocorrências de cada face do dado. A instrução única na linha 14 desse programa substitui as linhas 23–46 da Figura 6.8. A linha 14 utiliza um valor aleatório para determinar qual elemento `frequency` incrementar a cada iteração do loop. O cálculo na linha 14 produz números aleatórios de 1 a 6, então o array `frequency` deve ser grande o bastante para armazenar seis contadores. Contudo, utilizamos um array de sete elementos no qual ignoramos `frequency[0]` — é mais lógico que o valor nominal 1 incremente `frequency[1]` do que `frequency[0]`. Portanto, o valor de cada face é utilizado como um índice do array `frequency`. Na linha 14, o cálculo dentro dos colchetes é avaliado primeiro para determinar qual elemento do array incrementar e, então, o operador ++ adiciona um a esse elemento. Também substituímos as linhas 50–52 da Figura 6.8 fazendo loop pelo array `frequency` para gerar saída dos resultados (linhas 19–20).

Utilizando arrays para analisar resultados de enquetes

A Figura 7.8 utiliza arrays para resumir os resultados dos dados coletados em uma enquete:

> *Foi pedido a quarenta alunos para avaliar a qualidade da comida na cantina estudantil em uma escala de 1–10 (onde 1 significa péssimo e 10, excelente). Coloque as 40 respostas em um array de inteiros e resuma os resultados da enquete.*

Esse é um aplicativo típico de processamento de array. Queremos resumir o número de respostas de cada tipo (isto é, 1–10). O array `responses` (linhas 9–11) é um array `int` de 40 elementos das respostas à enquete. Utilizamos um array de 11 elementos `frequency` (linha 12) para contar o número de ocorrências de cada resposta (1–10). Cada elemento é inicializado em zero por padrão. Como na Figura 7.7, ignoramos `frequency[0]`.

```
1  // Figura 7.8: StudentPoll.java
2  // Programa de análise de enquete.
3
4  public class StudentPoll
5  {
6     public static void main( String[] args )
7     {
8        // array de respostas da enquete
9        int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
10          10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
11          4, 8, 6, 8, 10 };
12       int[] frequency = new int[ 11 ]; // array de contadores de frequência
13
14       // para cada resposta, seleciona elemento da resposta e
15       // usa esse valor como índice de frequência para determinar o elemento a incrementar
16       for ( int answer = 0; answer < responses.length; answer++ )
17          ++frequency[ responses[ answer ] ];
18
19       System.out.printf( "%s%10s", "Rating", "Frequency" );
20
21       // gera saída do valor de cada elemento do array
22       for ( int rating = 1; rating < frequency.length; rating++ )
23          System.out.printf( "%d%10d", rating, frequency[ rating ] );
24    } // fim de main
25 } // fim da classe StudentPoll
```

```
Rating Frequency
     1         2
     2         2
     3         2
     4         2
     5         5
     6        11
     7         5
     8         7
     9         1
    10         3
```

Figura 7.8 | Programa de análise de enquete.

O loop `for` na linhas 16–17 pega as respostas do array `responses` uma por vez e incrementa um dos 10 contadores no array `frequency` (`frequency[1]` to `frequency[10]`). A instrução-chave do loop é a linha 17, que incrementa o contador `frequency` adequado, dependendo do valor de `responses[answer]`.

Vamos considerar várias iterações do loop `for`. Quando a variável de controle `answer` é 0, o valor de `responses[answer]` é o valor de `responses [0]` (isto é, 1), portanto o programa interpreta `++frequency[responses[answer]]` como:

```
++frequency[ 1 ]
```

que incrementa o valor no elemento do array 1. Para avaliar a expressão, inicie com o valor no conjunto mais interno de colchetes (`answer`). Assim que souber o valor de `answer` (que é o valor da variável de controle de loop na linha 16), insira-o na expressão e avalie o próximo conjunto externo de colchetes (isto é, `responses[answer]`, que é um valor selecionado do array `responses` nas linhas 9–11). Então utilize o valor resultante como o índice do array `frequency` para especificar qual contador incrementar.

Quando `answer` é 1, `responses[answer]` é o valor de `responses[1]` (2), portanto o programa interpreta `++frequency[responses[answer]]` como:

```
++frequency[ 2 ]
```

o que incrementa o elemento do array 2.

Quando `answer` é 2, `responses[answer]` é o valor de `responses[2]` (6), portanto o programa interpreta `++frequency[responses[answer]]` como:

```
++frequency[ 6 ]
```

o que incrementa o elemento do array 6 e assim por diante. Independentemente do número de respostas processadas na enquete, o programa só exige um array de 11 elementos (ignorando o elemento zero) para resumir o resultado, porque todos os valores de resposta estão entre 1 e 10 e os valores de índice de um array de 11 elementos são de 0 a 10.

Se os dados no array `responses` contivessem valores inválidos, como 13, o programa teria tentado adicionar 1 a `frequency[13]`, que está fora dos limites do array. O Java não permite isso. Quando um programa Java executa, a JVM verifica índices de array para assegurar que eles são maiores que ou igual a 0 e menor que o comprimento do array — isso é chamado **verificação de limites**. Se um programa utilizar um índice inválido, o Java gera uma suposta exceção para indicar a ocorrência de um erro no programa em tempo de execução. Pode-se utilizar uma instrução de controle para evitar que ocorra esse erro "fora dos limites" (*out-of-bounds*). Por exemplo, a condição em uma instrução de controle poderia determinar se um índice é válido antes de permitir que ele seja utilizado em uma expressão de acesso ao array.

Dica de prevenção de erro 7.1

Uma exceção indica a ocorrência de um erro em um programa. Você pode escrever um código para recuperar-se de uma exceção e continuar a execução do programa, em vez de terminar o programa anormalmente. Quando um programa tentar acessar um elemento fora dos limites do array, ocorrerá uma `ArrayIndexOutOfBoundsException`. O tratamento de exceções é discutido no Capítulo 11.

Dica de prevenção de erro 7.2

Ao escrever um código para fazer loop por um array, assegure que o índice de array é sempre maior que ou igual a 0 e menor que o comprimento do array. A condição de continuação do loop deve impedir o acesso a elementos fora desse intervalo.

7.5 Estudo de caso: simulação de embaralhamento e distribuição de cartas

Os exemplos no capítulo até aqui utilizaram arrays contendo elementos de tipos primitivos. A partir da discussão da Seção 7.2, lembre-se de que os elementos de um array podem ser tipos primitivos ou tipos por referência. Esta seção utiliza a geração de números aleatórios e um array de elementos de tipo por referência, isto é, objetos que representam cartas de baralho, para desenvolver uma classe que simula o embaralhamento e distribuição das cartas. Essa classe pode então ser utilizada na implementação de aplicativos para jogos de cartas específicos. Os exercícios no fim do capítulo utilizam as classes desenvolvidas aqui para construir um aplicativo de pôquer simples.

Inicialmente, desenvolvemos a classe `Card` (Figura 7.9), que representa uma carta de baralho que tem uma face (por exemplo, `"Ace"`, `"Deuce"`, `"Three"`, ..., `"Jack"`, `"Queen"`, `"King"`) e um naipe (por exemplo, `"Hearts"`, `"Diamonds"`, `"Clubs"`, `"Spades"`). Em seguida, desenvolvemos a classe `DeckOfCards` (Figura 7.10), que cria um baralho de 52 cartas em que cada elemento é um objeto `Card`. Então criamos um aplicativo de teste (Figura 7.11) que demonstra as capacidades de embaralhamento e distribuição de cartas da classe `DeckOfCards`.

Classe Card

A classe `Card` (Figura 7.9) contém duas variáveis de instância `String` — `face` e `suit` — que são utilizadas para armazenar referências ao nome da face (ou valor) e o nome do naipe de uma carta (`Card`) específica. O construtor da classe (linhas 10–14) recebe duas `String`s que ele utiliza para inicializar `face` e `suit`. O método `toString` (linhas 17–20) cria uma `String` que consiste na `face` de `card`, a `String` `"of"` e o `suit` da carta. O método `toString` de `Card` pode ser invocado explicitamente para obter uma representação de string de um objeto `Card` (por exemplo, `"Ace of Spades"`). O método `toString` de um objeto é chamado implicitamente quando o objeto é utilizado onde uma `String` é esperada (por exemplo, quando `printf` gera saída do objeto como uma `String` utilizando o especificador de formato `%s` ou quando o objeto é concatenado para uma `String` utilizando o operador +). Para que esse comportamento ocorra, `toString` deve ser declarada com o cabeçalho mostrado na Figura 7.9.

```
1  // Figura 7.9: Card.java
2  // Classe Card representa uma carta de baralho.
3
4  public class Card
```

```
{
   private String face; // face da carta ("Ace", "Deuce", …)
   private String suit; // naipe da carta ("Hearts", "Diamonds", …)

   // construtor de dois argumentos inicializa face e naipe da carta
   public Card( String cardFace, String cardSuit )
   {
      face = cardFace; // inicializa face da carta
      suit = cardSuit; // inicializa naipe da carta
   } // fim do construtor Card de dois argumentos

   // retorna representação String de Card
   public String toString()
   {
      return face + " of " + suit;
   } // fim do método toString
} // fim da classe Card
```

Figura 7.9 | A classe `Card` representa uma carta de baralho.

Classe DeckOfCards

A classe `DeckOfCards` (Figura 7.10) declara como variável de instância um array `Card` chamado `deck` (linha 7). Observe que um array de um tipo por referência é declarado como qualquer outro array. A classe `DeckOfCards` também declara uma variável de instância do tipo inteiro `currentCard` (linha 8) para representar o próximo `Card` a ser a distribuído partir do array `deck` e uma constante identificada `NUMBER_OF_CARDS` (linha 9) para indicar o número de `Cards` no baralho (52).

```
// Figura 7.10: DeckOfCards.java
// classe DeckOfCards representa um baralho.
import java.util.Random;

public class DeckOfCards
{
   private Card[] deck; // array de objetos Card
   private int currentCard; // índice do próximo Card a ser distribuído
   private static final int NUMBER_OF_CARDS = 52; // número constante de Cards
   // gerador de número aleatório
   private static final Random randomNumbers = new Random();

   // construtor preenche baralho de cartas
   public DeckOfCards()
   {
      String[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
         "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
      String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };

      deck = new Card[ NUMBER_OF_CARDS ]; // cria array de objetos Card
      currentCard = 0; // configura currentCard então o primeiro Card distribuído é deck[ 0 ]

      // preenche baralho com objetos Card
      for ( int count = 0; count < deck.length; count++ )
         deck[ count ] =
            new Card( faces[ count % 13 ], suits[ count / 13 ] );
   } // fim do construtor DeckOfCards

   // embaralha as cartas com um algoritmo de uma passagem
   public void shuffle()
   {
      // depois de embaralhar, a distribuição deve iniciar em deck[ 0 ] novamente
      currentCard = 0; // reinicializa currentCard

      // para cada Card, seleciona outro Card aleatório e os compara
      for ( int first = 0; first < deck.length; first++ )
      {
         // seleciona um número aleatório entre 0 e 51
         int second =  randomNumbers.nextInt( NUMBER_OF_CARDS );
```

```
40
41             // compara Card atual com Card aleatoriamente selecionado
42             Card temp = deck[ first ];
43             deck[ first ] = deck[ second ];
44             deck[ second ] = temp;
45          } // fim do for
46       } // fim do método shuffle
47
48       // distribui um Card
49       public Card dealCard()
50       {
51          // determina se ainda há Cards a serem distribuídos
52          if (currentCard < deck.length )
53             return deck[ currentCard++ ]; // retorna Card atual no array
54          else
55             return null; // retorna nulo para indicar que todos os Cards foram distribuídos
56       } // fim do método dealCard
57    } // fim da classe DeckOfCards
```

Figura 7.10 | A classe `DeckOfCards` representa um baralho de cartas.

O construtor da classe instancia o array `deck` (linha 20) com `NUMBER_OF_CARDS` (52) elementos. Quando criados pela primeira vez, os elementos do array `deck` são `null` por padrão, então o construtor utiliza uma instrução `for` (linhas 24–26) para preencher o array `deck` com `Cards`. A instrução `for` inicializa a variável de controle `count` como `0` e realiza um loop enquanto `count` for menor que `deck.length`, fazendo com que `count` assuma cada valor inteiro de 0 a 51 (os índices do array `deck`). Cada `Card` é instanciada e inicializada com duas `Strings` — uma a partir do array `faces` (que contém as `Strings` `"Ace"` a `"King"`) e uma a partir do array `suits` (que contém as `Strings` `"Hearts"`, `"Diamonds"`, `"Clubs"` e `"Spades"`). O cálculo `count % 13` sempre resulta em um valor de 0–12 (os 13 índices do array `faces` nas linhas 16–17) e o cálculo `count / 13` sempre resulta em um valor de 0 a 3 (os quatro índices do array `suits` na linha 18). Quando o array `deck` é inicializado, ele contém as `Cards` com as faces `"Ace"` a `"King"` na ordem para cada naipe (`"Hearts"`, depois `"Diamonds"`, depois `"Clubs"` e então `"Spades"`). Observe que utilizamos arrays `String` para representar as faces e naipes nesse exemplo. No Exercício 7.34, solicitamos que você modifique esse exemplo para utilizar arrays de constantes de lista a fim de representar as faces e naipes.

O método `shuffle` (linhas 30–46) embaralha as `Cards` no baralho. O método faz um loop por todos os 52 `Cards` (índices de array 0 a 51). Para cada `Card`, um número entre 0 e 51 é escolhido aleatoriamente para selecionar outro `Card`. Em seguida, o objeto `Card` atual e o objeto `Card` aleatoriamente selecionado são trocados no array. Essa troca é realizada pelas três atribuições nas linhas 42–44. A variável extra `temp` armazena temporariamente um dos dois objetos `Card` sendo comparados. A comparação não pode ser realizada apenas com as duas instruções:

```
deck[ first ] = deck[ second ];
deck[ second ] = deck[ first ];
```

Se `deck[first]` for o `"Ace"` de `"Spades"` e `deck[second]` for `"Queen"` de `"Hearts"`, depois da primeira atribuição, ambos os elementos do array conterão a `"Queen"` de `"Hearts"` e o `"Ace"` de `"Spades"` é perdido — daí a necessidade de uma variável extra `temp`. Depois de o loop `for` terminar, os objetos `Card` são ordenados aleatoriamente. Um total de apenas 52 trocas é feito em uma única passagem pelo array inteiro e o array dos objetos `Card` é embaralhado!

O método `dealCard` (linhas 49–56) distribui uma `Card` no array. Lembre-se de que `currentCard` indica o índice do próximo `Card` a ser distribuído (isto é, o `Card` na parte superior do baralho). Portanto, a linha 52 compara `currentCard` com o comprimento do array `deck`. Se o `deck` não estiver vazio (isto é, se `currentCard` for menor que 52), a linha 53 retorna a primeira `Card` e pós-incrementa `currentCard` para preparar para a próxima chamada a `dealCard` — do contrário, `null` é retornado. A partir da discussão do Capítulo 3, lembre-se de que `null` representa uma "referência a nada".

Embaralhando e distribuindo cartas

A Figura 7.11 demonstra a classe `DeckOfCards` (Figura 7.10). A linha 9 cria um objeto `DeckOfCards` chamado `myDeckOfCards`. Lembre-se de que o construtor `DeckOfCards` cria o baralho com 52 objetos `Card` na ordem por naipe e face. A linha 10 invoca método `shuffle` de `myDeckOfCards` para reorganizar os objetos `Card`. As linhas 13–20 distribuem as 52 `Cards` e as imprime em quatro colunas com 13 `Cards` cada. A linha 16 distribui um objeto `Card` invocando o método `dealCard` de `myDeckOfCards` e, então, exibe a `Card` alinhada à esquerda em um campo de 19 caracteres. Quando uma `Card` é impressa como uma `String`, o método `toString` de `Card` (linhas 17–20 da Figura 7.9) é invocado implicitamente. As linhas 18–19 iniciam uma nova linha depois de cada quatro `Cards`.

```
1  // Figura 7.11: DeckOfCardsTest.java
2  // Embaralhando e distribuindo cartas.
3
4  public class DeckOfCardsTest
5  {
6     // executa o aplicativo
7     public static void main( String[] args )
8     {
9        DeckOfCards myDeckOfCards = new DeckOfCards();
10       myDeckOfCards.shuffle(); // coloca Cards em ordem aleatória
11
12       // imprime todas as 52 cartas na ordem em que são distribuídas
13       for ( int i = 1; i <= 52; i++ )
14       {
15          // distribui e exibe uma Card
16          System.out.printf( "%-19s", myDeckOfCards.dealCard() );
17
18          if ( i % 4 == 0 ) // imprime uma nova linha a cada 4 cartas
19             System.out.println();
20       } // fim do for
21    } // fim do main
22 } // fim da classe DeckOfCardsTest
```

```
Six of Spades      Eight of Spades    Six of Clubs       Nine of Hearts
Queen of Hearts    Seven of Clubs     Nine of Spades     King of Hearts
Three of Diamonds  Deuce of Clubs     Ace of Hearts      Ten of Spades
Four of Spades     Ace of Clubs       Seven of Diamonds  Four of Hearts
Three of Clubs     Deuce of Hearts    Five of Spades     Jack of Diamonds
King of Clubs      Ten of Hearts      Three of Hearts    Six of Diamonds
Queen of Clubs     Eight of Diamonds  Deuce of Diamonds  Ten of Diamonds
Three of Spades    King of Diamonds   Nine of Clubs      Six of Hearts
Ace of Spades      Four of Diamonds   Seven of Hearts    Eight of Clubs
Deuce of Spades    Eight of Hearts    Five of Hearts     Queen of Spades
Jack of Hearts     Seven of Spades    Four of Clubs      Nine of Diamonds
Ace of Diamonds    Queen of Diamonds  Five of Clubs      King of Spades
Five of Diamonds   Ten of Clubs       Jack of Spades     Jack of Clubs
```

Figura 7.11 | Embaralhando e distribuindo cartas.

7.6 A estrutura for aprimorada

A **instrução for aprimorada** itera pelos elementos de um array sem usar um contador, evitando assim a possibilidade de ultrapassar o limite do array. Mostramos como utilizar a instrução `for` aprimorada com as estruturas de dados predefinidas da API do Java (chamadas coleções) na Seção 7.14. A sintaxe de uma instrução `for` aprimorada é:

```
for ( parâmetro : nomeDoArray )
   instrução
```

onde *parâmetro* tem duas partes — um tipo e um identificador (por exemplo, `int number`) e *nomeDoArray* é o array pelo qual iterar. O tipo do parâmetro deve ser consistente com o tipo de elementos no array. Como ilustrado no próximo exemplo, o identificador representa valores sucessivos do array nas sucessivas iterações do loop.

A Figura 7.12 utiliza a instrução `for` aprimorada (linhas 12–13) para somar os números inteiros em um array de notas de alunos. O parâmetro da instrução `for` aprimorada é do tipo `int`, porque `array` contém valores `int` — o loop seleciona um valor `int` a partir do array a cada iteração. A instrução `for` aprimorada itera por valores sucessivos do array um a um. O cabeçalho da instrução pode ser lido como "para cada iteração, atribui o próximo elemento de `array` à variável `int number`, depois executa a instrução seguinte". Portanto, para cada iteração, o identificador `number` representa um valor `int` no `array`. As linhas 12–13 são equivalentes à seguinte repetição controlada por contador utilizada nas linhas 12–13 da Figura 7.5 para somar os inteiros no `array`:

```
for ( int counter = 0; counter < array.length; counter++ )
   total += array[ counter ];
```

```
1  // Figura 7.12: EnhancedForTest.java
2  // Usando instrução for aprimorada para somar inteiros em um array.
3
4  public class EnhancedForTest
5  {
6     public static void main( String[] args )
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // adiciona o valor de cada elemento ao total
12       for ( int number : array )
13          total += number;
14
15       System.out.printf( "Total of array elements: %d\n", total );
16    } // fim do main
17 } // fim da classe EnhancedForTest
```

```
Total of array elements: 849
```

Figura 7.12 | Utilizando a instrução `for` aprimorada para somar inteiros em um array.

A instrução `for` aprimorada simplifica o código para iterar por um array. Observe, porém, que a instrução `for` aprimorada pode ser utilizada apenas para obter elementos do array — ela não pode ser utilizada para modificar elementos. Se seu programa precisar modificar elementos, utilize a tradicional instrução `for` controlada por contador.

A instrução `for` aprimorada pode ser utilizada no lugar da instrução `for` controlada por contador sempre que o loop de código por um array não exigir acesso ao contador que indica o índice do elemento do array atual. Por exemplo, somar os inteiros em um array exige acesso apenas aos valores de elemento — o índice de cada elemento é irrelevante. Entretanto, se um programa precisar utilizar um contador por alguma razão diferente de simplesmente fazer loop por um array (por exemplo, imprimir um número de índice ao lado do valor de cada elemento do array, como nos primeiros exemplos deste capítulo), utilize a instrução `for` controlada por contador.

7.7 Passando arrays para métodos

Esta seção demonstra como passar arrays e elementos individuais de array como argumentos para métodos. Para passar um argumento de array para um método, especifique o nome do array sem nenhum colchete. Por exemplo, se o array `hourlyTemperatures` for declarado como:

```
double[] hourlyTemperatures = new double[ 24 ];
```

então a chamada de método:

```
modifyArray( hourlyTemperatures );
```

passa a referência do array `hourlyTemperatures` para o método `modifyArray`. Todo objeto do array "conhece" seu próprio comprimento (via seu campo `length`). Portanto, quando passamos a referência de um objeto de array em um método, não precisamos passar o comprimento de array como um argumento adicional.

Para um método receber uma referência de array por uma chamada de método, a lista de parâmetros do método deve especificar um parâmetro de array. Por exemplo, o cabeçalho de um método `modifyArray` poderia ser escrito assim:

```
void modifyArray( double[] b )
```

indicando que `modifyArray` recebe a referência de um array `double` no parâmetro `b`. A chamada de método passa a referência do array `hourlyTemperature`, então quando o método chamado utiliza a variável de array `b`, ele referencia o mesmo objeto de array que `hourlyTemperatures` no chamador.

Quando um argumento para um método for um array inteiro ou um elemento de array individual de um tipo por referência, o método chamado recebe uma cópia da referência. Entretanto, quando um argumento para um método for um elemento de array individual de um tipo primitivo, o método chamado recebe uma cópia do valor do elemento. Esses valores primitivos são chamados **escalares** ou **quantidades escalares**. Para passar um elemento individual de array para um método, utilize o nome indexado do array como um argumento na chamada de método.

A Figura 7.13 demonstra a diferença entre passar um array inteiro e passar um elemento do array do tipo primitivo para um método. Note que `main` invoca métodos `static` `modifyArray` (linha 19) e `modifyElement` (linha 30) diretamente. Lembre-se da Seção 6.5 de que um método `static` de uma classe pode invocar diretamente outros métodos `static` da mesma classe.

```
1  // Figura 7.13: PassArray.java
2  // Passando arrays e elementos de arrays individuais aos métodos.
3
4  public class PassArray
5  {
6     // main cria array e chama modifyArray e modifyElement
7     public static void main( String[] args )
8     {
9        int[] array = { 1, 2, 3, 4, 5 };
10
11       System.out.println(
12          "Effects of passing reference to entire array:\n" +
13          "The values of the original array are:" );
14
15       // gera saída de elementos do array original
16       for ( int value : array )
17          System.out.printf( "   %d", value );
18
19       modifyArray( array ); // passa a referência do array
20       System.out.println( "\n\nThe values of the modified array are:" );
21
22       // gera saída de elementos do array modificado
23       for ( int value : array )
24          System.out.printf( "   %d", value );
25
26       System.out.printf(
27          "\n\nEffects of passing array element value:\n" +
28          "array[3] before modifyElement: %d\n", array[ 3 ] );
29
30       modifyElement( array[ 3 ] ); // tenta modificar o array[ 3 ]
31       System.out.printf(
32          "array[3] after modifyElement: %d\n", array[ 3 ] );
33    } // fim de main
34
35    // multiplica cada elemento de um array por 2
36    public static void modifyArray( int[] array2 )
37    {
38       for ( int counter = 0; counter < array2.length; counter++ )
39          array2[ counter ] *= 2;
40    } // fim do método modifyArray
41
42    // multiplica argumento por 2
43    public static void modifyElement( int element )
44    {
45       element *= 2;
46       System.out.printf(
47          "Value of element in modifyElement: %d\n", element );
48    } // fim do método modifyElement
49 } // fim da classe PassArray
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Figura 7.13 | Passando arrays e elementos de arrays individuais para métodos.

A instrução `for` aprimorada nas linhas 16–17 gera saída dos cinco elementos `int` do `array`. A linha 19 invoca o método `modifyArray`, passando `array` como um argumento. O método `modifyArray` (linhas 36–40) recebe uma cópia da referência de `array` e utiliza essa referência para multiplicar cada um dos elementos do `array` por 2. Para provar que os elementos do `array` foram modificados, as linhas 23–24 geram saída dos cinco elementos do `array` novamente. Quando a saída é exibida, o método `modifyArray` duplica o valor de

cada elemento. Observe que não podemos utilizar a instrução `for` aprimorada nas linhas 38–39 porque estamos modificando os elementos do array.

A Figura 7.13 a seguir demonstra que quando uma cópia de um elemento de array individual do tipo primitivo é passada para um método, modificar a cópia no método chamado não afeta o valor original desse elemento no array do método chamador. As linhas 26–28 geram saída do valor de `array[3]` antes de invocar o método `modifyElement`. Lembre-se de que o valor desse elemento agora é 8 depois que ele foi modificado na chamada a `modifyArray`. A linha 30 chama o método `modifyElement` e passa `array[3]` como um argumento. Lembre-se de que `array[3]` é, de fato, um valor `int` (8) em `array`. Portanto, o programa passa uma cópia do valor de `array[3]`. O método `modifyElement` (linhas 43–48) multiplica o valor recebido como um argumento por 2, armazena o resultado em seu parâmetro `element` e gera saída do valor de `element` (16). Visto que os parâmetros de método, como as variáveis locais, deixam de existir quando o método em que eles são declarados conclui a execução, o parâmetro de método `element` é destruído quando o método `modifyElement` termina. Quando o programa retorna o controle para `main`, as linhas 31–32 geram saída do valor não modificado de `array[3]` (isto é, 8).

Notas sobre a passagem de argumentos para métodos

O exemplo anterior demonstrou como arrays e elementos do array de tipo primitivo são passados como argumentos para métodos. Agora examinamos mais atentamente como os argumentos em geral são passados para os métodos. Duas maneiras de passar argumentos em chamadas de método em muitas linguagens de programação são **passagem por valor** e **passagem por referência** (também denominadas **chamada por valor** e **chamada por referência**). Quando um argumento é passado por valor, uma cópia do valor do argumento é passada para o método chamado. O método chamado funciona exclusivamente com a cópia. As alterações na cópia do método chamado não afetam o valor da variável original no chamador.

Quando um argumento é passado por referência, o método chamado pode acessar o valor do argumento no chamador diretamente e modificar esses dados, se necessário. Passar por referência aprimora o desempenho eliminando a necessidade de copiar quantidades de dados possivelmente grandes.

Ao contrário de algumas outras linguagens, o Java não permite aos programadores escolher passar por valor ou por referência — todos os argumentos são passados por valor. Uma chamada de método pode passar dois tipos de valores para um método — cópias de valores primitivos (por exemplo, valores de tipo `int` e `double`) e cópias de referências para objetos. Os objetos em si não podem ser passados para os métodos. Quando um método modifica um parâmetro do tipo primitivo, as alterações no parâmetro não têm nenhum efeito no valor original do argumento no método chamador. Por exemplo, quando a linha 30 em `main` da Figura 7.13 passa `array[3]` para o método `modifyElement`, a instrução na linha 45 que dobra o valor do parâmetro `element` não tem nenhum efeito sobre o valor de `array[3]` em `main`. Isso também é verdadeiro para os parâmetros de tipo por referência. Se você modificar um parâmetro de tipo por referência para que ele referencie outro objeto, somente o parâmetro referencia o novo objeto — a referência armazenada na variável do chamador ainda referencia o objeto original.

Embora uma referência do objeto seja passada por valor, um método ainda pode interagir com o objeto referenciado chamando seus métodos `public` que utilizam a cópia da referência do objeto. Visto que a referência armazenada no parâmetro é uma cópia da referência que foi passada como um argumento, o parâmetro no método chamado e o argumento no método chamador referenciam o mesmo objeto na memória. Por exemplo, na Figura 7.13, tanto o parâmetro `array2` no método `modify-Array` como a variável `array` em `main` referenciam o mesmo objeto de array na memória. Quaisquer modificações feitas utilizando o parâmetro `array2` são executadas no objeto que `array` referencia no método chamador. Na Figura 7.13, as alterações feitas em `modifyArray` utilizando `array2` afetam o conteúdo do objeto de array referenciado por `array` em `main`. Portanto, com uma referência para um objeto, o método chamado pode manipular o objeto do chamador diretamente.

Dica de desempenho 7.1

Passar arrays por referência faz sentido por razões de desempenho. Se arrays fossem passados por valor, uma cópia de cada elemento seria passada. Para arrays grandes e frequentemente passados, isso desperdiçaria tempo e consumiria considerável capacidade de armazenamento para as cópias dos arrays.

7.8 Estudo de caso: classe GradeBook utilizando um array para armazenar notas

Versões anteriores de classe `GradeBook` processavam um conjunto de notas inserido pelo usuário, mas não mantinham os valores das notas individuais em variáveis de instância da classe. Portanto, os cálculos de repetição exigem que o usuário insira as mesmas notas novamente. Uma maneira de resolver esse problema seria armazenar cada nota inserida em uma instância individual da classe. Por exemplo, criaríamos as variáveis de instância `grade1`, `grade2`, ..., `grade10` na classe `GradeBook` para armazenar 10 notas de alunos. Mas isso tornaria complicado para o código somar as notas e determinar a média de classe, e a classe não seria capaz de processar nada mais que 10 notas por vez. Resolvemos esse problema armazenando as notas escolares em um array.

Armazenando notas de aluno em um array na classe GradeBook

A classe `GradeBook` (Figura 7.14) utiliza um array de `int`s para armazenar as notas de vários alunos em um único exame. Isso elimina a necessidade de inserir o mesmo conjunto de notas repetidamente. O array `grades` é declarado como uma variável de instância (linha 7),

portanto, cada objeto GradeBook mantém seu próprio conjunto de notas. O construtor (linhas 10–14) tem dois parâmetros — o nome do curso e um array de notas. Quando um aplicativo (por exemplo, a classe GradeBookTest na Figura 7.15) cria um objeto GradeBook, o aplicativo passa um array int existente para o construtor, que atribui a referência do array à variável de instância grades (linha 13). O tamanho do array grades é determinado pelo tamanho do array que é passado para o construtor. Portanto, um objeto GradeBook pode processar um número variável de notas. Os valores de nota no array passado poderiam ter sido inseridos a partir do usuário ou lidos de um arquivo em disco (como discutido no Capítulo 17). Em nosso aplicativo de teste, inicializamos um array com um conjunto de valores de nota (Figura 7.15, linha 10). Uma vez que as notas são armazenadas na variável de instância grades da classe GradeBook, todos os métodos da classe podem acessar os elementos de grades sempre que necessário para realizar vários cálculos.

```
1  // Figura 7.14: GradeBook.java
2  // classe GradeBook usando um array para armazenar notas de teste.
3
4  public class GradeBook
5  {
6     private String courseName; // nome do curso que essa GradeBook representa
7     private int[] grades; // array de notas de aluno
8
9     // construtor de dois argumentos inicializa courseName e o array de notas
10    public GradeBook( String name, int[] gradesArray )
11    {
12       courseName = name; // inicializa courseName
13       grades = gradesArray; // níveis de armazenamento
14    } // fim do construtor GradeBook de dois argumentos
15
16    // método para configurar o nome do curso
17    public void setCourseName( String name )
18    {
19       courseName = name; // armazena o nome do curso
20    } // fim do método setCourseName
21
22    // método para recuperar o nome do curso
23    public String getCourseName()
24    {
25       return courseName;
26    } // fim do método getCourseName
27
28   // exibe uma mensagem de boas-vindas para o usuário GradeBook
29    public void displayMessage()
30    {
31       // getCourseName obtém o nome do curso
32       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33          getCourseName() );
34    } // fim do método displayMessage
35
36    // realiza várias operações nos dados
37    public void processGrades()
38    {
39       // gera saída de array de notas
40       outputGrades();
41
42       // chama método getAverage para calcular a nota média
43       System.out.printf( "\nClass average is %.2f\n", getAverage() );
44
45       // chama métodos getMinimum e getMaximum
46       System.out.printf( "Lowest grade is %d\nHighest grade is %d\n\n",
47          getMinimum(),getMaximum() );
48
49       // chama outputBarChart para imprimir gráfico de distribuição de notas
50       outputBarChart();
51    } // fim do método processGrades
52
53    // localiza nota mínima
54    public int getMinimum()
55    {
56       int lowGrade = grades[ 0 ]; // assume que grades[ 0 ] é a menor nota
57
```

```
58          // faz um loop pelo array de notas
59          for ( int grade : grades )
60          {
61             // se nota for mais baixa que lowGrade, essa nota é atribuída a lowGrade
62             if ( grade < lowGrade )
63                lowGrade = grade; // nova nota mais baixa
64          } // fim do for
65
66          return lowGrade; // retorna nota mais baixa
67       } // fim do método getMinimum
68
69       // localiza nota máxima
70       public int getMaximum()
71       {
72          int highGrade = grades[ 0 ]; // assume que grades[ 0 ] é a maior nota
73
74          // faz um loop pelo array de notas
75          for ( int grade : grades )
76          {
77             // se a nota for maior que highGrade, atribui essa nota a highGrade
78             if ( grade > highGrade )
79                highGrade = grade; // nova nota mais alta
80          } // fim de for
81
82          return highGrade; // retorna nota mais alta
83       } // fim do método getMaximum
84
85       // determina média para o teste
86       public double getAverage()
87       {
88          int total = 0; // inicializa o total
89
90          // soma notas de um aluno
91          for ( int grade : grades )
92             total += grade;
93
94          // retorna média de notas
95          return (double) total / grades.length;
96       } // fim do método getAverage
97
98       // gera a saída do gráfico de barras exibindo distribuição de notas
99       public void outputBarChart()
100      {
101         System.out.println( "Grade distribution:" );
102
103         // armazena frequência de notas em cada intervalo de 10 notas
104         int[] frequency = new int[ 11 ];
105
106         // para cada nota, incrementa a frequência apropriada
107         for ( int grade : grades )
108            ++frequency[ grade / 10 ];
109
110         // para cada frequência de nota, imprime barra no gráfico
111         for ( int count = 0; count < frequency.length; count++ )
112         {
113            // imprime rótulo de barra ( "00-09: ", ..., "90-99: ", "100: " )
114            if ( count == 10 )
115               System.out.printf( "%5d: ", 100 );
116            else
117               System.out.printf( "%02d-%02d: ",
118                  count * 10, count * 10 + 9  );
119
120            // imprime a barra de asteriscos
121            for ( int stars = 0; stars < frequency[ count ]; stars++ )
122               System.out.print( "*" );
123
124            System.out.println(); // inicia uma nova linha de saída
125         } // fim do for externo
126      } // fim do método outputBarChart
```

```
127
128     // gera a saída do conteúdo do array de notas
129     public void outputGrades()
130     {
131        System.out.println( "The grades are:\n" );
132
133        // gera a saída da nota de cada aluno
134        for ( int student = 0; student < grades.length; student++ )
135           System.out.printf( "Student %2d: %3d\n",
136              student + 1, grades[ student ] );
137     } // fim do método outputGrades
138  } // fim da classe GradeBook
```

Figura 7.14 | Classe `GradeBook` utilizando um array para armazenar notas de teste.

O método `processGrades` (linhas 37–51) contém uma série de chamadas de método que imprimem um relatório que resume as notas. A linha 40 chama o método `outputGrades` para imprimir o conteúdo do array `grades`. As linhas 134–136 no método `outputGrades` utilizam uma instrução `for` para imprimir as notas dos alunos. Um `for` controlado por contador deve ser utilizado nesse caso, porque as linhas 135–136 utilizam o valor da variável contadora `student` para gerar saída de cada nota ao lado de um número de aluno particular (ver saída na Figura 7.15). Embora os índices de array iniciem em 0, em geral, um professor numeraria os alunos iniciando em 1. Portanto, as linhas 135–136 geram saída de `student + 1` como o número de aluno para produzir rótulos de nota `"Student 1: "`, `"Student 2: "` e assim por diante.

O método `processGrades` seguinte chama o método `getAverage` (linha 43) para obter a média das notas no array. O método `getAverage` (linhas 86–96) utiliza uma instrução `for` aprimorada para somar os valores no array `grades` antes de calcular a média. O parâmetro no cabeçalho `for` aprimorado (por exemplo, `int grade`) indica que para cada iteração, a variável `int grade` assume um valor no array `grades`. Observe que o cálculo da média na linha 95 utiliza `grades.length` para determinar o número de notas cuja média está sendo calculada.

As linhas 46–47 no método `processGrades` chamam os métodos `getMinimum` e `getMaximum` para determinar as notas mais baixas e mais altas de qualquer aluno no exame, respectivamente. Cada um desses métodos utiliza uma instrução `for` aprimorada para fazer loop pelo array `grades`. As linhas 59–64 no método `getMinimum` fazem um loop pelo array. As linhas 62–63 comparam cada nota a `lowGrade`; se uma nota for menor do que `lowGrade`, `lowGrade` é configurado com essa nota. Quando a linha 66 executar, `lowGrade` contém a nota mais baixa no array. O método `getMaximum` (linhas 70–83) funciona de maneira semelhante ao método `getMinimum`.

Por fim, a linha 50 no método `processGrades` chama o método `outputBarChart` para imprimir um gráfico de distribuição dos dados das notas utilizando uma técnica semelhante àquela na Figura 7.6. Nesse exemplo, calculamos manualmente o número das notas em cada categoria (isto é, 0–9, 10–19, ..., 90–99 e 100) simplesmente examinando o conjunto de notas. Nesse exemplo, as linhas 107–108 utilizam uma técnica semelhante à mostrada nas figuras 7.7 e 7.8 para calcular a frequência de notas em cada categoria. A linha 104 declara e cria o array `frequency` de 11 `int`s para armazenar a frequência de notas em cada categoria de nota. Para cada `grade` no array `grades`, as linhas 107–108 incrementam o elemento apropriado do array `frequency`. Para determinar qual elemento incrementar, a linha 108 divide a `grade` atual por 10 utilizando a divisão de inteiro. Por exemplo, se `grade` for `85`, a linha 108 incrementa `frequency[8]` para atualizar a contagem de notas no intervalo 80–89. As linhas 111–125 seguintes imprimem o gráfico de barras (ver Figura 7.15) com base nos valores do array `frequency`. Como as linhas 23–24 da Figura 7.6, as linhas 121–122 da Figura 7.14 utilizam um valor no array `frequency` para determinar o número de asteriscos a exibir em cada barra.

Classe `GradeBookTest` que demonstra a classe `GradeBook`

O aplicativo da Figura 7.15 cria um objeto da classe `GradeBook` (Figura 7.14) utilizando o array `int grades-Array` (declarado e inicializado na linha 10). As linhas 12–13 passam um nome de curso e o `gradesArray` para o construtor `GradeBook`. A linha 14 exibe uma mensagem de boas-vindas e a linha 15 invoca o método `processGrades` do objeto `GradeBook`. A saída resume as 10 notas em `myGradeBook`.

Observação de engenharia de software 7.1

Um aplicativo de teste é responsável por criar um objeto da classe sendo testado e fornecer-lhe dados. Esses dados poderiam vir de qualquer uma das várias fontes. Os dados de teste podem ser colocados diretamente em um array com um inicializador de array, podem vir do teclado, de um arquivo (como você verá no Capítulo 17) ou de uma rede (como você verá no Capítulo 27). Depois de passar esses dados para o construtor da classe para instanciar o objeto, o aplicativo de teste deve chamar o objeto para testar seus métodos e manipular seus dados. Reunir os dados em um aplicativo de teste como esse permite à classe manipular dados de várias fontes.

```
1  // Figura 7.15: GradeBookTest.java
2  // GradeBookTest cria um objeto GradeBook utilizando um array de notas,
3  // então invoca o método processGrades para analisá-las.
```

```
4  public class GradeBookTest
5  {
6     // método main inicia a execução de programa
7     public static void main( String[] args )
8     {
9        // array de notas de aluno
10       int[] gradesArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12       GradeBook myGradeBook = new GradeBook(
13          "CS101 Introduction to Java Programming", gradesArray );
14       myGradeBook.displayMessage();
15       myGradeBook.processGrades();
16    } // fim de main
17 } // fim da classe GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Figura 7.15 | `GradeBookTest` cria um objeto `GradeBook` usando um array de notas, então invoca o método `processGrades` para analisá-los.

7.9 Arrays multidimensionais

Os arrays multidimensionais com duas dimensões costumam ser usados para representar tabelas de valores consistindo em informações organizadas em linhas e colunas. Para identificar um elemento de tabela particular, devemos especificar dois índices. Por convenção, o primeiro identifica a linha do elemento e o segundo sua coluna. Os arrays que requerem dois índices para identificar um elemento particular são chamados **arrays bidimensionais**. (Os arrays multidimensionais podem ter mais de duas dimensões.) O Java não suporta arrays multidimensionais diretamente, mas permite especificar arrays unidimensionais cujos elementos também são arrays unidimensionais, alcançando assim o mesmo efeito. A Figura 7.16 ilustra um array bidimensional chamado `a` que contém três linhas e quatro colunas (isto é, um array três por quatro). Em geral, um array com *m* linhas e *n* colunas é chamado de ***array m por n***.

Cada elemento no array `a` é identificado na Figura 7.16 por uma expressão de acesso ao array da forma `a[row][column]`; `a` é o nome do array e `row` e `column` são os índices que unicamente identificam cada elemento no array `a` por número de linha e coluna. Note que todos os nomes dos elementos na linha 0 têm um primeiro índice de 0 e todos os nomes dos elementos na coluna 3 têm um segundo índice de 3.

Arrays de arrays unidimensionais

Como os arrays unidimensionais, os arrays multidimensionais podem ser inicializados com inicializadores de array em declarações. Um array bidimensional `b` com duas linhas e duas colunas poderia ser declarado e inicializado com **inicializadores de array aninhados** como a seguir:

```
int[][] b = { { 1, 2 }, { 3, 4 } };
```

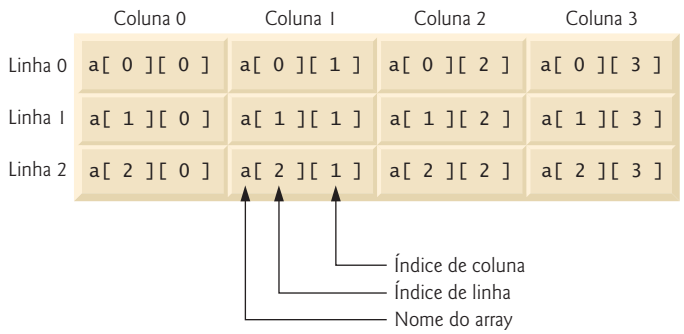


Figura 7.16 | O array bidimensional com três linhas e quatro colunas.

Os valores iniciais são agrupados por linha entre chaves. Então, `1` e `2` inicializam `b[0][0]` e `b[0][1]`, respectivamente, e `3` e `4` inicializam `b[1][0]` e `b[1][1]`, respectivamente. O compilador conta o número de inicializadores de array aninhados (representado por conjuntos de chaves dentro das chaves externas) para determinar o número de linhas no array `b`. O compilador conta os valores inicializadores no inicializador aninhado de array para uma linha para determinar o número de colunas nessa linha. Como veremos em breve, isso significa que as linhas podem ter diferentes comprimentos.

Os arrays multidimensionais são mantidos como arrays de arrays unidimensionais. Portanto o array `b` na declaração anterior é na realidade composto de dois arrays unidimensionais separados — um que contém o valor na primeira lista de inicializadores aninhados `{ 1, 2 }` e, o outro, que contém o valor na segunda lista de inicializadores aninhados `{ 3, 4 }`. Portanto, o próprio array `b` é um array de dois elementos, cada um desses elementos é um array unidimensional de valores `int`.

Arrays bidimensionais com linhas de diferentes comprimentos

A maneira como os arrays multidimensionais são representados os torna bem flexíveis. De fato, os comprimentos das linhas no array `b` não precisam ser os mesmos. Por exemplo:

```
int[][] b = { { 1, 2 }, { 3, 4, 5 } };
```

cria array de inteiros `b` com dois elementos (determinados pelo número de inicializadores de array aninhados) que representam as linhas do array bidimensional. Cada elemento de `b` é uma referência a um array unidimensional de variáveis `int`. O array `int` da linha `0` é um array unidimensional com dois elementos (`1` e `2`) e o array `int` da linha `1` é um array unidimensional com três elementos (`3`, `4` e `5`).

Criando arrays bidimensionais com expressões de criação de arrays

Um array multidimensional com o mesmo número de colunas em cada linha pode ser criado com uma expressão de criação de array. Por exemplo, as linhas a seguir declaram o array `b` e atribuem a ele uma referência para um array três por quatro:

```
int[][] b = new int[ 3 ][ 4 ];
```

Nesse caso, utilizamos os valores literais `3` e `4` para especificar o número de linhas e número de colunas, respectivamente, mas isso não é necessário. Programas também podem utilizar variáveis para especificar dimensões de array, porque `new` cria arrays em tempo de execução — não em tempo de compilação. Como com arrays unidimensionais, os elementos de um array multidimensional são inicializados quando o objeto array é criado.

Pode-se criar um array multidimensional em que cada linha tem um número diferente de colunas como mostrado a seguir:

```
int[][] b = new int[ 2 ][ ];   // cria 2 linhas
b[ 0 ] = new int[ 5 ]; // cria 5 colunas para a linha 0
b[ 1 ] = new int[ 3 ]; // cria 3 colunas para a linha 1
```

As instruções anteriores criam um array bidimensional com duas linhas. A linha `0` tem cinco colunas e linha `1` tem três colunas.

Exemplo de array bidimensional: exibindo valores de elemento

A Figura 7.17 demonstra a inicialização de arrays bidimensionais com inicializadores de array e a utilização de loops `for` aninhados para **percorrer** os arrays (isto é, manipular cada elemento de cada array). O método `main` da classe `InitArray` declara dois arrays. A declaração de `array1` (linha 9) utiliza inicializadores de array aninhados para inicializar a primeira linha do array para os valores 1, 2 e 3 e, a segunda linha, para os valores 4, 5 e 6. A declaração de `array2` (linha 10) utiliza inicializadores aninhados de diferentes comprimentos. Nesse caso, a primeira linha é inicializada para dois elementos com os valores 1 e 2, respectivamente. A segunda linha é inicializada para um elemento com o valor 3. A terceira linha é inicializada para três elementos com os valores 4, 5 e 6, respectivamente.

```
1  // Figura 7.17: InitArray.java
2  // Inicializando arrays bidimensionais.
3
4  public class InitArray
5  {
6     // cria e gera saída de arrays bidimensionais
7     public static void main( String[] args )
8     {
9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // exibe array1 por linha
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // exibe array2 por linha
17    } // fim de main
18
19    // gera saída de linhas e colunas de um array bidimensional
20    public static void outputArray(int[][] array )
21    {
22       // faz um loop pelas linhas do array
23       for ( int row = 0; row < array.length; row++ )
24       {
25          // faz um loop pelas colunas da linha atual
26          for ( int column = 0; column < array[ row ].length; column++ )
27             System.out.printf( "%d  ", array[ row ][ column ] );
28
29          System.out.println(); // inicia nova linha de saída
30       } // fim do for externo
31    } // fim do método outputArray
32 } // fim da classe InitArray
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Figura 7.17 | Inicializando arrays bidimensionais.

As linhas 13 e 16 chamam o método `outputArray` (linhas 20–31) para gerar saída aos elementos de `array1` e `array2`, respectivamente. O parâmetro do método `outputArray` — `int[][] array` — indica que o método recebe um array bidimensional. A instrução `for` (linhas 23–30) gera saída das linhas de um array bidimensional. Na condição de continuação do loop da instrução `for` externa, a expressão `array.length` determina o número de linhas no array. Na instrução `for` interna, a expressão `array[row].length` determina o número de colunas na linha atual do array. Essa condição permite ao loop determinar o número exato de colunas em cada linha.

Manipulações de arrays multidimensionais comuns realizadas com as instruções `for`

Muitas manipulações de array comuns utilizam as instruções `for`. Como um exemplo, a seguinte instrução `for` configura todos os elementos na linha 2 do array `a` na Figura 7.16 como zero:

```
for ( int column = 0; column < a[ 2 ].length; column++)
   a[ 2 ][ column ] = 0;
```

Especificamos a linha 2; portanto, sabemos que o primeiro índice é sempre 2 (0 é a primeira linha e 1 é a segunda linha). Esse loop `for` varia somente o segundo índice (isto é, o índice de coluna). Se a linha 2 do array `a` contiver quatro elementos, então a instrução `for` anterior será equivalente às instruções de atribuição:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

A seguinte instrução `for` aninhada soma os valores de todos os elementos no array `a`:

```
int total = 0;
for ( int row = 0; row < a.length; row++ )
{
   for ( int column = 0; column < a[ row ].length; column++ )
      total += a[ row ][ column ];
} // fim do for externo
```

Essas instruções `for` aninhadas somam os elementos do array uma linha por vez. A instrução `for` externa começa configurando o índice `row` como 0 para que os elementos da primeira linha possam ser somados pela instrução `for` interna. O `for` externo então incrementa `row` a 1 para que a segunda linha possa ser somada. Então, o `for` externo incrementa `row` para 2 de modo que a terceira linha pode ser somada. A variável `total` pode ser exibida quando a instrução `for` externa terminar. No próximo exemplo, mostramos como processar um array bidimensional de maneira semelhante utilizando instruções `for` aprimoradas aninhadas.

7.10 Estudo de caso: classe GradeBook utilizando um array bidimensional

Na Seção 7.8, apresentamos a classe `GradeBook` (Figura 7.14), que utilizou um array unidimensional para armazenar notas de aluno de uma única prova. Na maioria dos semestres, os alunos fazem vários exames. É provável que os professores queiram analisar as notas do semestre inteiro, de um único aluno e de toda a classe.

Armazenando notas de aluno em um array bidimensional na classe GradeBook

A Figura 7.18 contém uma versão da classe `GradeBook` que utiliza um array bidimensional `grades` para armazenar as notas de vários alunos em múltiplos exames. Cada linha do array representa as notas de um único aluno para o curso inteiro, e cada coluna representa as notas de todos os alunos que fizeram uma prova específica. Um aplicativo como `GradeBookTest` (Figura 7.19) passa o array como um argumento para o construtor `GradeBook`. Nesse exemplo, utilizamos um array dez por três contendo as notas de três exames de dez alunos. Cinco métodos realizam as manipulações de array para processar as notas. Cada método é semelhante a sua contraparte na versão anterior de um array unidimensional de classe `GradeBook` (Figura 7.14). O método `getMinimum` (linha 52–70) determina a nota mais baixa de qualquer aluno no semestre. O método `getMaximum` (linhas 73–91) determina a nota mais alta de qualquer aluno no semestre. O método `getAverage` (linhas 94–104) determina a média semestral de um aluno particular. O método `outputBarChart` (linhas 107–137) gera saída de um gráfico de barras da distribuição de todas as notas de aluno durante o semestre. O método `outputGrades` (linhas 140–164) gera saída do array bidimensional em um formato tabular, junto com a média semestral de cada aluno.

```
1  // Figura 7.18: GradeBook.java
2  // classe GradeBook com um array bidimensional para armazenar notas
3
4  public class GradeBook
5  {
6     private String courseName; // nome do curso que este livro de nota representa
7     private int[][] grades; // array bidimensional de notas de aluno
8
9     // construtor de dois argumentos inicializa courseName e array de notas
10    public GradeBook( String name, int[][] gradesArray )
```

```
11       {
12          courseName = name; // inicializa courseName
13          grades = gradesArray; // níveis de armazenamento
14       } // fim do construtor GradeBook de dois argumentos
15
16       // método para configurar o nome do curso
17       public void setCourseName( String name )
18       {
19          courseName = name; // armazena o nome do curso
20       } // fim do método setCourseName
21
22       // método para recuperar o nome do curso
23       public String getCourseName()
24       {
25          return courseName;
26       } // fim do método getCourseName
27
28       // exibe uma mensagem de boas-vindas para o usuário GradeBook
29       public void displayMessage()
30       {
31          // getCourseName obtém o nome do curso
32          System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33             getCourseName() );
34       } // fim do método displayMessage
35
36       // realiza várias operações nos dados
37       public void processGrades()
38       {
39          // gera saída de array de notas
40          outputGrades();
41
42          // chama métodos getMinimum e getMaximum
43          System.out.printf( "\n%s %d\n%s %d\n\n",
44             "Lowest grade in the grade book is", getMinimum(),
45             "Highest grade in the grade book is", getMaximum() );
46
47          // gera saída de gráfico de distribuição de notas de todas as notas em todos os testes
48          outputBarChart();
49       } // fim do método processGrades
50
51       // localiza nota mínima
52       public int getMinimum()
53       {
54          // assume que o primeiro elemento de array de notas é o menor
55          int lowGrade = grades[ 0 ][ 0 ];
56
57          // faz um loop pelas linhas do array de notas
58          for ( int[] studentGrades : grades )
59          {
60             // faz um loop pelas colunas da linha atual
61             for ( int grade : studentGrades )
62             {
63                // se a nota for menor que lowGrade, atribui a nota a lowGrade
64                if ( grade < lowGrade )
65                   lowGrade = grade;
66             } // fim do for interno
67          } // fim do for externo
68
69          return lowGrade; // retorna nota mais baixa
70       } // fim do método getMinimum
71
72       // localiza nota máxima
73       public int getMaximum()
74       {
75          // assume que o primeiro elemento de array de notas é o maior
76          int highGrade = grades[ 0 ][ 0 ];
77
```

```
78          // faz um loop pelas linhas do array de notas
79          for ( int[] studentGrades : grades )
80          {
81             // faz um loop pelas colunas da linha atual
82             for ( int grade : studentGrades )
83             {
84                // se a nota for maior que highGrade, atribui essa nota a highGrade
85                if ( grade > highGrade )
86                   highGrade = grade;
87             } // fim do for interno
88          } // fim do for externo
89
90          return highGrade; // retorna nota mais alta
91       } // fim do método getMaximum
92
93       // determina a média do conjunto de particular de notas
94       public double getAverage( int[] setOfGrades )
95       {
96          int total = 0; // inicializa o total
97
98          // soma notas de um aluno
99          for ( int grade : setOfGrades )
100            total += grade;
101
102         // retorna média de notas
103         return (double) total / setOfGrades.length;
104      } // fim do método getAverage
105
106      // gera saída do gráfico de barras para exibir distribuição total de notas
107      public void outputBarChart()
108      {
109         System.out.println( "Overall grade distribution:" );
110
111         // armazena frequência de notas em cada intervalo de 10 notas
112         int[] frequency = new int[ 11 ];
113
114         // para cada nota em GradeBook, incrementa a frequência certa
115         for ( int[] studentGrades : grades )
116         {
117            for ( int grade : studentGrades )
118               ++frequency[ grade / 10 ];
119         } // fim do for externo
120
121         // para cada frequência de nota, imprime barra no gráfico
122         for ( int count = 0; count < frequency.length; count++ )
123         {
124            // imprime rótulo de barra ( "00-09: ", ..., "90-99: ", "100: " )
125            if ( count == 10 )
126               System.out.printf( "%5d: ", 100 );
127            else
128               System.out.printf( "%02d-%02d: ",
129                  count * 10, count * 10 + 9  );
130
131            // imprime a barra de asteriscos
132            for ( int stars = 0; stars < frequency[ count ]; stars++ )
133               System.out.print( "*" );
134
135            System.out.println(); // inicia uma nova linha de saída
136         } // fim do for externo
137      } // fim do método outputBarChart
138
139      // gera a saída do conteúdo do array de notas
140      public void outputGrades()
141      {
142         System.out.println( "The grades are:\n" );
143         System.out.print( "            " ); // alinha títulos de coluna
144
145         // cria um título de coluna para cada um dos testes
146         for ( int test = 0; test < grades[ 0 ].length; test++ )
147            System.out.printf( "Test %d  ", test + 1 );
```

```
148
149         System.out.println( "Average" ); // título da coluna de média do aluno
150
151         // cria linhas/colunas de texto que representam notas de array
152         for ( int student = 0; student < grades.length; student++ )
153         {
154            System.out.printf( "Student %2d", student + 1 );
155
156            for ( int test : grades[ student ] ) // gera saída de notas do aluno
157               System.out.printf( "%8d", test );
158
159            // chama método getAverage para calcular a média do aluno;
160            // passa linha de notas como o argumento para getAverage
161            double average = getAverage( grades[ student ] );
162            System.out.printf( "%9.2f\n", average );
163         } // fim do for externo
164      } // fim do método outputGrades
165   } // fim da classe GradeBook
```

Figura 7.18 | Classe GradeBook utilizando um array bidimensional para armazenar notas.

Métodos getMinimum e getMaximum

Os métodos `getMinimum`, `getMaximum`, `outputBarChart` e `outputGrades` fazem loop pelo array `grades` utilizando instruções `for` aninhadas — por exemplo, a instrução `for` aprimorada aninhada a partir da declaração de método `getMinimum` (linhas 58–67). A instrução `for` aprimorada externa itera pelo array bidimensional `grades`, atribuindo linhas sucessivas ao parâmetro `studentGrades` em sucessivas iterações. Os colchetes que se seguem ao nome do parâmetro indicam que `studentGrades` referencia o array `int` unidimensional — isto é, uma linha no array `grades` que contém a nota de um aluno. Para localizar a nota geral mais baixa, a instrução `for` interna compara os elementos do array unidimensional `studentGrades` atual com a variável `lowGrade`. Por exemplo, na primeira iteração do `for` externo, a linha 0 de `grades` é atribuída ao parâmetro `studentGrades`. A instrução `for` aprimorada interna então faz um loop por `studentGrades` e compara cada valor `grade` com `lowGrade`. Se uma nota for menor que `lowGrade`, `lowGrade` é configurado como essa nota. Na segunda iteração da instrução `for` aprimorada externa, a linha 1 de `grades` é atribuída a `studentGrades` e os elementos dessa linha são comparados com a variável `lowGrade`. Isso se repete até que todas as linhas de `grades` tenham sido percorridas. Quando a execução da instrução aninhada é concluída, `lowGrade` contém a nota mais baixa no array bidimensional. O método `getMaximum` funciona de maneira semelhante ao método `getMinimum`.

Método outputBarChart

O método `outputBarChart` na Figura 7.18 é quase idêntico àquele na Figura 7.14. Contudo, para imprimir a distribuição geral das notas para um semestre inteiro, o método aqui utiliza instruções aprimoradas `for` aninhadas (linhas 115–119) para criar o array unidimensional `frequency` com base em todas as notas no array bidimensional. O resto do código em cada um dos dois métodos `outputBarChart` que exibe o gráfico é idêntico.

Método outputGrades

O método `outputGrades` (linhas 140–164) utiliza instruções `for` aninhadas para gerar saída de valores do array `grades` e a média semestral de cada aluno. A saída (Figura 7.19) mostra o resultado, que é semelhante ao formato tabular de um livro de notas de um professor de física. As linhas 146–147 imprimem os títulos de coluna para cada teste. Utilizamos uma instrução `for` controlada por contador aqui para que possamos identificar cada teste com um número. De maneira semelhante, a instrução `for` nas linhas 152–163 gera primeiro a saída de um rótulo de linha utilizando uma variável contadora para identificar cada aluno (linha 154). Embora os índices de array iniciem em 0, as linhas 147 e 154 geram saída de `test + 1` e `student + 1`, respectivamente, para produzir números de teste e de aluno que iniciam em 1 (ver Figura 7.19). A instrução `for` interna (linhas 156–157) utiliza a variável contadora `student` da instrução `for` externa para fazer um loop por uma linha específica do array `grades` e gerar saída da nota de teste de cada aluno. Observe que uma instrução `for` aprimorada pode ser aninhada em uma instrução `for` controlada por contador e vice-versa. Por fim, a linha 161 obtém a média de semestre de cada aluno passando a linha atual de `grades` (isto é, `grades[student]`) para o método `getAverage`.

Método getAverage

O método `getAverage` (linhas 94–104) aceita um argumento — um array unidimensional dos resultados de teste de um aluno particular. Quando a linha 161 chama `getAverage`, o argumento é `grades[student]`, que especifica que uma linha particular do array bidimensional `grades` deve ser passada para `getAverage`. Por exemplo, com base no array criado na Figura 7.19, o argumento `grades[1]` representa os três valores (um array unidimensional de notas) armazenados na linha `1` do array bidimensional `grades`. Lembre-se de que um array bidimensional é um array cujos elementos são arrays unidimensionais. O método `getAverage` calcula a soma dos elementos do array, divide o total pelo número de resultados do teste e retorna o resultado de ponto flutuante como um valor `double` (linha 103).

Classe GradeBookTest que demonstra a classe GradeBook

A Figura 7.19 cria um objeto da classe GradeBook (Figura 7.18) utilizando o array bidimensional de ints chamado gradesArray (declarado e inicializado nas linhas 10–19). As linhas 21–22 passam um nome de curso e o gradesArray para o construtor GradeBook. As linhas 23–24 então invocam os métodos displayMessage e processGrades de myGradeBook para exibir uma mensagem de boas-vindas e obter um relatório que resume as notas semestrais dos alunos, respectivamente.

```
1  // Figura 7.19: GradeBookTest.java
2  // GradeBookTest cria o objeto GradeBook usando um array bidimensional
3  // das notas, então invoca o método proctessGrades para analisá-las.
4  public class GradeBookTest
5  {
6     // método main inicia a execução de programa
7     public static void main( String[] args )
8     {
9        // array bidimensional de notas de aluno
10       int[][] gradesArray = { { 87, 96, 70 },
11                               { 68, 87, 90 },
12                               { 94, 100, 90 },
13                               { 100, 81, 82 },
14                               { 83, 65, 85 },
15                               { 78, 87, 65 },
16                               { 85, 75, 83 },
17                               { 91, 94, 100 },
18                               { 76, 72, 84 },
19                               { 87, 93, 73 } };
20
21       GradeBook myGradeBook = new GradeBook(
22          "CS101 Introduction to Java Programming", gradesArray );
23       myGradeBook.displayMessage();
24       myGradeBook.processGrades();
25    } // fim de main
26 } // fim da classe GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Figura 7.19 | GradeBookTest cria o objeto GradeBook utilizando um array bidimensional de notas e invoca o método processGrades para analisá-las.

7.11 Listas de argumentos de comprimento variável

Com **listas de argumentos de tamanho variável**, você pode criar métodos que recebem um número não especificado de argumentos. Um tipo seguido por **reticências (...)** na lista de parâmetros de um método indica que o método recebe um número variável de argumentos desse tipo particular. Essa utilização das reticências só pode ocorrer uma única vez em uma lista de parâmetros, e as reticências, juntamente com seu tipo, devem ser colocadas no fim da lista de parâmetros. Embora você possa utilizar sobrecarga de método e passagem de array para realizar grande parte do que é realizado com listas de argumentos de comprimento variável, é mais conciso utilizar as reticências na lista de parâmetros de um método.

A Figura 7.20 demonstra o método `average` (linhas 7–16), que recebe uma sequência de comprimento variável de `double`s. O Java trata a lista de argumentos de comprimento variável como um array cujos elementos são todos do mesmo tipo. Portanto, o corpo de método pode manipular o parâmetro `numbers` como um array de `double`s. As linhas 12–13 usam o loop `for` aprimorado para percorrer o array e calcular o total dos `double`s no array. A linha 15 acessa `numbers.length` para obter o tamanho do array `numbers` para uso no cálculo da média. As linhas 29, 31 e 33 em `main` chamam o método `average` com dois, três e quatro argumentos, respectivamente. O método `average` tem uma lista de argumentos de comprimento variável (linha 7), então ele pode calcular a média de quantos argumentos `double` o chamador passar. A saída mostra que cada chamada ao método `average` retorna o valor correto.

```
1  // Figura 7.20: VarargsTest.java
2  // Utilizando listas de argumentos de comprimento variável.
3
4  public class VarargsTest
5  {
6     // calcula a média
7     public static double average( double... numbers )
8     {
9        double total = 0.0; // inicializa o total
10
11       // calcula o total utilizando a instrução for aprimorada
12       for ( double d : numbers )
13          total += d;
14
15       return total / numbers.length;
16    } // fim do método average
17
18    public static void main( String[] args )
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4
26          = %.1f\n\n", d1, d2, d3, d4 );
27
28       System.out.printf( "Average of d1 and d2 is %.1f\n",
29          average( d1, d2 ) );
30       System.out.printf( "Average of d1, d2 and d3 is %.1f\n",
31          average( d1, d2, d3 ) );
32       System.out.printf( "Average of d1, d2, d3 and d4 is %.1f\n",
33          average( d1, d2, d3, d4 ) );
34    } // fim de main
35 } // fim da classe VarargsTest
```

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Figura 7.20 | Utilizando listas de argumentos de comprimento variável.

Erro comum de programação 7.6

Inserir reticências indicando uma lista de argumentos de tamanho variável no meio de uma lista de parâmetros é um erro de sintaxe. As reticências só podem ser colocadas no fim da lista de parâmetros.

7.12 Utilizando argumentos de linha de comando

Em muitos sistemas é possível passar argumentos a partir da linha de comando (esses são conhecidos como **argumentos de linha de comando**) para um aplicativo, incluindo um parâmetro do tipo `String[]` (isto é, um array de `String`s), na lista de parâmetro de `main`, exatamente como fizemos em cada aplicativo do livro. Por convenção, esse parâmetro é chamado `args`. Quando um aplicativo é executado utilizando-se o comando `java`, o Java passa os argumentos da linha de comando que aparecem depois do nome de classe do comando `java` para o método `main` do aplicativo como `String`s do array `args`. O número de argumentos de linha de comando é obtido acessando o atributo `length` do array. Por exemplo, o comando `"java MyClass a b"` passa dois argumentos de linha de comando, `a` e `b`, para o aplicativo `MyClass`. Observe que os argumentos de linha de comando são separados por espaço em branco, não vírgulas. Quando esse comando é executado, o método `main` de `MyClass` recebe o array de dois elementos `args` (isto é, `args.length` é `2`) no qual `args[0]` contém a `String` `"a"` e `args[1]` contém a `String` `"b"`. Utilizações comuns de argumentos de linha de comando incluem passar opções e nomes de arquivo para aplicativos.

A Figura 7.21 utiliza três argumentos de linha de comando para inicializar um array. Quando o programa executa, se `args.length` não for `3`, o programa imprimirá uma mensagem de erro e terminará (linhas 9–12). Caso contrário, as linhas 14–32 inicializam e exibem o array com base nos valores dos argumentos de linha de comando.

```
1  // Figura 7.21: InitArray.java
2  // Inicializando um array com argumentos de linha de comando.
3
4  public class InitArray
5  {
6     public static void main(String[] args )
7     {
8        // verifica o número de argumentos de linha de comando
9        if (args.length != 3 )
10          System.out.println(
11             "Error: Please re-enter the entire command, including\n" +
12             "an array size, initial value and increment." );
13       else
14       {
15          // obtém o tamanho do array a partir do primeiro argumento de linha de comando
16          int arrayLength = Integer.parseInt( args[ 0 ] );
17          int[] array = new int[ arrayLength ]; // cria o array
18
19          // obtém o valor inicial e o incrementa a partir dos argumentos de linha de comando
20          int initialValue = Integer.parseInt( args[ 1 ] );
21          int increment = Integer.parseInt( args[ 2 ] );
22
23          // calcula o valor de cada elemento do array
24          for ( int counter = 0; counter < array.length; counter++ )
25             array[ counter ] = initialValue + increment * counter;
26
27          System.out.printf( "%s%8s\n", "Index", "Value" );
28
29          // exibe o valor e o índice de array
30          for ( int counter = 0; counter < array.length; counter++ )
31             System.out.printf( "%5d%8d\n", counter, array[ counter ] );
32       } // fim de else
33    } // fim de main
34 } // fim da classe InitArray
```

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Figura 7.21 | Inicializando um array com argumentos de linha de comando.

Os argumentos de linha de comando tornam-se disponíveis para `main` como `String`s em `args`. A linha 16 obtém `args[0]` — uma `String` que especifica o tamanho do array — e a converte em um valor `int` que o programa utiliza para criar o array da linha 17. O método `static` `parseInt` da classe `Integer` converte seu argumento `String` em um `int`.

As linhas 20–21 convertem os argumentos de linha de comando `args[1]` e `args[2]` em valores `int` e os armazenam em `initialValue` e `increment`, respectivamente. As linhas 24–25 calculam o valor de cada elemento do array.

A saída da primeira execução mostra que o aplicativo recebeu um número insuficiente de argumentos de linha de comando. A segunda execução utiliza os argumentos 5, 0 e 4 para especificar o tamanho do array (5), o valor do primeiro elemento (0) e o incremento de cada valor do array (4), respectivamente. A saída correspondente mostra que esses valores criam um array contendo os inteiros 0, 4, 8, 12 e 16. A saída da terceira execução mostra que os argumentos de linha de comando 8, 1 e 2 produzem um array cujos 8 elementos são os inteiros ímpares não negativos de 1–15.

7.13 Classe Arrays

A classe **`Arrays`** ajuda a evitar a reinventar a roda fornecendo métodos `static` para manipulações de array comuns. Esses métodos incluem **`sort`** para ordenar um array (isto é, organizando os elementos na ordem crescente), **`binarySearch`** para pesquisar um array (isto é, determinando se um array contém um valor específico e, nesse caso, onde o valor está localizado), **`equals`** para comparar arrays e **`fill`** para inserir valores em um array. Esses métodos são sobrecarregados para arrays de tipo primitivo e arrays de objetos. O foco desta seção é utilizar as capacidades internas fornecidas pela API do Java. O Capítulo 19, "Pesquisa, classificação e Big O", mostra como implementar seus próprios algoritmos de ordenação e pesquisa, bem importantes para pesquisadores e estudantes da ciência da computação.

A Figura 7.22 utiliza métodos de array `sort`, `binarySearch`, `equals` e `fill`, e mostra como copiar arrays com o método `static` **`arraycopy`** da classe `System`. Em `main`, a linha 11 classifica os elementos do array `doubleArray`. O método `static` `sort` da classe `Arrays` ordena os elementos do array na ordem crescente por padrão. Discutiremos mais adiante neste capítulo como classificar em ordem *decrescente*. Versões sobrecarregadas de `sort` permitem-lhe classificar um intervalo específico de elementos. As linhas 12–15 imprimem o array classificado.

```
1  // Figura 7.22: ArrayManipulations.java
2  // Métodos da classe Arrays e System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations
6  {
7     public static void main( String[] args )
8     {
9        // classifica doubleArray em ordem crescente
10       double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort( doubleArray );
12       System.out.printf( "\ndoubleArray: " );
13
```

```
14          for ( double value : doubleArray )
15             System.out.printf( "%.1f ", value );
16
17          // preenche o array de 10 elementos com 7s
18          int[] filledIntArray = new int[ 10 ];
19          Arrays.fill( filledIntArray, 7 );
20          displayArray( filledIntArray, "filledIntArray" );
21
22          // copia array intArray em array intArrayCopy
23          int[] intArray = { 1, 2, 3, 4, 5, 6 };
24          int[] intArrayCopy = new int[ intArray.length ];
25          System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
26          displayArray( intArray, "intArray" );
27          displayArray( intArrayCopy, "intArrayCopy" );
28
29          // compara a igualdade de intArray e intArrayCopy
30          boolean b = Arrays.equals( intArray, intArrayCopy );
31          System.out.printf( "\n\nintArray %s intArrayCopy\n",
32             ( b ? "==" : "!=" ) );
33
34          // compara a igualdade de intArray e filledIntArray
35          b = Arrays.equals( intArray, filledIntArray );
36          System.out.printf( "intArray %s filledIntArray\n",
37             ( b ? "==" : "!=" ) );
38
39          // pesquisa em intArray o valor 5
40          int location = Arrays.binarySearch( intArray, 5 );
41
42          if ( location >= 0 )
43             System.out.printf(
44                "Found 5 at element %d in intArray\n", location );
45          else
46             System.out.println( "5 not found in intArray" );
47
48          // pesquisa em intArray o valor 8763
49          location = Arrays.binarySearch( intArray, 8763 );
50
51          if ( location >= 0 )
52             System.out.printf(
53                "Found 8763 at element %d in intArray\n", location );
54          else
55             System.out.println( "8763 not found in intArray" );
56       } // fim de main
57
58       // gera saída de valores em cada array
59       public static void displayArray( int[] array, String description )
60       {
61          System.out.printf( "\n%s: ", description );
62
63          for ( int value : array )
64             System.out.printf( "%d ", value );
65       } // fim do método displayArray
66    } // fim da classe ArrayManipulations
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Figura 7.22 | Métodos da classe `Arrays`.

A linha 19 chama o método `static` `fill` da classe `Arrays` para preencher todos os 10 elementos de `filledIntArray` com 7s. Versões sobrecarregadas de `fill` permitem-lhe preencher um intervalo específico de elementos com o mesmo valor. A linha 20 chama o método `displayArray` da nossa classe (declarada nas linhas 59–65) para imprimir o conteúdo de `filledIntArray`.

A linha 25 copia os elementos de intArray para intArrayCopy. O primeiro argumento (intArray) passado para o método System arraycopy é o array a partir do qual os elementos devem ser copiados. O segundo argumento (0) é o índice que especifica o ponto inicial no intervalo de elementos a copiar a partir do array. Esse valor pode ser qualquer índice de array válido. O terceiro argumento (intArrayCopy) especifica o array de destino que armazenará a cópia. O quarto argumento (0) especifica o índice no array de destino em que o primeiro elemento copiado deve ser armazenado. O último argumento especifica o número de elementos a ser copiado a partir do array no primeiro argumento. Nesse caso, copiamos todos os elementos no array.

As linhas 30 e 35 chamam o método static equals da classe Arrays para determinar se todos os elementos de dois arrays são equivalentes. Se os arrays contiverem os mesmos elementos na mesma ordem, o método retorna true; caso contrário, retorna false.

As linhas 40 e 49 chamam o método static binarySearch da classe Arrays para realizar uma pesquisa binária em intArray, utilizando o segundo argumento (5 e 8763, respectivamente) como a chave. Se value for encontrado, binarySearch retorna o índice do elemento; caso contrário, binarySearch retorna um valor negativo. O valor negativo retornado é baseado no ponto de inserção da chave de pesquisa — o índice em que a chave seria inserida no array se fôssemos realizar uma operação de inserção. Depois de binarySearch determinar o ponto de inserção, ele muda seu sinal para negativo e subtrai 1 para obter o valor de retorno. Por exemplo, na Figura 7.22, o ponto de inserção para o valor 8763 é o elemento com índice 6 no array. O método binarySearch muda o ponto de inserção para –6, subtrai 1 dele e retorna o valor –7. Subtrair 1 do ponto de inserção garante que o método binarySearch retorna valores positivos (>=0) se e somente se a chave for encontrada. Esse valor de retorno é útil para inserir elementos em um array classificado. O Capítulo 19 discute em detalhes pesquisas binárias.

Erro comum de programação 7.7

Passar um array não classificado para binarySearch é um erro de lógica — o valor retornado é indefinido.

7.14 Introdução a coleções e classe ArrayList

A API do Java fornece várias estruturas de dados predefinidas, chamadas **coleções**, utilizadas para armazenar grupos de objetos relacionados. Essas classes fornecem métodos eficientes que organizam, armazenam e recuperam seus dados sem que seja necessário conhecer como os dados são armazenados. Isso reduz o tempo de desenvolvimento de aplicativos.

Você utilizou array para armazenar sequências de objetos. Arrays não alteram automaticamente seu tamanho em tempo de execução para acomodar elementos adicionais. A classe de coleção **ArrayList <T>** (do pacote java.util) fornece uma solução conveniente a esse problema — ela pode alterar dinamicamente seu tamanho para acomodar mais elementos. O T é um espaço reservado — ao declarar um novo ArrayList, substitua-o pelo tipo de elementos que você quer que ArrayList contenha. Isso é semelhante a especificar o tipo ao declarar um array, exceto que apenas tipos não primitivos podem ser utilizados com essas classes de coleção. Por exemplo:

```
ArrayList< String > list;
```

declara list como uma coleção ArrayList que só pode armazenar Strings. Classes com essa espécie de marcador de lugar que podem ser utilizadas com qualquer tipo são chamadas **classes genéricas**. Classes de coleção genéricas e classes genéricas adicionais são discutidas nos capítulos 20–21, respectivamente. A Figura 7.23 mostra alguns métodos comuns da classe ArrayList<T>.

A Figura 7.24 demonstra algumas capacidades ArrayList comuns. A linha 10 cria uma nova ArrayList vazia de Strings com uma capacidade inicial de 10 elementos. A capacidade indica quantos itens ArrayList podem conter sem que seu tamanho aumente. ArrayList é implementada utilizando um array nos bastidores. Quando o tamanho de ArrayList aumenta, ela precisa criar um array

Método	Descrição
add	Adiciona um elemento ao fim de ArrayList.
clear	Remove todos os elementos de ArrayList.
contains	Retorna true se ArrayList contiver o elemento especificado; do contrário, retorna false.
get	Retorna o elemento no índice especificado.
indexOf	Retorna o índice da primeira ocorrência do elemento especificado em ArrayList.
remove	Remove a primeira ocorrência do valor especificado.
remove	Remove o elemento no índice especificado.
size	Retorna o número de elementos armazenados no ArrayList.
trimToSize	Reduz a capacidade de ArrayList de acordo com o número de elementos atual.

Figura 7.23 | Alguns métodos e propriedades da classe ArrayList<T>.

interno maior e copiar cada elemento para o novo array. Essa é uma operação demorada. Não seria eficiente se o tamanho de `ArrayList` aumentasse toda vez que um elemento é adicionado. Em vez disso, o tamanho só aumenta quando um elemento é adicionado *e* o número de elementos é igual à capacidade — isto é, não há nenhum espaço para o novo elemento.

O método **`add`** adiciona elementos à `ArrayList` (linhas 12–13). O método `add` com um argumento acrescenta seu argumento ao fim de `ArrayList`. O método `add` com dois argumentos insere um novo elemento na posição especificada. O primeiro argumento é um índice. Como com arrays, índices de coleção iniciam em zero. O segundo argumento é o valor a ser inserido nesse índice. Todos os elementos no índice especificado e acima são deslocados para cima por uma posição. Isso normalmente é mais lento do que adicionar um elemento ao final de `ArrayList`.

As linhas 20–21 exibem os itens em `ArrayList`. O método **`size`** retorna o número de elementos atualmente em `ArrayList`. O método **`get`** `ArrayLists` (linha 21) obtém o elemento em um índice especificado. As linhas 24–25 exibem os elementos novamente invocando o método display (definido nas linhas 46–55). As linhas 27–28 adicionam mais dois elementos a `ArrayList`, então a linha 29 exibe os elementos novamente para confirmar se os dois elementos foram adicionados ao final da coleção.

O método **`remove`** é utilizado para remover um elemento com um valor específico (linha 31). Observe que ele só remove o primeiro elemento. Se não houver esse elemento em `ArrayList`, `remove` não faz nada. Uma versão sobrecarregada do método remove o elemento no índice especificado (linha 34). Quando um elemento é removido por meio de qualquer um desses métodos, todos os elementos acima desse índice são deslocados para baixo por um.

A linha 39 utiliza o método **`contains`** para verificar se um item está em `ArrayList`. O método `contains` retorna `true` se o elemento for localizado em `ArrayList`, e `false` caso contrário. O método compara seu argumento com cada elemento de `ArrayList` na ordem, portanto utilizar `contains`. A linha 42 exibe o tamanho de `ArrayList`.

```
1  // Figura 7.24: ArrayListCollection.java
2  // Demonstração da coleção ArrayList genérica.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main( String[] args )
8     {
9        // cria um novo ArrayList de strings
10       ArrayList< String > items = new ArrayList< String >();
11
12       items.add( "red" ); // acrescenta um item à lista
13       items.add( 0, "yellow" ); // insere o valor no índice 0
14
15       // cabeçalho
16       System.out.print(
17          "Display list contents with counter-controlled loop:" );
18
19       // exiba as cores na lista
20       for ( int i = 0; i < items.size(); i++ )
21          System.out.printf( " %s", items.get( i ) );
22
23       // exibe as cores utilizando foreach no método display
24       display( items,
25          "\nDisplay list contents with enhanced for statement:" );
26
27       items.add( "green" ); // adiciona "green" ao final da lista
28       items.add( "yellow" ); // adiciona "yellow" ao final da lista
29       display( items, "List with two new elements:" );
30
31       items.remove( "yellow" ); // remove o primeiro "yellow"
32       display( items, "Remove first instance of yellow:" );
33
34       items.remove( 1 ); // remove o item no índice 1
35       display( items, "Remove second list element (green):" );
36
37       // verifica se um valor está em List
38       System.out.printf( "\"red\" is %sin the list\n",
39          items.contains( "red" ) ? "": "not " );
40
41       // exibe o número de elementos em List
42       System.out.printf( "Size: %s\n", items.size() );
43    } // fim de main
44
```

```
45      // exibe os elementos do ArrayList no console
46      public static void display( ArrayList< String > items, String header )
47      {
48         System.out.print( header ); // exibe o cabeçalho
49
50         // exibe cada elemento nos itens
51         for ( String item : items )
52            System.out.printf( " %s", item );
53
54         System.out.println(); // exibe o fim de linha
55      } // fim do método display
56   } // fim da classe ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Figura 7.24 | Demonstração da coleção `ArrayList<T>` genérica.

7.15 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando arcos

Utilizando os recursos de imagens gráficas do Java, podemos criar desenhos complexos que seriam mais tediosos de codificar linha por linha. Nas figuras 7.25–7.26, utilizamos os arrays e instruções de repetição para desenhar um arco-íris utilizando o método `Graphics` `fillArc`. Desenhar arcos no Java é semelhante a desenhar ovais — um arco é simplesmente uma fatia de uma oval.

A Figura 7.25 inicia com as costumeiras instruções `import` para criar desenhos (linhas 3–5). As linhas 10–11 declaram e criam duas novas constantes de cor — `VIOLET` e `INDIGO`. Como você pode saber, as cores de um arco-íris são vermelho, laranja, amarelo, azul, índigo, verde e violeta. O Java tem constantes predefinidas apenas para as cinco primeiras cores. As linhas 15–17 inicializam um array com as cores do arco-íris, iniciando primeiro com os arcos mais internos. O array inicia com dois elementos `Color.WHITE`, que, como você logo verá, serão para desenhar os arcos vazios no centro do arco-íris. Observe que as variáveis de instância podem ser inicializadas quando elas são declaradas, como mostrado nas linhas 10–17. O construtor (linhas 20–23) contém uma única instrução que chama o método **`setBackground`** (que é herdado da classe `JPanel`) com o parâmetro `Color.WHITE`. O método `setBackground` aceita um único argumento `Color` e configura o fundo do componente com essa cor.

```
1   // Figura 7.25: DrawRainbow.java
2   // Demonstra a utilização de cores em um array.
3   import java.awt.Color;
4   import java.awt.Graphics;
5   import javax.swing.JPanel;
6
7   public class DrawRainbow extends JPanel
8   {
9      // define as cores índigo e violeta
10     private final static Color VIOLET = new Color( 128, 0, 128 );
11     private final static Color INDIGO = new Color( 75, 0, 130 );
12
13     // a utilizar no arco-íris, iniciando da parte mais interna
14     // As duas entradas em branco resultam em um arco vazio no centro
15     private Color[] colors =
16        { Color.WHITE, Color.WHITE, VIOLET, INDIGO, Color.BLUE,
17          Color.GREEN, Color.YELLOW, Color.ORANGE, Color.RED };
18
19     // construtor
20     public DrawRainbow()
21     {
22        setBackground( Color.WHITE ); // configura o fundo como branco
23     } // fim do construtor DrawRainbow
24
25     // desenha um arco-íris utilizando arcos concêntricos
```

```
26      public void paintComponent( Graphics g )
27      {
28         super.paintComponent( g );
29
30         int radius = 20; // raio de um arco
31
32         // desenha o arco-íris perto da parte central inferior
33         int centerX = getWidth() / 2;
34         int centerY = getHeight() - 10;
35
36         // desenha arcos preenchidos com o mais externo
37         for ( int counter = colors.length; counter > 0; counter-- )
38         {
39            // configura a cor para o arco atual
40            g.setColor( colors[ counter - 1 ] );
41
42            // preenche o arco de 0 a 180 graus
43            g.fillArc( centerX - counter * radius,
44               centerY - counter * radius,
45               counter * radius * 2, counter * radius * 2, 0, 180 );
46         } // fim de for
47      } // fim do método paintComponent
48   } // fim da classe DrawRainbow
```

Figura 7.25 | Desenhando um arco-íris com arcos e um array de cores.

A linha 30 em `paintComponent` declara a variável local `radius`, que determina o raio de cada arco. As variáveis locais `centerX` e `centerY` (linhas 33–34) determinam a localização do ponto intermediário na base do arco-íris. O loop nas linhas 37–46 utiliza a variável de controle `counter` para contar para trás a partir do fim do array, desenhando primeiro os arcos maiores e colocando cada arco menor sucessivo por cima do arco anterior. A linha 40 configura a cor para desenhar o arco atual do array. A razão de termos entradas `Color.WHITE` no começo do array é criar o arco vazio no centro. Caso contrário, o centro do arco-íris seria apenas um semicírculo roxo sólido. [*Nota:* você pode alterar as cores individuais e o número de entradas no array para criar novos desenhos].

A chamada de método **`fillArc`** nas linhas 43–45 desenha um semicírculo preenchido. O método `fillArc` requer seis parâmetros. Os quatro primeiros parâmetros representam o retângulo delimitador em que o arco será desenhado. Os dois primeiros deles especificam as coordenadas do canto superior esquerdo do retângulo delimitador e os dois seguintes especificam sua largura e altura. O quinto parâmetro é o ângulo inicial na oval e o sexto especifica a **varredura**, ou a quantidade de arco a cobrir. O ângulo inicial e a varredura são medidos em graus, com zero grau apontando para a direita. Uma varredura positiva desenha o arco anti-horário, enquanto uma negativa desenha o arco no sentido horário. Um método semelhante a `fillArc` é **`drawArc`** — ele requer os mesmos parâmetros que `fillArc`, mas desenha a borda do arco em vez de preenchê-lo.

A classe `DrawRainbowTest` (Figura 7.26) cria e configura um `JFrame` para exibir o arco-íris. Uma vez que o programa torna o `JFrame` visível, o sistema chama o método `paintComponent` na classe `DrawRainbow` para desenhar o arco-íris na tela.

```
1   // Figura 7.26: DrawRainbowTest.java
2   // Aplicativo de teste para exibir um arco-íris.
3   import javax.swing.JFrame;
4
5   public class DrawRainbowTest
6   {
7      public static void main( String[] args )
8      {
9         DrawRainbow panel = new DrawRainbow();
10        JFrame application = new JFrame();
11
12        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13        application.add( panel );
14        application.setSize( 400, 250 );
15        application.setVisible( true );
16     } // fim de main
17  } // fim da classe DrawRainbowTest
```

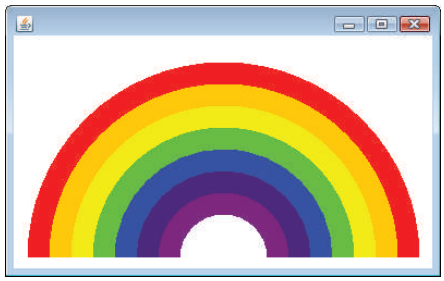


Figura 7.26 | Criando `JFrame` para exibir um arco-íris.

Exercício do Estudo de caso GUI e imagens gráficas

7.1 (***Desenhando espirais***) Neste exercício, você desenhará espirais com os métodos `drawLine` e `drawArc`.

a) Desenhe uma espiral com a forma quadrada (como na captura de tela esquerda da Figura 7.27), centralizado no painel, utilizando o método `drawLine`. Uma técnica é utilizar um loop que aumenta o comprimento da linha depois de desenhar cada duas linhas. A direção na qual desenhar a próxima linha deve seguir um padrão distinto, por exemplo, para baixo, para a esquerda, para cima, para a direita.

b) Desenhe uma espiral circular (como na captura de tela à direita da Figura 7.27), utilizando o método `drawArc` para desenhar um semicírculo por vez. Cada semicírculo sucessivo deve ter um raio maior (conforme especificado pela largura do retângulo delimitador) e deve continuar a desenhar onde o semicírculo anterior concluir.

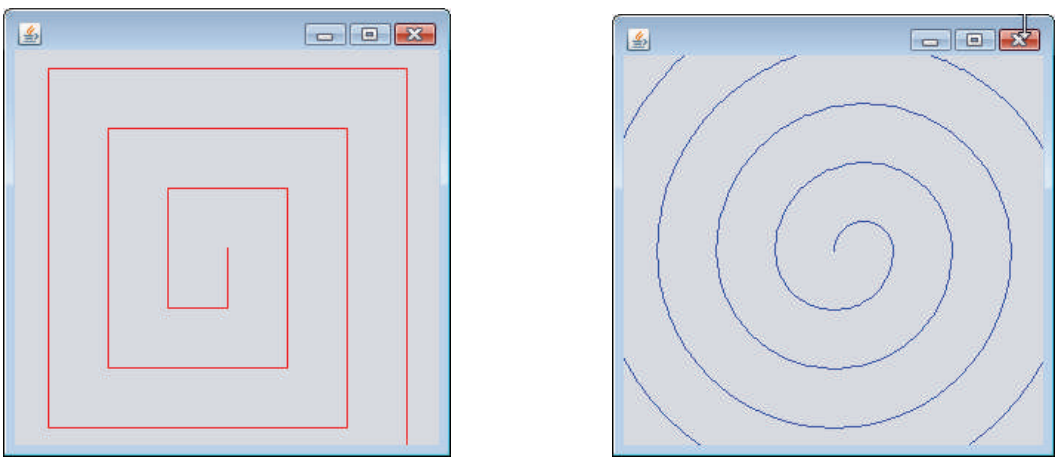


Figura 7.27 | Desenhando uma espiral com `drawLine` (esquerda) e `drawArc` (direita).

7.16 Conclusão

Este capítulo iniciou nossa introdução às estruturas de dados, explorando o uso de arrays para armazenar e recuperar dados de listas e tabelas de valores. Os exemplos do capítulo demonstraram como declarar e inicializar um array e referenciar elementos individuais de um array. O capítulo introduziu a instrução `for` aprimorada para iterar por arrays. Também ilustramos como passar arrays para métodos e declarar e manipular arrays multidimensionais. Por fim, o capítulo mostrou como escrever os métodos que utilizam a lista de argumentos de comprimento variável e como ler argumentos passados para um programa a partir da linha de comando.

Introduzimos a coleção genérica `ArrayList<T>`, que fornece todas as funcionalidades e desempenho dos arrays, junto com outras capacidades úteis, como redimensionamento dinâmico. Utilizamos métodos `add` para adicionar novos itens ao fim de uma `ArrayList` e inserir itens em uma `ArrayList`. O método `remove` foi utilizado para remover a primeira ocorrência de um item especificado, e uma versão sobrecarregada de `remove` foi utilizada para remover um item em um índice especificado. Utilizamos o método `size` para obter o número de itens em `ArrayList`.

Continuaremos nossa cobertura das estruturas de dados no Capítulo 20, "Coleções genéricas". O Capítulo 20 introduz o Java Collections Framework, que utiliza genéricos para permitir aos programadores especificar os tipos de objetos exatos que uma estrutura de dados particular armazenará. O Capítulo 20 também introduz outras estruturas de dados predefinidas do Java. A Collections API fornece a classe `Arrays`-, que contém métodos utilitários para manipulação de array. O Capítulo 20 utiliza vários métodos `static` da classe `Arrays` para realizar essas manipulações, como classificar e pesquisar os dados de um array. Você será capaz de utilizar alguns dos métodos `Arrays` discutidos no

Capítulo 20 depois de ler o capítulo atual, mas alguns dos métodos `Arrays` requerem conhecimento de conceitos apresentados mais adiante no livro. O Capítulo 21 apresenta o tópico dos genéricos, que fornecem o meio para criar modelos gerais de métodos e classes que podem ser declaradas uma vez, mas utilizados com muitos tipos de dados diferentes. O Capítulo 22 mostra como construir estruturas de dados dinâmicas, como listas, filas, pilhas e árvores, cujo tamanho pode aumentar e diminuir à medida que os programas executam.

Neste capítulo, apresentamos os conceitos básicos de classes, objetos, instruções de controle, métodos, arrays e coleções. No próximo capítulo, faremos um exame mais aprofundado de classes e objetos.

Resumo

Seção 7.1 Introdução

- **Arrays** são estruturas de dados consistindo em itens de dados do mesmo tipo relacionados. Os arrays são entidades de largura fixa — eles mantêm o mesmo comprimento uma vez que são criados, embora a referência de um comprimento diferente possa ser novamente atribuída à variável de array.

Seção 7.2 Arrays

- Um array é um grupo de variáveis (chamados elementos ou componentes) que contém valores todos do mesmo tipo. Os arrays são objetos, portanto, são considerados tipos por referência.
- Um programa referencia qualquer um dos elementos de um array com uma expressão de acesso ao array que inclui o nome do array seguido pelo índice do elemento particular em colchetes (`[]`).
- O primeiro elemento em cada array tem índice zero e às vezes é chamado de zero-ésimo elemento.
- Um índice deve ser um inteiro não negativo. Um programa pode utilizar uma expressão como um índice.
- Um objeto array conhece seu próprio comprimento e armazena essas informações em uma variável de instância `length`.

Seção 7.3 Declarando e criando arrays

- Para criar um objeto array, especifique o tipo de elemento e o número de elementos do array como parte de uma expressão de criação de arrays que utiliza a palavra-chave `new`.
- Quando um array é criado, cada elemento recebe um valor padrão — zero para elementos de tipo primitivo numéricos, `false` para elementos booleanos e `null` para referências.
- Em uma declaração de array, o tipo e os colchetes podem ser combinados no início da declaração para indicar que todos os identificadores na declaração são variáveis de array.
- Todo elemento de um array do tipo primitivo contém uma variável do tipo declarado do array. Cada elemento de um array de tipo por referência é uma referência a um objeto do tipo declarado do array.

Seção 7.4 Exemplos que utilizam arrays

- Um programa pode criar um array e inicializar seus elementos com um inicializador de array.
- As variáveis constantes são declaradas com a palavra-chave `final`, devem ser inicializadas antes de serem usadas e não podem ser modificadas depois.
- Quando um programa Java executa, a JVM verifica os índices de array para assegurar que eles são maiores ou iguais a 0 e menores que o tamanho do array. Se um programa utilizar um índice inválido, o Java gera uma suposta exceção para indicar a ocorrência de um erro no programa em tempo de execução.

Seção 7.5 Estudo de caso: simulação de embaralhamento e distribuição de cartas

- O método `toString` de um objeto é chamado implicitamente quando o objeto é utilizado onde uma `String` é esperada (por exemplo, quando `printf` gera saída do objeto como uma `String` utilizando o especificador de formato `%s` ou quando o objeto é concatenado para uma `String` utilizando o operador `+`).

Seção 7.6 A estrutura `for` aprimorada

- A instrução `for` aprimorada permite-lhe iterar pelos elementos de um array ou uma coleção sem utilizar um contador. A sintaxe de uma instrução `for` aprimorada é:

```
for ( parâmetro : nomeDoArray)
   instrução
```

  onde *parâmetro* tem um tipo e um identificador (por exemplo, `int number` `number`) e *nomeDoArray* é o array pelo qual iterar.
- A instrução `for` aprimorada não pode ser utilizada para modificar elementos em um array. Se um programa precisar modificar elementos, utilize a tradicional instrução `for` controlada por contador.

Seção 7.7 Passando arrays para métodos

- Quando um argumento é passado por valor, uma cópia do valor do argumento é feita e passada para o método chamador. O método chamado funciona exclusivamente com a cópia.
- Quando um argumento é passado por referência, o método chamado pode acessar o valor do argumento no chamador diretamente e, possivelmente, modificá-lo.
- O Java não permite aos programadores escolher entre passar por valor e passar por referência — todos os argumentos são passados por valor. Uma chamada de método pode passar dois tipos de valores para um método — cópias dos valores primitivos e cópias das referências a objetos. Embora uma referência do objeto seja passada por valor, um método ainda pode interagir com o objeto referenciado chamando seus métodos `public` que utilizem a cópia da referência do objeto.
- Para passar uma referência de objeto para um método, simplesmente especifique na chamada de método o nome da variável que referencia o objeto.
- Quando você passa um array ou um elemento de array individual de um tipo por referência para um método, o método chamado recebe uma cópia do array ou uma referência do elemento. Quando você passa um elemento individual de um tipo primitivo, o método chamado recebe uma cópia do valor do elemento.
- Para passar um elemento de array individual para um método, utilize o nome indexado do array como um argumento na chamada de método.

Seção 7.9 Arrays multidimensionais

- Os arrays multidimensionais com duas dimensões costumam ser utilizados para representar tabelas de valores consistindo em informações organizadas em linhas e colunas.
- Os arrays que requerem dois índices para identificar um elemento particular são chamados arrays bidimensionais. Um array com *m* linhas e *n* colunas é chamado de array *m* por *n*. Um array bidimensional pode ser inicializado com um inicializador de array da forma:

  tipoDoArray`[][]` *nomeDoArray* `= { {` *linha1 inicializador* `}, {` *linha2 inicializador* `}, ... };`
- Os arrays multidimensionais são mantidos como arrays de arrays unidimensionais separados. Como resultado, não é necessário que os comprimentos das linhas em um array bidimensional sejam os mesmos.
- Um array multidimensional com o mesmo número de colunas em cada linha pode ser criado com uma expressão de criação de array na forma:

  tipoDoArray`[][]` *nomeDoArray* `= new` *tipoDoArray*`[` *númeroDeLinhas* `][` *númeroDeColunas* `];`

Seção 7.11 Listas de argumentos de comprimento variável

- Um tipo de argumento seguido por reticências (...) na lista de parâmetros de um método indica que o método recebe um número variável de argumentos desse tipo particular. As reticências podem ocorrer apenas uma vez na lista de parâmetro de um método e devem estar no fim da lista de parâmetros.
- Uma lista de argumentos de comprimento variável é tratada como um array dentro do corpo do método. O número de argumentos no array pode ser obtido utilizando o campo `length` do array.

Seção 7.12 Utilizando argumentos de linha de comando

- A passagem de argumentos para `main` a partir da linha de comando é alcançada incluindo um parâmetro de tipo `String[]` na lista de parâmetros de `main`. Por convenção, o parâmetro de `main` é chamado `args`.
- O Java passa os argumentos da linha de comando que aparecem depois do nome de classe no comando `java` para o método `main` do aplicativo como `String`s no array `args`. O número de argumentos passados a partir da linha de comando é obtido acessando o atributo `length` do array.

Seção 7.13 Classe `Arrays`

- A classe `Arrays` fornece métodos `static` que desempenham manipulações de array comuns, incluindo `sort` para classificar um array, `binarySearch` para pesquisar um array classificado, `equals` para comparar arrays e `fill` para inserir itens em um array.
- O método `arraycopy` da classe `System` permite copiar os elementos de um array para outro.

Seção 7.14 Introdução a coleções e classe `ArrayList`

- As classes de coleção da API do Java fornecem métodos eficientes que organizam, armazenam e recuperam dados sem exigir que você conheça como os dados são armazenados.
- Uma `ArrayList<T>` é semelhante a um array, mas pode ser dinamicamente redimensionada.
- O método `add` com um argumento acrescenta um elemento ao fim de um `ArrayList`.
- O método `add` com dois argumentos insere um novo elemento em uma posição especificada em uma `ArrayList`.
- O método `size` retorna o número de elementos atualmente em uma `ArrayList`.
- O método `remove` com uma referência a um objeto como um argumento remove o primeiro elemento que corresponde ao valor do argumento.
- O método `remove` com um argumento de número inteiro remove o elemento no índice especificado e todos os elementos acima desse índice são deslocados para baixo por um.
- O método `contains` retorna `true` se o elemento for localizado em `ArrayList`, e `false` caso contrário.

Terminologia

0, flag, 196
array, 190
array bidimensional, 209
ArrayList<T>, classe genérica, 221
Arrays, classe, 219
chamada por referência, 205
chamada por valor, 205
classe genérica, 221
colchetes, [], 190
coleção, 221
componente de um array, 190
constante nomeada, 195
contains, método da classe ArrayList<T>, 222
drawArc, método da classe Graphics, 224
elemento de um array, 190
escalar, 203
estrutura de dados, 190
expressão de acesso ao array, 190
expressão de criação de array, 191
fillArc, método da classe Graphics, 224
for, instrução de repetição aprimorada, 202
índice (subscript), 190
índice zero, 190
inicializador de array, 193
length, variável de instância de um array, 191
linha de comando, argumento, 218
lista de argumento de tamanho variável, 217
lista inicializadora, 193
m por *n*, array, 209
nome de um array, 191
passar por referência, 205
passar por valor, 205
percorrer um array, 211
redimensionamento dinâmico, 190
remove, método da classe ArrayList<T>, 222
reticências (...) em uma lista de parâmetros de método, 217
sort, método da classe Arrays, 219
subscrito (índice), 190
varredura, 224
verificação de limites, 199
zero-ésimo elemento, 190

Exercícios de autorrevisão

7.1 Preencha a(s) lacuna(s) em cada uma das seguintes instruções:

a) Listas e tabelas de valores podem ser armazenadas em _____.
b) Um array é um grupo de _____ (chamadas elementos ou componentes) contendo valores que contêm todos o mesmo _____.
c) O _____ permite aos programadores iterar pelos elementos em um array sem utilizar um contador.
d) O número utilizado para referenciar um elemento particular de array é chamado _____ do elemento.
e) Um array que utiliza dois índices é referido como um array _____.
f) Utilize a instrução `for` aprimorada _____ para percorrer array `double numbers`.
g) Argumentos de linha de comando são armazenados em _____.
h) Utilize a expressão _____ para receber o número total de argumentos em uma linha de comando. Suponha que os argumentos da linha de comando são armazenados em `String[] args`.
i) Dado o comando `java MyClass test`, o primeiro argumento de linha de comando é _____.
j) Um(a) _____ na lista de parâmetro de um método indica que o método pode receber um número variável de argumentos.

7.2 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Um array pode armazenar muitos tipos de valores diferentes.
b) Um índice de array deve ser normalmente do tipo `float`.
c) Um elemento individual de um array que é passado para um método e modificado nesse método conterá o valor modificado quando o método chamado completar sua execução.
d) Argumentos de linha de comando são separados por vírgulas.

7.3 Realize as seguintes tarefas para um array chamado `fractions`:

a) Declare uma constante `ARRAY_SIZE` que é inicializada como 10.
b) Declare um array com elementos `ARRAY_SIZE` do tipo `double`, e inicialize os elementos em `0`.
c) Referencie o elemento 4 do array.
d) Atribua o valor `1.667` ao elemento 9 do array.
e) Atribua o valor `3.333` ao elemento 6 do array.
f) Some todos os elementos do array, utilizando uma instrução `for`. Declare a variável inteira `x` como uma variável de controle para o loop.

7.4 Realize as seguintes tarefas para um array chamado `table`:

a) Declare e crie o array como um array de inteiros que tem três linhas e três colunas. Assuma que a constante `ARRAY_SIZE` foi declarada como `3`.
b) Quantos elementos o array contém?
c) Utilize uma instrução `for` para inicializar cada elemento do array com a soma de seus índices. Assuma que as variáveis inteiras `x` e `y` são declaradas como variáveis de controle.

7.5 Localize e corrija o erro em cada um dos seguintes segmentos de programa:

a)
```
final int ARRAY_SIZE = 5;
ARRAY_SIZE = 10;
```

b) Suponha `int [] b = new int [10];`
```
for ( int i = 0; i <= b.length; i++ )
   b[ i ] = 1;
```

c) Suponha `int [][] a = {{1, 2}, {3, 4}};`
```
   a[ 1, 1 ] = 5;
```

Respostas dos exercícios de autorrevisão

7.1 a) arrays. b) variáveis, tipo. c) instrução `for` aprimorada. d) índice (ou subscrito ou número de posição). e) bidimensional. f) `for ( double d : numbers )`. g) um array de `String`s, chamado `args` por convenção. h) `args.length`. i) `test`. j) reticências (...).

7.2 a) Falso. Um array pode armazenar apenas valores do mesmo tipo.

b) Falso. Um índice de array deve ser um inteiro ou uma expressão do tipo inteiro.

c) Para elementos individuais do tipo primitivo de um array: Falso. Um método chamado recebe e manipula uma cópia do valor desse elemento, então as modificações não afetam o valor original. Mas se a referência de um array for passada para um método, as modificações nos elementos do array feitas no método chamado são de fato refletidas no original. Para elementos individuais de um tipo por referência: Verdadeiro. Um método chamado recebe uma cópia da referência desse elemento e as mudanças no objeto referenciado serão refletidas no elemento do array original.

d) Falso. Os argumentos de linha de comando são separados por um espaço em branco.

7.3 a)
```
final int ARRAY_SIZE = 10;
```
b)
```
double[] fractions = new double[ ARRAY_SIZE ];
```
c)
```
fractions[ 4 ]
```
d)
```
fractions[ 9 ] = 1.667;
```
e)
```
fractions[ 6 ] = 3.333;
```
f)
```
double total = 0.0;
for ( int x = 0; x < fractions.length; x++ )
   total += fractions[ x ];
```

7.4 a)
```
int[][] table = new int[ ARRAY_SIZE ][ ARRAY_SIZE ];
```
b) Nove.

c)
```
for ( int x = 0; x < table.length; x++ )
      for ( int y = 0; y < table[ x ].length; y++ )
         table[ x ][ y ] = x + y;
```

7.5 a) Erro: atribuir um valor a uma constante depois de ela ter sido inicializada.
Correção: atribua o valor correto à constante em um `final int ARRAY_SIZE` declaração ou declara outra variável.

b) Erro: referenciar um elemento de array além dos limites do array (`b[10]`).
Correção: atere o operador <= para <.

c) Erro: a indexação do array é realizada incorretamente.
Correção: altere a instrução para `a[ 1 ][ 1 ] = 5;`.

Exercícios

7.6 Preencha as lacunas em cada uma das seguintes afirmações:

a) O array unidimensional `p` contém quatro elementos. Os nomes desses elementos são _____, _____, _____ e _____.

b) Nomear um array, declarar seu tipo e especificar o número de dimensões no array é chamado de _____ array.

c) Em um array bidimensional, o primeiro índice identifica o _____ de um elemento e o segundo índice identifica o _____ de um elemento.

d) Um array *m* por *n* contém _____ linhas, _____ colunas e _____ elementos.

e) O nome do elemento na linha 3 e na coluna 5 do array `d` é _____.

7.7 Determine se cada um dos seguintes é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Para referir-se a uma localização particular ou elemento dentro de um array, especificamos o nome do array e o valor do elemento particular.

b) Uma declaração de array reserva espaço para o array.

c) Para indicar que 100 localizações devem ser reservadas para o array de inteiros `p`, o programador escreve a declaração
```
p[ 100 ];
```

d) Um aplicativo que inicializa os elementos de um array de 15 elementos como zero deve conter pelo menos uma instrução `for`.

e) Um aplicativo que soma os elementos de um array bidimensional deve conter instruções `for` aninhadas.

7.8 Escreva instruções Java para realizar cada uma das seguintes tarefas:

a) Exiba o valor do elemento 6 do array `f`.

b) Inicialize cada um dos cinco elementos de array de inteiros unidimensional `g` como `8`.

c) Some os 100 elementos do array de ponto flutuante `c`.

d) Copie o array `a` de 11 elementos para a primeira parte de array `b`, que contém 34 elementos.

e) Determine e exiba os maiores e menores valores contidos no array de ponto flutuante `w` de 9 elementos.

7.9 Considere um array de inteiros dois por três `t`.

a) Escreva uma instrução que declara e cria `t`.

b) Quantas linhas tem `t`?

c) Quantas colunas tem `t`?

d) Quantos elementos tem `t`?

e) Escreva expressões de acesso para todos os elementos na linha 1 de `t`.

f) Escreva expressões de acesso para todos os elementos na coluna 2 de `t`.

g) Escreva uma única instrução que configura o elemento de `t` na linha 0 e na coluna 1 como zero.

h) Escreva instruções específicas para inicializar cada elemento de `t` em zero.

i) Escreva uma instrução `for` aninhada que inicializa cada elemento de `t` como zero.

j) Escreva uma instrução `for` aninhada que insere os valores para os elementos de `t` a partir do usuário.

k) Escreva uma série de instruções que determina e exibe o valor menor em `t`.

l) Escreva uma única instrução `printf` que exibe os elementos da primeira linha de `t`.

m) Escreva uma instrução que soma os elementos da terceira coluna de `t`. Não utilize repetição.

n) Escreva uma série de instruções que exibe o conteúdo de `t` no formato tabular. Liste os índices de coluna como títulos na parte superior e liste os índices de linha à esquerda de cada linha.

7.10 **(*Comissões de vendas*)** Utilize um array unidimensional para resolver o seguinte problema: uma empresa paga seu pessoal de vendas por comissão. O pessoal de vendas recebe R$ 200 por semana mais 9% de suas vendas brutas durante essa semana. Por exemplo, um vendedor que vende R$ 5.000 brutos em uma semana recebe R$ 200 mais 9% de R$ 5.000 ou um total de R$ 650. Escreva um aplicativo (utilizando um array de contadores) que determina quanto o pessoal de vendas ganhou em cada um dos seguintes intervalos (assuma que o salário de cada vendedor foi truncado para uma quantia inteira):

a) $200–299

b) $300–399

c) $400–499

d) $500–599

e) $600–699

f) $700–799

g) $800–899

h) $900–999

i) R$ 1.000 e acima

Resuma os resultados em formato tabular.

7.11 Escreva instruções que realizam as seguintes operações de um array unidimensional:

a) Configure os 10 elementos do array de inteiros `counts` como zeros.

b) Adicione um a cada um dos 15 elementos do array de inteiros `bonus`.

c) Exiba os cinco valores de array de inteiros `bestScores` em formato de coluna.

7.12 **(*Eliminação de duplicatas*)** Utilize um array unidimensional para resolver o seguinte problema: Escreva um aplicativo que insere cinco números, cada um entre 10 e 100, inclusive. À medida que cada número é lido, só o exiba se ele não for uma duplicata de um número já lido. Cuide de tratar o "pior caso", em que todos os cinco números são diferentes. Utilize o menor array possível para resolver esse problema. Exiba o conjunto completo de valores únicos inseridos depois que o usuário inserir cada valor novo.

7.13 Rotule os elementos do array bidimensional três por cinco `sales` para indicar a ordem em que eles são configurados como zero pelo seguinte segmento de programa:

```
for ( int row = 0; row < sales.length; row++ )
{
   for ( int col = 0; col < sales[ row ].length; col++ )
   {
      sales[ row ][ col ] = 0;
   }
}
```

7.14 Escreva um aplicativo que calcula o produto de uma série de inteiros que são passados para método `product` utilizando uma lista de argumentos de comprimento variável. Teste seu método com várias chamadas, cada uma com um número diferente de argumentos.

7.15 Reescreva a Figura 7.2 para que o tamanho do array seja especificado pelo primeiro argumento de linha de comando. Se nenhum argumento de linha de comando for fornecido, utilize 10 como o tamanho padrão do array.

7.16 Escreva um aplicativo que utiliza uma instrução `for` aprimorada para somar os valores `double` passados pelos argumentos de linha de comando. [*Dica:* Utilize o método `static` `parseDouble` da classe `Double` para converter uma `String` em um valor `double`.]

7.17 **(*Jogo de dados*)** Escreva um aplicativo para simular o lançamento de dois dados. O aplicativo deve utilizar um objeto de classe `Random` uma vez para lançar o primeiro dado e novamente para lançar o segundo dado. A soma dos dois valores deve então ser calculada. Cada dado pode mostrar um valor inteiro de 1 a 6, portanto a soma dos valores irá variar de 2 a 12, com 7 sendo a soma mais frequente e 2 e 12, as somas menos frequentes.

A Figura 7.28 mostra as 36 possíveis combinações de dois dados. Seu aplicativo deve lançar o dado 36.000 vezes. Utilize um array unidimensional para contar o número de vezes que cada possível soma aparece. Exiba os resultados em formato tabular. Determine se os totais são razoáveis (por exemplo, há seis maneiras de lançar um 7, então aproximadamente um sexto de todos os lançamentos deve ser 7).

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 7.28 | As 36 possíveis somas de dois dados.

7.18 **(*Jogo de dados Craps*)** Escreva um aplicativo que executa 1.000 jogos de dados (Figura 6.9) e responda às seguintes perguntas:

a) Quantos jogos são ganhos na primeira rolagem, segunda rolagem, ..., vigésima rolagem e depois da vigésima rolagem?

b) Quantos jogos são perdidos na primeira rolagem, segunda rolagem, ..., vigésima rolagem e depois da vigésima rolagem?

c) Quais são as chances de ganhar no jogo de dados? [*Nota*: Você deve descobrir que o *craps* é um dos jogos mais comuns de cassino. O que você supõe que isso significa?]

d) Qual é o comprimento médio de um jogo de dados *craps*?

e) As chances de ganhar se aumentam com o comprimento do jogo?

7.19 **(*Sistema de reservas de passagens áreas*)** Uma pequena companhia aérea acabou de comprar um computador para seu novo sistema automatizado de reservas. Você foi solicitado a desenvolver o novo sistema. Você escreverá um aplicativo para atribuir assentos em cada voo da companhia aérea (capacidade: 10 assentos).

Seu aplicativo deve exibir as seguintes alternativas: digite `1` para `First Class` e `2` para `Economy`. Se o usuário digitar `1`, seu aplicativo deve atribuir assentos na primeira classe (poltronas 1–5). Se o usuário digitar `2`, seu aplicativo deve atribuir um assento na classe econômica (poltronas 6–10). Seu aplicativo deve exibir um cartão de embarque indicando o número da poltrona da pessoa e se ela está na primeira classe ou na classe econômica.

Utilize um array unidimensional do tipo primitivo `boolean` para representar o gráfico de assentos do avião. Inicialize todos os elementos do array como `false` para indicar que todas as poltronas estão desocupadas. À medida que cada assento é atribuído, configure o elemento correspondente do array como `true` para indicar que o assento não está mais disponível.

Seu aplicativo nunca deve atribuir uma poltrona que já foi reservada. Quando a classe econômica estiver lotada, seu aplicativo deve perguntar à pessoa se ela aceita ficar na primeira classe (e vice-versa). Se for, faça a atribuição apropriada de assento. Se não for, exiba a mensagem `"Next flight leaves in 3 hours" [O próximo voo parte em 3 horas].`

7.20 **(*Vendas totais*)** Utilize um array bidimensional para resolver o seguinte problema: Uma empresa tem quatro equipes de vendas (1 a 4) que vendem cinco produtos diferentes (1–5). Uma vez por dia, cada vendedor passa uma nota de cada tipo de produto diferente vendido. Cada nota contém o seguinte:

a) O número do vendedor

b) O número do produto

c) O valor total em reais desse produto vendido nesse dia

Portanto, cada vendedor passa entre 0 e 5 notas de vendas por dia. Assuma que as informações a partir de todas as notas durante o mês último estão disponíveis. Escreva um aplicativo que leia todas essas informações sobre as vendas do último mês e resuma as vendas totais por vendedor e por produto. Todos os totais devem ser armazenados no array bidimensional `sales`. Depois de processar todas as informações do último mês, exiba os resultados em formato tabular, com cada coluna representando um vendedor particular e cada linha representando um produto particular. Some cada linha para obter o total das vendas de cada produto no último mês. Some cada coluna para obter o total de vendas por vendedor no último mês. Sua saída tabular deve incluir esses totais cruzados à direita das linhas totalizadas e na parte inferior das colunas totalizadas.

7.21 **(*Gráficos de tartaruga*)** A linguagem Logo tornou famoso o conceito de *gráficos de tartaruga*. Imagine uma tartaruga mecânica que caminha no lugar sob o controle de um aplicativo Java. A tartaruga segura uma caneta em uma de duas posições, para cima ou para baixo. Enquanto a caneta está para baixo, a tartaruga desenha formas à medida que se move, e enquanto a caneta está para cima, a tartaruga move-se quase livremente sem escrever nada. Nesse problema, você simulará a operação da tartaruga e criará um bloco de rascunho computadorizado.

Utilize um array de 20 por 20 `floor` que é inicializado como zeros. Leia comandos a partir de um array que contenha esses comandos. Monitore a posição atual da tartaruga todas as vezes e se a caneta está atualmente para cima ou para baixo. Assuma que a tartaruga sempre inicia na posição (0, 0) do chão com sua caneta para cima. O conjunto de comandos de tartaruga que seu aplicativo deve processar é mostrado na Figura 7.29.

Comando	Significado
1	Caneta para cima
2	Caneta para baixo
3	Vira para a direita
4	Vira para a esquerda
5,10	Avance 10 espaços (substitua 10 por um número diferente de espaços)
6	Exiba o array 20 por 20
9	Fim dos dados (sentinela)

Figura 7.29 | Comandos dos gráficos de tartaruga.

Suponha que a tartaruga esteja em algum lugar próximo ao centro do chão. O seguinte "programa" desenharia e exibiria um quadrado de 12 por 12 deixando a caneta na posição levantada:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

À medida que a tartaruga se move com a caneta por baixo, configure os elementos apropriados do array `floor` como `1`s. Quando o comando `6` (exibir o array) for dado, onde quer que haja um `1` no array, exiba um asterisco ou o caractere que você escolher. Onde quer que haja um `0`, exiba um espaço em branco.

Escreva um aplicativo para implementar as capacidades dos gráficos de tartaruga discutidas aqui. Escreva vários programas de gráfico de tartaruga para desenhar formas interessantes. Adicione outros comandos para aumentar as capacidades de sua linguagem de gráfico de tartaruga.

7.22 (***Passeio do Cavalo***) Um problema interessante para os fãs de xadrez é o problema do *Passeio do cavalo*, originalmente proposto pelo matemático Euler. A peça do cavalo pode mover-se em um tabuleiro vazio e tocar cada um dos 64 quadrados somente uma única vez. Aqui, estudamos esse intrigante problema em profundidade.

O cavalo só faz movimentos em forma de L (dois espaços em uma direção e um outro em uma direção perpendicular). Portanto, conforme a Figura 7.30, partindo de um quadrado próximo do centro de um tabuleiro de xadrez vazio, o cavalo (rotulado K) pode fazer oito movimentos diferentes (numerados de 0 a 7).

a) Desenhe um tabuleiro de xadrez oito por oito em uma folha de papel e tente o Passeio do Cavalo manualmente. Coloque um `1` no quadrado inicial, um `2` no segundo quadrado, um `3` no terceiro e assim por diante. Antes de iniciar o passeio, estime até onde você chegará, lembrando-se de que um passeio completo consiste em 64 movimentos. Até onde você foi? Isso foi próximo de sua estimativa?

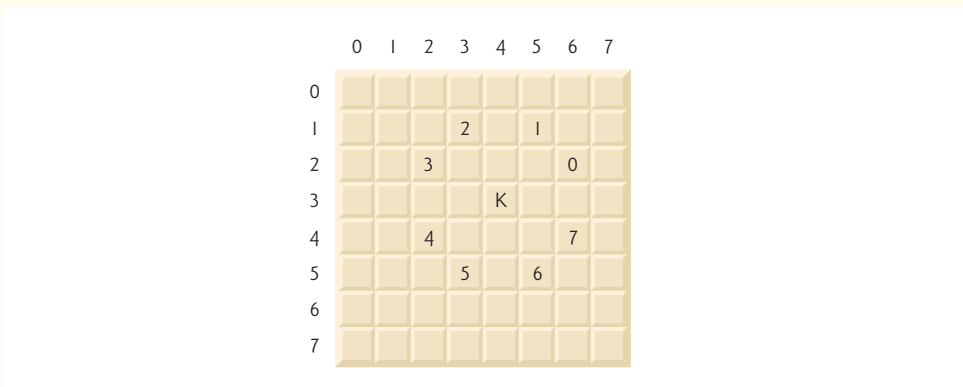


Figura 7.30 | Os oito possíveis movimento do cavalo.

b) Agora vamos desenvolver um aplicativo que moverá o cavalo pelo tabuleiro. O tabuleiro é representado por um array bidimensional oito por oito `board`. Cada quadrado é inicializado como zero. Descrevemos cada um dos oito possíveis movimentos em termos de seus componentes vertical e horizontal. Por exemplo, um movimento do tipo 0 como mostrado na Figura 7.30 consiste em mover dois quadrados horizontalmente

para direita e um quadrado verticalmente para cima. Um movimento do tipo 2 consiste em mover um quadrado horizontalmente para a esquerda e dois quadrados verticalmente para cima. Movimentos horizontais para a esquerda e movimentos verticais para cima são indicados com números negativos. Os oitos movimentos podem ser descritos por dois arrays unidimensionais, `horizontal` e `vertical`, como segue:

```
horizontal[ 0 ] = 2       vertical[ 0 ] = -1
horizontal[ 1 ] = 1       vertical[ 1 ] = -2
horizontal[ 2 ] = -1      vertical[ 2 ] = -2
horizontal[ 3 ] = -2      vertical[ 3 ] = -1
horizontal[ 4 ] = -2      vertical[ 4 ] = 1
horizontal[ 5 ] = -1      vertical[ 5 ] = 2
horizontal[ 6 ] = 1       vertical[ 6 ] = 2
horizontal[ 7 ] = 2       vertical[ 7 ] = 1
```

Faça com que as variáveis `currentRow` e `currentColumn` indiquem, respectivamente, a linha e a coluna da posição atual do cavalo. Para fazer um movimento do tipo `moveNumber`, em que `moveNumber` está entre 0 e 7, seu aplicativo deve utilizar as instruções:

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Escreva um aplicativo para mover o cavalo pelo tabuleiro. Mantenha um contador que varia de `1` a `64`. Registre a última contagem em cada quadrado para que o cavalo se mova. Teste cada movimento potencial para ver se o cavalo já visitou esse quadrado. Teste cada movimento potencial para assegurar que o cavalo não saia fora do tabuleiro. Execute o aplicativo. Quantos movimentos o cavalo fez?

c) Depois de tentar escrever e executar um aplicativo para o Passeio do Cavalo, você provavelmente desenvolveu algumas ideias valiosas. Utilizaremos essas ideias para desenvolver uma *heurística* (isto é, uma regra de senso comum) para mover o cavalo. A heurística não garante sucesso, mas uma heurística cuidadosamente desenvolvida aprimora significativamente a chance de sucesso. Você pode ter observado que os quadrados externos são mais incômodos que os quadrados próximos do centro do tabuleiro. De fato, os quadrados mais problemáticos ou inacessíveis são os quatro cantos.

A intuição pode sugerir que você deva tentar mover o cavalo para os quadrados mais problemáticos primeiro e deixar abertos aqueles que são fáceis de alcançar de modo que quando o tabuleiro ficar congestionado próximo do fim do passeio haja uma maior chance de sucesso.

Poderíamos desenvolver uma "acessibilidade heurística" classificando cada um dos quadrados de acordo com seu grau de acessibilidade e sempre movendo os cavalos (utilizando os movimentos em forma de L do cavalo) para o quadrado mais inacessível. Rotulamos um array bidimensional `accessibility` com números indicando a partir de quantos quadrados cada quadrado particular é acessível. Em um tabuleiro vazio, cada um dos 16 quadrados mais próximos do centro é avaliado como `8`, o quadrado de cada canto é avaliado como `2` e os outros quadrados têm números de acessibilidade de `3`, `4` ou `6` como segue:

```
2  3  4  4  4  4  3  2
3  4  6  6  6  6  4  3
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
3  4  6  6  6  6  4  3
2  3  4  4  4  4  3  2
```

Escreva uma versão do Passeio do Cavalo utilizando a heurística de acessibilidade. O cavalo sempre deve se mover para o quadrado com o número de acessibilidade mais baixo. Em caso de um impasse, o cavalo pode mover-se para qualquer quadrado já visitado. Portanto, o passeio pode iniciar em qualquer um dos quatro cantos. [*Nota:* à medida que o cavalo se move pelo tabuleiro de xadrez, seu aplicativo deve reduzir os números de acessibilidade a medida que mais quadrados tornam-se ocupados. Dessa maneira, em qualquer dado tempo durante o passeio, o número de acessibilidade de cada de quadrado disponível permanecerá precisamente igual ao número de quadrados a partir dos quais esse quadrado pode ser alcançado.] Execute essa versão do aplicativo. Você obteve um passeio completo? Modifique o aplicativo para executar 64 passeios, iniciando a partir de cada quadrado do tabuleiro de xadrez. Quantos tours completos você obteve?

d) Escreva uma versão do aplicativo Passeio do Cavalo que, diante de um impasse entre dois ou mais quadrados, decide qual quadrado escolher vislumbrando os quadrados alcançáveis a partir dos quadrados geradores do impasse. Seu aplicativo deve mover para o quadrado empatado para o qual o próximo movimento chegaria a um quadrado com o número de acessibilidade mais baixo.

7.23 **(*Passeio do Cavalo: abordagens de força bruta*)** Na parte (c) do Exercício 7.22, desenvolvemos uma solução para o problema do Passeio do Cavalo. A abordagem utilizada, chamada "acessibilidade heurística", gera muitas soluções e executa eficientemente.

À medida que os computadores continuam crescendo em potência, seremos capazes de resolver cada vez mais problemas com a pura capacidade do computador e algoritmos relativamente simples. Vamos chamar essa abordagem de solução de problemas de abordagem de "força bruta".

a) Utilize geração de números aleatórios para permitir ao cavalo andar no tabuleiro de xadrez (em seus movimentos válidos em forma de L) de maneira aleatória. Seu aplicativo deve executar um passeio e exibir o tabuleiro final. Até onde o cavalo chegou?

b) Muito provavelmente, o aplicativo na parte (a) produziu um passeio relativamente curto. Agora modifique seu aplicativo para tentar 1.000 passeios. Utilize um array unidimensional para monitorar o número de passeios de cada comprimento. Quando seu aplicativo terminar de tentar os 1.000 passeios, ele deve exibir organizadamente essas informações em formato tabular. Qual foi o melhor resultado?

c) Muito provavelmente, o aplicativo na parte (b) forneceu alguns passeios "respeitáveis", mas nenhum passeio completo. Agora deixe seu aplicativo executar até que produza um passeio completo. [*Atenção:* essa versão do aplicativo poderia ser executada durante horas em um computador poderoso.] Mais uma vez, mantenha uma tabela do número de tours de cada comprimento e exiba essa tabela quando o primeiro passeio completo for localizado. Quantos percursos seu aplicativo tenta antes de produzir um passeio completo? Quanto tempo que ele levou?

d) Compare a versão de força bruta do Passeio do Cavalo com a versão de acessibilidade heurística. Qual exigiu um estudo mais cuidadoso do problema? Qual algoritmo foi mais difícil de desenvolver? Qual exigiu mais capacidade do computador? Poderíamos ter certeza (com antecedência) de obter um passeio completo com a abordagem de acessibilidade heurística? Poderíamos ter certeza (com antecedência) de obter um passeio completo com a abordagem de força bruta? Argumente as vantagens e desvantagens de resolver problema de força bruta em geral.

7.24 (***Oito Rainhas***) Outro problema difícil para fãs de xadrez é o problema das Oitos Rainhas, que pede o seguinte: é possível colocar oito rainhas em um tabuleiro de xadrez vazio de modo que nenhuma rainha esteja "atacando" qualquer outra (isto é, sem que duas rainhas estejam na mesma linha, na mesma coluna ou na mesma diagonal)? Utilize a consideração desenvolvida no Exercício 7.22 a fim de formular uma heurística para resolver o problema das Oito Rainhas. Execute seu aplicativo. [*Dica:* é possível atribuir um valor para cada quadrado do tabuleiro de xadrez para indicar quantos quadrados de um tabuleiro de xadrez vazio "são eliminados" se uma rainha for colocada nesse quadrado. A cada um dos cantos seria atribuído o valor 22, como demonstrado pela Figura 7.31. Depois que esses "números de eliminação" são inseridos nos 64 quadrados, uma heurística apropriada poderia ser: coloque a próxima rainha no quadrado com o menor número de eliminação. Por que essa estratégia é intuitivamente atraente?]

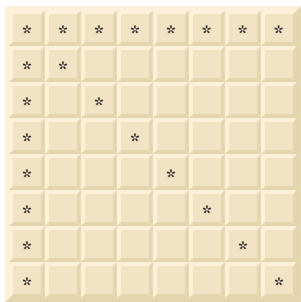


Figura 7.31 | Os 22 quadrados eliminados posicionando uma rainha no canto esquerdo superior.

7.25 (***Oito Rainhas: abordagens de força bruta***) Nesse exercício você desenvolverá várias abordagens de força bruta para resolver o problema das Oito de Rainhas introduzido no Exercício 7.24.

a) Utilize a técnica da força bruta aleatória desenvolvida no Exercício 7.23 para resolver o problema das Oitos Rainhas.

b) Utilize uma técnica exaustiva (isto é, tentar todas as combinações possíveis de oito rainhas no tabuleiro) para resolver esse problema.

c) Por que a abordagem de força bruta exaustiva poderia não ser apropriada para resolver o problema de Passeio do Cavalo?

d) Compare e contraste as abordagens de força bruta aleatória e da força bruta exaustiva.

7.26 (***Passeio do Cavalo: teste do passeio fechado***) No Passeio do Cavalo (Exercício 7.22), um passeio completo ocorre quando o cavalo move-se tocando cada um dos 64 quadrados do tabuleiro de xadrez somente uma única vez. Um passeio fechado ocorre quando o 64º movimento cai no quadrado em que o cavalo iniciou o passeio. Modifique o aplicativo escrito no Exercício 7.22 para testar o caso de um passeio fechado se um passeio completo tiver ocorrido.

7.27 (***Crivo de Eratóstenes***) Um número primo é qualquer inteiro maior que 1 que é igualmente divisível apenas por si mesmo e por 1. O crivo de Eratóstenes é um método de encontrar números primos. Ele opera como segue:

a) Crie um array `boolean` de tipo primitivo com todos os elementos inicializados como `true`. Os elementos do array com índices primos permanecerão `true`. Todos os outros elementos do array por fim são configurados como `false`.

b) Iniciando com o índice de array 2, determine se um dado elemento é `true`. Se for, faça um loop pelo restante do array e configure como `false` cada elemento cujo índice é um múltiplo do índice para o elemento com valor `true`. Então continue o processo com o próximo elemento com valor `true`. Para o índice de array 2, todos os elementos além do elemento 2 no array que tiverem índices que são múltiplos de 2 (índices 4, 6, 8, 10 etc.) serão configurados como `false`; para o índice de array 3, todos os elementos além do elemento 3 no array que tiverem índices que são múltiplos de 3 (índices 6, 9, 12, 15 etc.) serão configurados como `false`; e assim por diante.

Quando esse processo for concluído, os elementos de array que ainda forem `true` indicam que o índice é um número primo. Esses índices podem ser exibidos. Escreva um aplicativo que utiliza um array de 1.000 elementos para determinar e exibir os números primos entre 2 e 999. Ignore elementos de array 0 e 1.

7.28 (***Simulação: a tartaruga e a lebre***) Neste problema, você recriará a clássica corrida da tartaruga e da lebre. Você utilizará geração de números aleatórios para desenvolver uma simulação desse evento memorável.

Nossos competidores começam a corrida no quadrado 1 de 70 quadrados. Cada quadrado representa uma possível posição ao longo do percurso da competição. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar o quadrado 70 é recompensado com um cesto de cenouras frescas e alface. O percurso envolve subir uma montanha escorregadia, então ocasionalmente os competidores perdem terreno.

Um relógio emite um tique por segundo. A cada tique do relógio, seu aplicativo deve ajustar a posição dos animais de acordo com as regras na Figura 7.32. Utilize variáveis para monitorar as posições dos animais (isto é, os números de posição são 1-70). Inicie cada animal na posição 1 (a "largada"). Se um animal escorregar para a esquerda do quadrado 1, mova-o novamente para o quadrado 1.

Animal	Tipo de movimento	% do tempo	Movimento real
Tartaruga	Caminhada rápida	50%	3 quadrados à direita
	Escorregão	20%	6 quadrados à esquerda
	Caminhada lenta	30%	1 quadrado à direita
Lebre	Dormir	20%	Sem nenhum movimento
	Salto grande	20%	9 quadrados à direita
	Escorregão grande	10%	12 quadrados à esquerda
	Salto pequeno	30%	1 quadrado à direita
	Escorregão pequeno	20%	2 quadrados à esquerda

Figura 7.32 | Regras para ajustar as posições da tartaruga e da lebre.

Gere as porcentagens na Figura 7.32 produzindo um número inteiro aleatório i no intervalo $1 \leq i \leq 10$. Para a tartaruga, execute uma "passo rápido" quando $1 \leq i \leq 5$, um "tropeço" ou "escorregão" quando $6 \leq i \leq 7$ ou uma "passo lento" quando $8 \leq i \leq 10$. Use uma técnica parecida para mover a lebre.

Comece a corrida exibindo

```
BANG!!!!!
E LÁ VÃO ELES!!!!!
```

Então, para cada tique do relógio (isto é, para cada repetição de um loop), exiba uma linha de 70 posições mostrando a letra `T` na posição da tartaruga e a letra `H` na posição da lebre. Ocasionalmente, os competidores ocuparão o mesmo quadrado. Nesse caso, a tartaruga morde a lebre e seu aplicativo deve exibir `AI!!!` começando nessa posição. Todas as posições de saída além de `T`, `H` ou `OUCH!!!` (no caso de um empate) devem estar em branco.

Depois de cada linha ser exibida, teste se o animal alcançou ou passou o quadrado 70. Se tiver alcançado, exiba o vencedor e termine a simulação. Se a tartaruga ganhar, exiba `A TARTARUGA VENCEU!!! ÊH!!!` Se a lebre ganhar, exiba `A LEBRE GANHOU. OH.` Se os dois ganharem na mesma hora, você pode querer favorecer a tartaruga (a "coitadinha") ou pode querer exibir `OCORREU UM EMPATE`. Se nenhum animal ganhar, realize o loop novamente para simular o próximo tique do relógio. Quando você estiver pronto para executar seu aplicativo, monte um grupo de fãs para observar a corrida. Você se surpreenderá com a empolgação da sua audiência!

Mais adiante no livro, introduzimos várias capacidades do Java, como gráficos, imagens, animação, som e multithreading. À medida que estudar esses recursos, você pode se divertir aprimorando sua simulação da competição entre a lebre e a tartaruga.

7.29 **(*Série de Fibonacci*)** A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com os termos 0 e 1 e tem a propriedade de que cada termo sucessivo é a soma dos dois termos precedentes.

a) Escreva um método `fibonacci( n )` que calcula o n-ésimo número de Fibonacci. Incorpore esse método a um aplicativo que permita ao usuário inserir o valor de `n`.

b) Determine o maior número de Fibonacci que pode ser exibido em seu sistema.

c) Modifique o aplicativo que você escreveu na parte (a) para utilizar `double` em vez de `int` para calcular e retornar números de Fibonacci e utilizar esse aplicativo modificado para repetir a parte (b).

Os exercícios 7.30–7.33 são razoavelmente desafiadores. Depois de concluir esses exercícios, você deve ser capaz de implementar facilmente os jogos de cartas mais populares.

7.30 **(*Embaralhamento e distribuição*)** Modifique o aplicativo da Figura 7.11 para distribuir uma mão de cinco cartas de pôquer. Então modifique a classe `DeckOfCards` da Figura 7.10 para incluir métodos que determinam se uma mão contém:

a) um par

b) dois pares

c) trinca (por exemplo, três valetes)

d) quadra (por exemplo, quatro ases)

e) flush (isto é, cinco cartas do mesmo naipe)

f) straight (isto é, cinco cartas de valores consecutivos)

g) uma full house (isto é, duas cartas de um valor e três cartas de outro valor)

[*Dica:* Adicione os métodos `getFace` e `getSuit` à classe `Card` da Figura 7.9.]

7.31 **(*Embaralhamento e distribuição de carta*)** Utilize os métodos desenvolvidos no Exercício 7.30 para escrever um aplicativo que distribua duas mãos de pôquer de cinco cartas, avalia cada mão e determina qual é a melhor mão.

7.32 ***(Embaralhamento e distribuição de carta)*** Modifique o aplicativo desenvolvido no Exercício 7.31 para que ele possa simular o carteador. A mão de cinco cartas do carteador é distribuída "no escuro" para que o jogador não possa vê-la. O programa deve então avaliar a mão do carteador e, com base na qualidade da mão, o carteador deve distribuir uma, duas ou três mais cartas para substituir o número correspondente de cartas desnecessárias na mão original. O aplicativo então deve reavaliar a mão do carteador. [*Atenção:* esse é um problema difícil!]

7.33 (***Embaralhamento e distribuição de carta***) Modifique o aplicativo desenvolvido no Exercício 7.32 para que ele possa tratar a mão do carteador automaticamente, mas o jogador tenha permissão de decidir que cartas ele quer substituir. O aplicativo então deve avaliar ambas as mãos e determinar quem ganha. Agora utilize esse novo aplicativo para disputar 20 jogos contra o computador. Quem ganha mais jogos, você ou o computador? Peça para amigo jogar 20 jogos contra o computador. Quem ganha mais jogos? Com base nos resultados desses jogos, refine seu aplicativo de pôquer. (Esse, também, é um problema difícil.) Dispute mais 20 jogos. Seu aplicativo modificado joga melhor?

7.34 (***Embaralhando e distribuindo cartas***) Modifique o aplicativo das figuras 7.9–7.11 para usar as enumerações `Face` e `Suit` a fim de representar as faces e os naipes das cartas. Declare cada uma dessas enumerações como um tipo `public` em arquivo-fonte próprio. Cada `Card` deve ter uma variável de instância `Face` e `Suit`. Estas devem ser inicializadas pelo construtor `Card`. Na classe `DeckOfCards`, crie um array de `faces` que é inicializado com os nomes das constantes na enumeração `Face`, e um array `Suits` que é inicializado com os nomes das constantes na enumeração `Suit`. [*Nota:* ao gerar a saída de uma constante de enumeração como uma `String`, o nome da constante é exibido.]

Seção especial: construindo seu próprio computador

Nos vários problemas a seguir, paramos temporariamente nosso estudo sobre o mundo da programação de linguagem de alto nível para abrir um computador e examinar sua estrutura interna. Introduzimos programação de linguagem de máquina e escrevemos vários programas de linguagem de máquina. Para tornar essa experiência especialmente valiosa, construímos um computador (pela técnica de *simulação* baseada em software) no qual é possível executar seus programas de linguagem de máquina.

7.35 (***Programação em linguagem de máquina***) Vamos criar um computador chamado Simpletron. Como o nome sugere, é uma máquina simples, mas poderosa. O Simpletron executa programas escritos na única linguagem que ele entende diretamente: Simpletron Machine Language (SML).

O Simpletron contém um *acumulador* — um registrador especial em que as informações são colocadas antes de o Simpletron utilizar essas informações em cálculos ou examiná-las de várias maneiras. Todas as informações no Simpletron são tratadas em termos de *palavras*. Uma palavra é um número decimal de quatro dígitos com sinal, como `+3364`, `-1293`, `+0007` e `-0001`. O Simpletron é equipado com uma memória de 100 palavras e essas palavras são referenciadas por seus números posicionais `00`, `01`, ..., `99`.

Antes de executar um programa de SML, devemos *carregar*, ou colocar, o programa na memória. A primeira instrução (ou expressão) de cada programa de SML sempre é colocada na posição `00`. O simulador começará a executar nessa posição.

Cada instrução escrita em SML ocupa uma palavra da memória do Simpletron (daí porque as instruções são números decimais de quatro dígitos com sinal). Assumiremos que o sinal de uma instrução de SML é sempre mais, mas o sinal de uma palavra de dados pode ser mais ou menos. Cada localização na memória de Simpletron pode conter uma instrução, um valor de dados utilizado por um programa ou uma área de memória não utilizada (e portanto indefinida). Os primeiros dois dígitos de cada instrução do SML são os *códigos de operação* que especificam a operação a ser realizada. Os códigos de operação de SML são resumidos na Figura 7.33.

Código de operação	Significado
Operações de entrada/saída:	
`final int READ = 10;`	Lê uma palavra do teclado para uma posição específica da memória.
`final int WRITE = 11;`	Escreve na tela uma palavra de uma posição específica da memória.
Operações de carregamento/armazenamento:	
`final int LOAD = 20;`	Carrega uma palavra de uma posição específica na memória para o acumulador.
`final int STORE = 21;`	Armazena uma palavra do acumulador para uma posição específica na memória.
Operações aritméticas:	
`final int ADD = 30;`	Adiciona uma palavra de uma posição específica na memória à palavra no acumulador (deixa o resultado no acumulador).
`final int SUBTRACT = 31;`	Subtrai uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador).
`final int DIVIDE = 32;`	Divide uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador).
`final int MULTIPLY = 33;`	Multiplica uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
Operações de transferência de controle:	
`final int BRANCH = 40;`	Desvia para uma posição específica na memória.
`final int BRANCHNEG = 41;`	Desvia para uma posição específica na memória se o acumulador for negativo.
`final int BRANCHZERO = 42;`	Desvia para uma posição específica na memória se o acumulador for zero.
`final int HALT = 43;`	Pare. O programa completou sua tarefa.

Figura 7.33 | Códigos de operação de Simpletron Machine Language (SML).

Os últimos dois dígitos de uma instrução de SML são os *operandos* — o endereço da posição da memória contendo a palavra à qual a operação se aplica. Vamos considerar vários programas simples de SML.

O primeiro programa SML (Figura 7.34) lê dois números do teclado e calcula e exibe sua soma. A instrução `+1007` lê o primeiro número do teclado e o coloca na posição `07` (que foi inicializada como 0). Então a instrução `+1008` lê o próximo número na posição `08`. A instrução *load*, `+2007`, coloca o primeiro número no acumulador e a instrução *add*, `+3008`, adiciona o segundo número ao número no acumulador. *Todas as instruções aritméticas da SML deixam seus resultados no acumulador.* A instrução *store*, `+2109`, coloca o resultado de volta na posição `09` da memória de que a instrução *write*, `+1109`, pega o número e exibe (como um número decimal, de quatro dígitos com sinal). A instrução *halt*, `+4300`, termina a execução.

Posição	Número	Instrução
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Figura 7.34 | O programa SML que lê dois inteiros e calcula sua soma.

O segundo programa SML (Figura 7.35) lê dois números do teclado e determina e exibe o valor maior. Note o uso da instrução `+4107` como uma transferência condicional de controle, muito parecida com a instrução `if` do Java.

Posição	Número	Instrução
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 7.35 | Programa SML que lê dois inteiros e determina o maior.

Agora escreva programas de SML para realizar cada uma das seguintes tarefas:

a) Utilize um loop controlado por sentinela para ler 10 números positivos. Calcule e exiba sua soma.

b) Utilize um loop controlado por contador para ler sete números, alguns negativos e alguns positivos, e compute e exiba sua média.

c) Leia uma série de números, e determine e exiba o maior número. A primeiro número lido indica quantos números devem ser processados.

7.36 (***Simulador de computador***) Nesse problema, você vai construir o seu próprio computador. Não, você não irá soldar componentes. Mais precisamente, você utilizará a poderosa técnica de *simulação baseada em software* para criar um *modelo de software* orientado a objetos do Simpletron do Exercício 7.35. O seu simulador de Simpletron transformará o computador que você está utilizando em um Simpletron e você realmente será capaz de executar, testar e depurar os programas de SML escritos no Exercício 7.35.

Quando você executa seu simulador de Simpletron, ele deve começar exibindo:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction   ***
*** (or data word) at a time. I will display    ***
*** the location number and a question mark (?) ***
*** You then type the word for that location.   ***
*** Type -99999 to stop entering your program.  ***
```

Seu aplicativo deve simular a memória do Simpletron com um array unidimensional `memory` que tem 100 elementos. Agora assuma que o simulador está executando e vamos examinar o diálogo ao entrarmos no programa da Figura 7.35 (Exercício 7.35):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
```

Seu programa deve exibir a posição da memória seguida por um ponto de interrogação. Cada valor à direita de um ponto de interrogação é inserido pelo usuário. Quando o valor de sentinela `-99999` for inserido, o programa deve exibir o seguinte:

```
*** Program loading completed ***
*** Program execution begins  ***
```

O programa de SML agora foi colocado (ou carregado) no array `memory`. Agora o Simpletron executa o programa SML. A execução inicia com a instrução na posição `00` e, como o Java, continua sequencialmente, a menos que dirigido para alguma outra parte do programa por uma transferência de controle.

Utilize a variável `accumulator` para representar o registrador do acumulador. Utilize a variável `instructionCounter` para monitorar a posição na memória que contém a instrução sendo realizada. Utilize a variável `operationCode` para indicar a operação que está sendo atualmente realizado (isto é, os dois dígitos esquerdos da palavra de instrução). Utilize a variável `operand` para indicar a posição da memória em que a instrução atual opera. Portanto, `operand` são os dois dígitos mais à direita da instrução sendo atualmente realizada. Não execute instruções diretamente de memória. Mais precisamente, transfira a próxima instrução que será realizada da memória para uma variável chamada `instructionRegister`. Então "pegue" os dois dígitos esquerdos e os coloque em `operationCode` e "pegue" os dois dígitos direitos e os coloque no `operand`. Quando o Simpletron começa a executar, os registradores especiais são todos inicializados como zero.

Agora vamos "percorrer" a execução da primeira instrução de SML, `+1009` na posição de memória `00`. Esse procedimento é chamado de ciclo de execução da instrução.

O `instructionCounter` informa a posição da próxima instrução que será realizada. Realizamos uma *busca* (fetch) do conteúdo dessa posição a partir de `memory` utilizando a instrução Java:

```
instructionRegister = memory[ instructionCounter ];
```

O código de operação e o operando são extraídos do registrador de instrução pelas instruções:

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Agora o Simpletron deve determinar que o código de operação é realmente uma *read* (*versus* um *write*, um *load* e assim por diante). Um `switch` diferencia entre as 12 operações de SML. Na instrução `switch`, o comportamento das várias instruções SML é simulado como mostrado na Figura 7.36. Discutiremos instruções de desvio em breve e deixaremos as outros para você.

Instrução	Descrição
read:	Exiba o prompt `"Enter an integer"`, depois insira o inteiro e armazene-o na localização `memory[operand]`.
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
halt:	Essa instrução exibe a mensagem `*** Simpletron execution terminated ***`

Figura 7.36 | O comportamento de várias instruções de SML no Simpletron.

Quando o programa de SML completar a execução, o nome e o conteúdo de cada registrador bem como o conteúdo completo de memória devem ser exibidos. Esse tipo de saída costuma ser chamado de dump de computador. Para ajudá-lo a programar seu método de dump, um formato de dump de exemplo é mostrado na Figura 7.37. Observe que um dump depois de executar um programa Simpletron mostraria os valores reais das instruções e os valores dos dados quando a execução terminasse.

```
REGISTERS:
accumulator          +0000
instructionCounter      00
instructionRegister  +0000
operationCode           00
operand                 00
MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Figura 7.37 | Um dump de exemplo.

Vamos prosseguir com a execução da primeira instrução de nosso programa — a saber o `+1009` na posição `00`. Como indicamos, a instrução `switch` simula essa tarefa pedindo ao usuário para inserir um valor, lendo o valor e armazenando-o na posição da memória `memory[operand]`. O valor então é lido na posição `09`.

Nesse ponto, a simulação da primeira instrução é concluída. Tudo que resta é preparar o Simpletron para executar a próxima instrução. Como a instrução recém-realizada não foi uma transferência de controle, precisamos apenas incrementar o registro do contador de instruções como segue:

```
instructionCounter++;
```

Essa ação completa a execução simulada da primeira instrução. O processo inteiro (isto é, o ciclo de execução da instrução) começa de novo com a busca da próxima instrução a ser executada.

Agora vamos considerar como as instruções de desvio — as transferências do controle — são simuladas. Tudo que precisamos fazer é ajustar o valor no contador de instrução apropriadamente. Portanto, a instrução de desvio incondicional (`40`) é simulada dentro do `switch` como:

```
instructionCounter = operand;
```

A instrução "desvie se o acumulador for zero" condicional é simulada como:

```
if ( accumulator == 0 )
   instructionCounter = operand;
```

Nesse ponto, você deve implementar seu simulador de Simpletron e executar cada um dos programas de SML que você escreveu no Exercício 7.35. Se quiser, você pode decorar o SML com recursos adicionais e oferecer esses recursos no seu simulador.

Seu simulador deve verificar vários tipos de erros. Durante a fase de carregamento do programa, por exemplo, cada número que o usuário digita na `memory` do Simpletron deve estar no intervalo `-9999` a `+9999`. O seu simulador deve testar se cada número inserido está nesse intervalo e, se não estiver, continuar solicitando para o usuário reinserir o número até que o usuário insira um número correto.

Durante a fase de execução, seu simulador deve verificar vários erros sérios, como tentativas de divisão por zero, tentativas de execução de códigos de operação inválidos e estouros de acumulador (isto é, operações aritméticas resultando em valores maiores que `+9999` ou menores que `-9999`). Os erros sérios são chamados *erros fatais.* Quando um erro fatal é detectado, seu simulador deve exibir uma mensagem de erro como:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

e deve exibir um dump de computador completo no formato que discutimos previamente. Esse tratamento ajudará o usuário localizar o erro no programa.

7.37 (***Modificações no simulador Simpletron***) No Exercício 7.36, você escreveu uma simulação de software de um computador que executa programas escritos em Simpletron Machine Language (SML). Nesse exercício, são propostas várias modificações e aprimoramentos para o Simulador de Simpletron. Nos exercícios do Capítulo 22, propomos a construção de um compilador que converte programas escritos em uma linguagem de programação de alto nível (uma variação do Basic) para a Simpletron Machine Language. Algumas das seguintes modificações e melhorias podem ser necessárias para executar os programas produzidos pelo compilador:

a) Estender a memória do Simpletron Simulator para conter 1.000 posições da memória a fim de permitir que o Simpletron trate programas maiores.

b) Permita que o simulador realize os cálculos restantes. Essa modificação requer uma instrução SML adicional.

c) Permitir que o simulador realize cálculos de exponenciação. Essa modificação requer uma instrução SML adicional.

d) Modifique o simulador para utilizar valores hexadecimais em vez de valores inteiros para representar as instruções SML.

e) Modificar o simulador para permitir saída de uma nova linha. Essa modificação requer uma instrução SML adicional.

f) Modificar o simulador para processar valores de ponto flutuante além de valores inteiros.

g) Modificar o simulador para tratar entrada de string. [*Dica:* Cada palavra do Simpletron pode ser dividida em dois grupos, cada uma armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o (ver o Apêndice B) equivalente decimal ASCII de um caractere. Adicione uma instrução de linguagem de máquina que irá inserir uma string e armazenar a string, iniciando em uma posição da memória específica do Simpletron. A primeira metade da palavra nessa posição será uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada sucessiva meia-palavra contém um caractere ASCII como dois dígitos decimais expressos. A instrução de linguagem de máquina converte cada caractere em seu equivalente ASCII e atribui a ele uma meia-palavra.]

h) Modificar o simulador para tratar saída de strings armazenadas no formato da parte (g). [*Dica:* Adicione uma instrução de linguagem de máquina que exiba uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada sucessiva meia-palavra contém um caractere ASCII como dois dígitos decimais expressos. A instrução de linguagem de máquina verifica o comprimento e exibe a string traduzindo cada número de dois dígitos em seu caractere equivalente.]

Fazendo a diferença

7.38 **(*Enquete*)** A Internet e a Web permitem que mais pessoas comuniquem-se, abracem uma causa, expressem opiniões etc. Em 2008, os candidatos presidenciais norte-americanos utilizaram a Internet intensivamente para enviar mensagens e arrecadar fundos para suas campanhas. Neste exercício, você escreverá um programa simples de enquete que permite aos usuários avaliar cinco questões relacionadas à consciência social de 1 (a menos importante) a 10 (a mais importante). Selecione cinco causas que são importantes para você (por exemplo, questões políticas, problemas ambientais globais). Utilize um array unidimensional `topics` (do tipo `String`) para armazenar as cinco causas. Para resumir as respostas da enquete, utilize um array bidimensional `responses` de 5 linhas e 10 colunas (do tipo `int`), cada linha correspondendo a um elemento no array `topics`. Quando o programa executa, ele deve solicitar que o usuário avalie cada questão. Peça que seus amigos e sua família respondam à pesquisa. Então faça com que o programa exiba um resumo dos resultados, incluindo:

a) Um relatório tabulado com os cinco tópicos na primeira coluna e as 10 avaliações na primeira linha, listando em cada coluna subsequente o número de avaliações recebidas por cada tópico.

b) Na última coluna, mostre a média das avaliações para essa questão.

c) Que questão recebeu o total de pontos mais alto? Exiba a questão e o total de pontos.

d) Que questão recebeu o total de pontos mais baixo? Exiba a questão e o total de pontos.

Em vez dessa divisão absurda em sexos, deveríamos classificar as pessoas como estáticas e dinâmicas.
— **Evelyn Waugh**

Este é um mundo para esconder virtudes?
— **William Shakespeare**

Mas o que, para servir a nossos propósitos particulares, proíbe trapacear nossos amigos?
— **Charles Churchill**

Mas, sobretudo, sê a ti próprio fiel.
— **William Shakespeare**

Não seja "consistente", mas simplesmente autêntico.
— **Oliver Wendell Holmes, Jr.**

8

Classes e objetos: uma visão mais aprofundada

Objetivos

Neste capítulo, você aprenderá:

- O encapsulamento e o ocultamento de dados.
- A utilizar a palavra-chave `this`.
- A utilizar variáveis e métodos `static`.
- A importar membros `static` de uma classe.
- A utilizar o tipo `enum` para criar conjuntos de constantes com identificadores únicos.
- A declarar constantes `enum` com parâmetros.
- A organizar classes em pacotes para promover a reutilização.

Sumário

8.1 Introdução
8.2 Estudo de caso da classe `Time`
8.3 Controlando o acesso a membros
8.4 Referenciando membros do objeto atual com a referência `this`
8.5 Estudo de caso da classe `Time`: construtores sobrecarregados
8.6 Construtores padrão e sem argumentos
8.7 Notas sobre os métodos *set* e *get*
8.8 Composição
8.9 Enumerações
8.10 Coleta de lixo e o método `finalize`
8.11 Membros da classe `static`
8.12 Importação `static`
8.13 Variáveis de instância `final`
8.14 Estudo de caso da classe `Time`: criando pacotes
8.15 Acesso de pacote
8.16 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando objetos com imagens gráficas
8.17 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

8.1 Introdução

Agora faremos uma análise mais profunda da construção de classes, controle de acesso a membros de uma classe e criação de construtores. Discutiremos a composição — uma capacidade que permite a uma classe conter referências a objetos de outras classes como membros. Reexaminamos o uso dos métodos *set* e *get*. Lembre-se de que a Seção 6.10 introduziu o tipo `enum` básico para declarar um conjunto de constantes. Neste capítulo, discutimos o relacionamento entre tipos e classes `enum`, demonstrando que um `enum`, como ocorre com uma classe, pode ser declarado no seu próprio arquivo com construtores, métodos e campos. O capítulo também discute os membros da classe `static` e variáveis de instância `final` em detalhes. Investigaremos questões como a capacidade de reutilização de software, abstração e encapsulamento de dados. Por fim, explicaremos como organizar as classes nos pacotes para ajudar a gerenciar grandes aplicativos e promover a reutilização e então mostraremos um relacionamento especial entre as classes no mesmo pacote.

O Capítulo 9, "Programação orientada a objetos: herança", e o Capítulo 10, "Programação orientada a objetos: polimorfismo", apresentaram mais duas tecnologias-chave de programação orientada a objetos.

8.2 Estudo de caso da classe `Time`

Nosso primeiro exemplo consiste em duas classes — `Time1` (Figura 8.1) e `Time1Test` (Figura 8.2). A classe `Time1` representa a hora do dia. A classe `Time1Test` é uma classe de aplicativo na qual o método `main` cria um objeto da classe `Time1` e invoca seus métodos. Essas classes devem ser declaradas em arquivos separados porque são classes `public`. A saída desse programa é mostrada na Figura 8.2.

Declaração da classe `Time1`

As variáveis de instância `private int hour`, `minute` e `second` da classe `Time1` (Figura 8.1, linhas 6–8) representam a hora no formato de hora universal (formato de relógio de 24 horas em que as horas estão no intervalo de 0 a 23). A classe `Time1` contém os métodos `public setTime` (linhas 12–17), `toUniversalString` (linhas 20–23) e `toString` (linhas 26–31). Esses métodos também são chamados **serviços** `public` ou **interface** `public` que a classe fornece para seus clientes.

```
1  // Figura 8.1: Time1.java
2  // Declaração de classe Time1 mantém a hora no formato de 24 horas.
3
4  public class Time1
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // configura um novo valor de hora usando formato universal;
11    // assegura que os dados permaneçam consistentes configurando valores inválidos como zero
12    public void setTime( int h, int m, int s )
13    {
```

```
14          hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida horas
15          minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida minutos
16          second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida segundos
17       } // fim do método setTime
18
19       // converte em String no formato de hora universal (HH:MM:SS)
20       public String toUniversalString()
21       {
22          return String.format( "%02d:%02d:%02d", hour, minute, second );
23       } // fim do método do toUniversalString
24
25       // converte em String no formato padrão hora (H:MM:SS AM ou PM)
26       public String toString()
27       {
28          return String.format( "%d:%02d:%02d %s",
29             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30             minute, second, ( hour < 12 ? "AM" : "PM" ) );
31       } // fim do método toString
32    } // fim da classe Time1
```

Figura 8.1 | Declaração da classe `Time1` mantém a hora no formato de 24 horas.

Construtor padrão

Nesse exemplo, a classe `Time1` não declara um construtor, portanto a classe tem um construtor padrão fornecido pelo compilador. Cada variável de instância recebe implicitamente o valor padrão de 0 para um `int`. Observe que as variáveis de instância também podem ser inicializadas quando declaradas no corpo da classe utilizando-se a mesma sintaxe de inicialização de uma variável local.

O método `setTime` e como manter consistentes os dados

O método `setTime` (linhas 12–17) é um método `public` que declara três parâmetros `int` e os utiliza para configurar a hora. Uma expressão condicional testa a validade de cada argumento para determinar se ele está ou não em um intervalo especificado. Por exemplo, o valor `hour` (linha 14) deve ser maior ou igual a 0 e menor que 24, porque o formato de hora universal representa horas como inteiros de 0 a 23 (por exemplo, 1 PM é a hora 13 e 11 PM é a hora 23; meia-noite é a hora 0 e meio-dia é a hora 12). De maneira semelhante, os valores `minute` e `second` (linhas 15–16) devem ser maiores ou iguais a 0 e menores que 60. Quaisquer valores fora desses intervalos são configurados como zero para assegurar que um objeto `Time1` sempre contenha dados consistentes — isto é, os valores dos dados do objeto sempre são mantidos em um intervalo, mesmo se os valores fornecidos como argumentos para o método `setTime` estiverem *incorretos*. Nesse exemplo, zero é um valor consistente para `hour`, `minute` e `second`. De fato, quando se cria um objeto `Time1`, seus `hour`, `minute` e `second` são configurados como zero por padrão; assim, um objeto `Time1` contém dados consistentes a partir do momento em que é criado.

Valores corretos versus *valores consistentes*

É importante distinguir entre um valor *correto* e um valor *consistente*. Um valor consistente para `minute` deve estar entre o intervalo 0 a 59. Um valor correto para `minute` em determinado aplicativo, como simplesmente informar as horas, seria o minuto real naquela hora do dia. Suponha que você esteja acertando a hora de um relógio. Se a hora real estiver 17 minutos depois da hora e você acertar o relógio acidentalmente como 19 minutos depois da hora, 19 é um valor *consistente* (0 a 59), mas não um valor *correto*. Se acertar o relógio para 17 minutos depois da hora, então 17 é um valor correto — e um valor correto é *sempre* um valor consistente.

Projetamos o nosso método `setTime` para que quando receber um valor inconsistente, ele configure a variável de instância correspondente como zero. Isso certamente é um valor *consistente*, mas provavelmente não é *correto*. Há uma abordagem melhor? Podemos simplesmente deixar o objeto no estado atual, sem alterar a variável de instância. Considerando que objetos `Time` iniciam em um estado consistente (com `hour`, `minute` e `second` configurados como zero) e que o método `setTime` rejeitaria qualquer valor inconsistente, haveria sempre a *garantia* de o objeto estar em um estado consistente. Muitas vezes esse seria o último estado correto do objeto, o qual alguns designers acham ser superior a configurar as variáveis de instância como zero.

Problemas potenciais

Um problema com as abordagens discutidas até aqui é que elas não informam o código de cliente de valores inconsistentes. Poderíamos fazer `setTime` retornar um valor como `true` se todos os valores forem consistentes e `false` se alguns dos valores forem inconsistentes. O chamador verificaria o valor de retorno, e se ele fosse `false`, tentaria estabelecer a hora novamente. Um problema com essa abordagem é que algumas tecnologias Java (como JavaBeans) exigem que os métodos *set* retornem `void`. No Capítulo 11, "Tratamento de exceções", você aprenderá técnicas elegantes que permitem que seus métodos indiquem quando valores inconsistentes são recebidos.

Métodos `toUniversalString` e `toString`

O método `toUniversalString` (linhas 20–23) não aceita argumentos e retorna uma `String` no formato de horas universal, consistindo em dois dígitos cada para a hora, minuto e segundo. Por exemplo, se a hora fosse 1:30:07 PM, o método retornaria `13:30:07`. A linha 22 utiliza o método `static` **`format`** da classe `String` para retornar uma `String` que contém os valores `hour`, `minute` e `second` formatados, cada um com dois dígitos e possivelmente um `0` à esquerda (especificado com o flag `0`). O método `format` é semelhante ao método `System.out.printf` exceto que `format` retorna uma `String` formatada antes de exibi-la em uma janela de comando. A `String` formatada é retornada pelo método `toUniversalString`.

O método `toString` (linhas 26–31) não recebe argumentos e retorna uma `String` em formato de hora padrão, que consiste nos valores `hour`, `minute` e `second` separados por dois-pontos e seguidos por AM ou PM (por exemplo, `1:27:06 PM`). Como o método `toUniversalString`, o método `toString` utiliza o método `static String format` para formatar os valores `minute` e `second` como valores de dois dígitos com zeros à esquerda, se necessário. A linha 29 utiliza um operador condicional (`?:`) para determinar o valor para `hour` na `String` — se `hour` for `0` ou `12` (AM ou PM), ele aparece como 12; caso contrário, ele aparece como um valor entre 1 e 11. O operador condicional na linha 30 determina se AM ou PM será retornado como parte da `String`.

Com base na Seção 6.5, lembre-se de que todos os objetos em Java contêm um método `toString` que retorna uma representação `String` do objeto. Escolhemos retornar uma `String` que contém a hora no formato de hora padrão. O método `toString` pode ser chamado implicitamente sempre que um objeto `Time1` aparece no código em que uma `String` é necessária, como o valor para gerar a saída com um especificador de formato `%s` em uma chamada a `System.out.printf`.

Utilizando a classe `Time1`

Como viu no Capítulo 3, cada classe que você declara representa um novo tipo em Java. Portanto, depois de declarar a classe `Time1`, você pode utilizá-la como um tipo em declarações como

```
Time1 sunset; // sunset pode manter uma referência a um objeto Time1
```

A classe de aplicativo `Time1Test` (Figura 8.2) utiliza a classe `Time1`. A linha 9 declara e cria um objeto `Time1` e o atribui à variável local `time`. Observe que `new` invoca implicitamente o construtor padrão da classe `Time1`, uma vez que `Time1` não declara nenhum construtor. As linhas 12–16 primeiro geram a saída da hora no formato de hora universal (invocando o método `toUniversalString` de `time` na linha 13), então no formato de hora padrão (invocando explicitamente o método `toString` de `time` na linha 15) para confirmar que o objeto `Time1` foi inicializado adequadamente.

```
1  // Figura 8.2: Time1Test.java
2  // objeto Time1 utilizado em um aplicativo.
3
4  public class Time1Test
5  {
6     public static void main( String[] args )
7     {
8        // cria e inicializa um objeto Time1
9        Time1 time = new Time1(); // invoca o construtor Time1
10
11       // gera saída de representações de string da hora
12       System.out.print( "The initial universal time is: " );
13       System.out.println(time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println(time.toString() );
16       System.out.println(); // gera saída de uma linha em branco
17
18       // altera a hora e gera saída da hora atualizada
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println(time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println(time.toString() );
24       System.out.println(); // gera saída de uma linha em branco
25
26       // configura a hora com valores inválidos; gera saída da hora atualizada
27       time.setTime( 99, 99, 99 );
28       System.out.println( "After attempting invalid settings:" );
29       System.out.print( "Universal time: " );
30       System.out.println(time.toUniversalString() );
31       System.out.print( "Standard time: " );
32       System.out.println(time.toString() );
33    } // fim de main
34 } // fim da classe Time1Test
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Figura 8.2 | Objeto `Time1` utilizado em um aplicativo.

A linha 19 invoca o método `setTime` do objeto `time` sem mudar a hora. Em seguida, as linhas 20–24 geram a saída da hora novamente nos dois formatos para confirmar que ela foi configurada corretamente.

Para ilustrar que ele mantém o objeto em um estado consistente, a linha 27 chama o método `setTime` com argumentos inconsistentes de 99 para `hour`, `minute` e `second`. As linhas 28–32 geram a saída da hora novamente nos dois formatos para confirmar o estado consistente do objeto e, então, o programa termina. As duas últimas linhas da saída mostram que a hora é reconfigurada para meia-noite — o valor inicial de um objeto `Time1` — depois de uma tentativa de configurar a hora com três valores fora do intervalo.

Notas sobre a declaração da classe `Time1`

Considere as várias questões de projeto classe com relação à classe `Time1`. As variáveis de instância `hour`, `minute` e `second` são declaradas `private`. A representação real dos dados utilizada dentro da classe não diz respeito aos clientes da classe. Por exemplo, seria perfeitamente razoável para `Time1` representar a hora internamente como o número de segundos a partir da meia-noite ou o número de minutos e segundos a partir da meia-noite. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados sem estarem cientes disso. (O Exercício 8.5 pede para você representar a hora na classe `Time1` como o número de segundos a partir da meia-noite e mostrar que de fato nenhuma alteração está visível aos clientes da classe.)

Observação de engenharia de software 8.1

As classes simplificam a programação, porque o cliente pode utilizar somente os métodos `public` expostos pela classe. Normalmente, esses métodos são direcionados aos clientes em vez de à implementação. Os clientes não estão cientes de, nem envolvidos em, uma implementação da classe. Os clientes geralmente se preocupam com o que *a classe faz, mas não* como *a classe faz isso.*

Observação de engenharia de software 8.2

As interfaces mudam com menos frequência que as implementações. Quando uma implementação muda, o código dependente de implementação deve alterar correspondentemente. Ocultar a implementação reduz a possibilidade de que outras partes do programa irão se tornar dependentes dos detalhes sobre a implementação da classe.

8.3 Controlando o acesso a membros

Os modificadores de acesso `public` e `private` controlam o acesso às variáveis e métodos de uma classe. No Capítulo 9, introduziremos o modificador de acesso adicional `protected`. Como afirmado na Seção 8.2, o principal propósito dos métodos `public` é apresentar aos clientes da classe uma visualização dos serviços que a classe fornece (a interface `public` da classe). Os clientes não precisam se preocupar com a maneira como a classe realiza suas tarefas. Por essa razão, as variáveis `private` e os métodos `private` da classe (isto é, os detalhes da sua implementação) não são acessíveis aos seus clientes.

A Figura 8.3 demonstra que os membros da classe `private` não são acessíveis fora da classe. As linhas 9–11 tentam acessar diretamente as variáveis de instância `private` `hour`, `minute` e `second` do objeto `Time1 time`. Quando esse programa é compilado, o compilador gera mensagens de erro de que esses membros `private` não são acessíveis. [*Nota:* esse programa pressupõe que seja utilizada a classe `Time1` da Figura 8.1.]

Erro comum de programação 8.1

Uma tentativa por parte de um método que não é membro de uma classe de acessar um membro `private` dessa classe é um erro de compilação.

```
1  // Figura 8.3: MemberAccessTest.java
2  // Membros privados da classe Time1 não são acessíveis.
3  public class MemberAccessTest
4  {
```

```
5       public static void main( String[] args )
6       {
7          Time1 time = new Time1(); // cria e inicializa o objeto Time1
8
9          time.hour = 7; // erro: hour tem acesso privado em Time1
10         time.minute = 15; // erro: minute tem acesso privado em Time1
11         time.second = 30; // erro: second tem acesso privado em Time1
12      } // fim de main
13   } // fim da classe MemberAccessTest
```

```
MemberAccessTest.java:9: hour has private access in Time1
      time.hour = 7; // erro: hour tem acesso privado in Time1
         ^
MemberAccessTest.java:10: minute has private access in Time1
      time.minute = 15; // erro: minute tem acesso privado in Time1
         ^
MemberAccessTest.java:11: second has private access in Time1
      time.second = 30; // erro: second tem acesso privado in Time1
         ^
3 errors
```

Figura 8.3 | Membros privados da classe `Time1` não são acessíveis.

8.4 Referenciando membros do objeto atual com a referência `this`

Todo objeto pode acessar uma referência a si próprio com a palavra-chave **`this`** (às vezes chamada **referência `this`**). Quando um método não `static` é chamado para um objeto particular, o corpo do método utiliza implicitamente a palavra-chave `this` para referenciar as variáveis de instância do objeto e outros métodos. Isso permite que o código da classe saiba que o objeto deve ser manipulado. Como verá na Figura 8.4, você também pode usar a palavra-chave `this` explicitamente no corpo de um método não `static`. A Seção 8.5 mostra uma outra utilização interessante de palavra-chave `this`. A Seção 8.11 explica por que palavra-chave `this` não pode ser utilizada em um método `static`.

Agora demonstramos o uso implícito e explícito da referência `this` (Figura 8.4). Note que esse exemplo é o primeiro em que declaramos duas classes em um único arquivo — a classe `ThisTest` é declarada nas linhas 4–11 e a classe `SimpleTime`, nas linhas 14–47. Fizemos isso para demonstrar que quando você compila um arquivo `.java` contendo mais de uma classe, o compilador produz um arquivo separado da classe com a extensão `.class` para cada classe compilada. Nesse caso, dois arquivos separados são produzidos — `SimpleTime.class` e `ThisTest.class`. Quando um arquivo código-fonte (`.java`) contém várias declarações de classe, o compilador coloca ambos os arquivos de classe dessas classes no mesmo diretório. Note também na Figura 8.4 que apenas a classe `ThisTest` é declarada `public`. Um arquivo de código-fonte pode conter somente uma classe `public` — caso contrário, um erro de compilação ocorre. Classes não `public` só podem ser utilizadas por outras classes no mesmo pacote. Desse modo, nesse exemplo, a classe `SimpleTime` pode ser utilizada somente pela classe `ThisTest`.

```
1    // Figura 8.4: ThisTest.java
2    // this utilizado implícita e explicitamente para referência a
     // membros de um objeto.
3
4    public class ThisTest
5    {
6       public static void main( String[] args )
7       {
8          SimpleTime time = new SimpleTime( 15, 30, 19 );
9          System.out.println( time.buildString() );
10      } // fim de main
11   } // fim da classe ThisTest
12
13   // classe SimpleTime demonstra a referência "this"
14   class SimpleTime
15   {
16      private int hour; // 0-23
17      private int minute; // 0-59
18      private int second; // 0-59
19
20      // se o construtor utilizar nomes de parâmetro idênticos a
```

```
21       // nomes de variáveis de instância a referência "this" será
22       // exigida para distinguir entre nomes
23       public SimpleTime( int hour, int minute, int second )
24       {
25          this.hour = hour; // configura a hora do objeto "this"
26          this.minute = minute; // configura os minutos do objeto "this"
27          this.second = second; // configura os segundos do objeto "this"
28       } // fim do construtor SimpleTime
29
30       // usa "this" explícito e implícito para chamar toUniversalString
31       public String buildString()
32       {
33          return String.format( "%24s: %s\n%24s: %s",
34             "this.toUniversalString()",this.toUniversalString(),
35             "toUniversalString()",toUniversalString() );
36       } // fim do método buildString
37
38       // converte em String no formato de hora universal (HH:MM:SS)
39       public String toUniversalString()
40       {
41          // "this" não é requerido aqui para acessar variáveis de instância,
42          // porque o método não tem variáveis locais com os mesmos
43          // nomes das variáveis de instância
44          return String.format( "%02d:%02d:%02d",
45             this.hour, this.minute, this.second );
46       } // fim do método do toUniversalString
47    } // fim da classe SimpleTime
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Figura 8.4 | `this` utilizado implícita e explicitamente como uma referência a membros de um objeto.

A classe `SimpleTime` (linhas 14–47) declara três variáveis de instância `private` — `hour`, `minute` e `second` (linhas 16–18). O construtor (linha 23–28) recebe três argumentos `int` para inicializar um objeto `SimpleTime`. Observe que utilizamos nomes de parâmetro para o construtor (linha 23) idênticos aos nomes das variáveis de instância da classe (linhas 16–18). Não recomendamos essa prática, mas a mostramos aqui para "sombrear" (ocultar) as variáveis de instância correspondentes a fim de que pudéssemos ilustrar um caso em que o uso explícito da referência `this` é exigido. Se um método contiver uma variável local com o mesmo nome de um campo, esse método irá referenciar a variável local em vez do campo. Nesse caso, a variável local "sombreia" o campo no escopo do método. Entretanto, o método pode utilizar a referência `this` para referenciar o campo "sombreado" explicitamente, como mostrado nos lados esquerdos das atribuições nas linhas 25–27 para variáveis de instância escondidas de `SimpleTime`.

O método `buildString` (linhas 31–36) retorna uma `String` criada por uma instrução que usa a referência `this` explícita e implicitamente. A linha 34 a utiliza explicitamente para chamar o método `toUniversalString`. A linha 35 a usa implicitamente para chamar o mesmo método. Observe que as duas linhas realizam a mesma tarefa. Em geral, você não utilizará `this` explicitamente para referenciar outros métodos dentro do objeto atual. Além disso, observe que a linha 45 no método `toUniversalString` utiliza explicitamente a referência `this` para acessar cada variável de instância. Isso não é necessário aqui, porque o método não tem variáveis locais que sombreiam as variáveis de instância da classe.

Erro comum de programação 8.2

Frequentemente é um erro de lógica quando um método contém um parâmetro ou variável local com o mesmo nome de um campo da classe. Nesse caso, utilize a referência `this` se desejar acessar o campo da classe — caso contrário, o parâmetro ou variável local do método será referenciado.

Dica de prevenção de erro 8.1

Evite nomes de parâmetros ou variáveis locais nos métodos que conflitem com nomes de campos. Isso ajuda a evitar bugs sutis, difíceis de corrigir.

Dica de desempenho 8.1

O Java conserva armazenamento mantendo somente uma cópia de cada método por classe — esse método é invocado por cada objeto dessa classe. Cada objeto, por outro lado, tem sua própria cópia das variáveis de instância da classe (isto é, campos não `static`*). Cada método da classe utiliza implicitamente* `this` *para determinar o objeto específico da classe a manipular.*

A classe de aplicativo `ThisTest` (linhas 4–11) demonstra a classe `SimpleTime`. A linha 8 cria uma instância da classe `SimpleTime` e invoca seu construtor. A linha 9 invoca o método `buildString` do objeto e então exibe os resultados.

8.5 Estudo de caso da classe `Time`: construtores sobrecarregados

Como você sabe, é possível declarar seu próprio construtor para especificar como objetos de uma classe devem ser inicializados. A seguir, demonstramos uma classe com vários **construtores sobrecarregados** que permitem que objetos dessa classe sejam inicializados de diferentes maneiras. Para sobrecarregar construtores, basta fornecer múltiplas declarações de construtor com assinaturas diferentes. A partir da Seção 6.12, lembre-se de que o compilador diferencia assinaturas pelo *número* de parâmetros, por *tipos* dos parâmetros e pela *ordem* dos tipos de parâmetro em cada assinatura.

Classe `Time2` com construtores sobrecarregados

O construtor padrão da classe `Time1` (Figura 8.1) inicializou `hour`, `minute` e `second` com seus valores `0` padrão (que é meia-noite na hora universal). O construtor padrão não permite que clientes da classe inicializem a hora com valores não zero específicos. A classe `Time2` (Figura 8.5) contém cinco construtores sobrecarregados que fornecem maneiras convenientes de inicializar os objetos da nova classe `Time2`. Cada construtor inicializa o objeto para iniciar em um estado consistente. Nesse programa, quatro construtores invocam um quinto, que, por sua vez, chama o método `setTime` para assegurar que o valor fornecido para `hour` está dentro do intervalo de 0 a 23 e que os valores para `minute` e `second` estão no intervalo de 0 a 59. Se um valor estiver fora do intervalo, ele é configurado como zero por `setTime` (mais uma vez para assegurar que cada variável de instância permaneça em um estado consistente). O compilador invoca o construtor apropriado correspondendo o número, os tipos e a ordem dos tipos dos argumentos especificados na chamada de construtor com o número, os tipos e a ordem dos tipos dos parâmetros especificados em cada declaração de construtor. Observe que a classe `Time2` também fornece métodos *set* e *get* para cada variável de instância.

```
1  // Figura 8.5: Time2.java
2  // declaração da classe Time2 com construtores sobrecarregados.
3
4  public class Time2
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // construtor sem argumento Time2 : inicializa cada variável de instância
11    // com zero; assegura que objetos Time2 iniciam em um estado consistente
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoca o construtor Time2 com três argumentos
15    } // fim do construtor sem argumento Time2
16
17    // Construtor Time2: hour fornecido, min e sec padronizados como 0
18    public Time2( int h )
19    {
20       this( h, 0, 0 ); // invoca o construtor Time2 com três argumentos
21    } // fim do construtor de um argumento Time2
22
23    // Construtor Time2: hour e min fornecidos, sec padronizado como 0
24    public Time2( int h, int m )
25    {
26       this( h, m, 0 ); // invoca o construtor Time2 com três argumentos
27    } // fim do construtor de dois argumentos Time2
28
29    // Construtor Time2: hour, min e sec fornecidos
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoca setTime para validar a hora
33    } // fim do construtor de três argumentos Time2
34
```

```
35       // Construtor Time2: outro objeto Time2 fornecido
36       public Time2( Time2 time )
37       {
38          // invoca o construtor de três argumentos Time2
39          this( time.getHour(), time.getMinute(), time.getSecond() );
40       } // fim do construtor Time2 com um argumento de objeto Time2
41
42       // Métodos set
43       // configura um novo valor de hora usando o formato universal;
44       // assegura que os dados permaneçam consistentes configurando
44       // valores inválidos como zero
45       public void setTime( int h, int m, int s )
46       {
47          setHour( h ); // configura hour
48          setMinute( m ); // configura minute
49          setSecond( s ); // configura second
50       } // fim do método setTime
51
52       // valida e configura a hora
53       public void setHour( int h )
54       {
55          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56       } // fim do método setHour
57
58       // valida e configura os minutos
59       public void setMinute( int m )
60       {
61          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62       } // fim do método setMinute
63
64       // valida e configura os segundos
65       public void setSecond( int s )
66       {
67          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68       } // fim do método setSecond
69
70       // Métodos get
71       // obtém valor da hora
72       public int getHour()
73       {
74          return hour;
75       } // fim do método getHour
76
77       // obtém valor dos minutos
78       public int getMinute()
79       {
80          return minute;
81       } // fim do método getMinute
82
83       // obtém valor dos segundos
84       public int getSecond()
85       {
86          return second;
87       } // fim do método getSecond
88
89       // converte em String no formato de hora universal (HH:MM:SS)
90       public String toUniversalString()
91       {
92          return String.format(
93             "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
94       } // fim do método do toUniversalString
95
96       // converte em String no formato padrão de data (H:MM:SS AM ou PM)
97       public String toString()
98       {
99          return String.format( "%d:%02d:%02d %s",
100            ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
101            getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
102      } // fim do método toString
103   } // fim da classe Time2
```

Figura 8.5 | Classe Time2 com construtores sobrecarregados.

Construtores da classe `Time2`

As linhas 12–15 declaram o chamado **construtor sem argumento** que é invocado sem argumentos. Assim que você declara qualquer construtor em uma classe, o compilador não fornecerá um construtor padrão. Esse construtor sem argumento assegura que os clientes da classe `Time2` podem criar objetos `Time2` com valores padrão. Esse construtor simplesmente inicializa o objeto como especificado no corpo do construtor. No corpo, introduzimos um uso da referência `this` que só é permitido como a primeira instrução no corpo de um construtor. A linha 14 utiliza `this` na sintaxe de chamada de método para invocar o construtor `Time2` que recebe três argumentos (linhas 30–33). O construtor sem argumentos passa valores de 0 de `hour`, `minute` e `second` para o construtor com três parâmetros. Usar a referência `this` como mostrado aqui é uma maneira popular de reutilizar código de inicialização fornecido por outro dos construtores da classe em vez de definir um código semelhante no corpo do construtor sem argumentos. Utilizamos essa sintaxe em quatro dos cinco construtores `Time2` para tornar a classe mais fácil de manter e modificar. Se for necessário alterar a maneira como objetos da classe `Time2` são inicializados, somente o construtor que os outros construtores da classe chamam precisará ser modificado. De fato, mesmo esse construtor talvez não precise ser modificado nesse exemplo. Esse construtor simplesmente chama o método `setTime` para realizar a inicialização real, portanto é possível que as alterações que a classe poderia exigir seriam localizadas pelos métodos *set*.

Erro comum de programação 8.3

É um erro de sintaxe se `this` for utilizado no corpo de um construtor para chamar um outro construtor da mesma classe se essa chamada não for a primeira instrução do construtor. Também é um erro de sintaxe se um método tentar invocar um construtor diretamente via `this`.

As linhas 18–21 declaram um construtor `Time2` com um parâmetro `int` único que representa a `hour` que é passada com 0 em `minute` e `second` para o construtor nas linhas 30–33. As linhas 24–27 declaram um construtor `Time2` que recebe dois parâmetros `int` que representam a `hour` e `minute`, passados com 0 de `second` para o construtor nas linhas 30–33. Como ocorre com o construtor sem argumentos, cada um desses construtores invoca o construtor nas linhas 30–33 para minimizar a duplicação de código. As linhas 30–33 declaram o construtor `Time2` que recebe três parâmetros `int` que representam a `hour`, `minute` e `second`. Esse construtor chama `setTime` para inicializar as variáveis de instância com valores consistentes.

Erro comum de programação 8.4

Um construtor pode chamar métodos da classe. Esteja ciente de que as variáveis de instância talvez ainda não estejam em um estado consistente, porque o construtor está no processo de inicialização do objeto. Utilizar variáveis de instância antes de elas serem inicializadas adequadamente é um erro de lógica.

As linhas 36–40 declaram um construtor `Time2` que recebe uma referência para outro objeto `Time2`. Nesse caso, os valores do argumento `Time2` são passados ao construtor de três argumentos nas linhas 30–33 para inicializar `hour`, `minute` e `second`. Observe que a linha 39 poderia ter acessado diretamente os valores de `hour`, `minute` e `second` do argumento do construtor `time` com as expressões `time.hour`, `time.minute` e `time.second` — mesmo que `hour`, `minute` e `second` sejam declarados como variáveis `private` da classe `Time2`. Isso se deve a um relacionamento especial entre objetos da mesma classe. Veremos a seguir por que é preferível utilizar os métodos *get*.

Observação de engenharia de software 8.3

Quando um objeto de uma classe contém uma referência a um outro objeto da mesma classe, o primeiro objeto pode acessar todos os dados e métodos do segundo objeto (incluindo aqueles que são `private`).

Notas com relação aos construtores e métodos set *e* get *da classe `Time2`*

Observe que os métodos *set* e *get* de `Time2` são chamados em todas as partes da classe. Em particular, o método `setTime` chama os métodos `setHour`, `setMinute` e `setSecond` nas linhas 47–49, e os métodos `toUniversalString` e `toString` chamam os métodos `getHour`, `getMinute` e `getSecond` na linha 93 e linhas 100–101, respectivamente. Em cada caso, esses métodos poderiam ter acessado os dados privados da classe diretamente sem chamar os métodos *set* e *get*. Mas considere a possibilidade de alterar a representação da hora de três valores `int` (requerendo 12 bytes de memória) para um único valor `int` a fim de representar o número total de segundos que se passou desde a meia-noite (requerendo 4 bytes de memória). Se fizéssemos essa alteração, somente o corpo dos métodos que acessam os dados `private` diretamente precisaria mudar — em particular, os métodos *set* e *get* individuais para `hour`, `minute` e `second`. Não haveria necessidade de modificar o corpo dos métodos `setTime`, `toUniversalString` ou `toString` porque eles não acessam os dados diretamente. Projetar a classe dessa maneira reduz a probabilidade de erros de programação ao alterar a implementação da classe.

Da mesma forma, cada construtor `Time2` poderia ser escrito para incluir uma cópia das instruções apropriadas a partir dos métodos `setHour`, `setMinute` e `setSecond`. Fazer isso talvez seja um pouco mais eficiente, porque a chamada extra ao construtor e a chamada a `setTime` são eliminadas. Entretanto, duplicar instruções em múltiplos métodos ou construtores dificulta a alteração da representação interna de dados da classe. Fazer com que os construtores `Time2` chamem o construtor com três argumentos (ou mesmo chamem `setTime` diretamente) requer que as alterações na implementação de `setTime` sejam feitas somente uma vez. Além disso, o compilador pode otimizar programas removendo chamadas para métodos simples e substituindo-os pelo código expandido de suas declarações — uma técnica conhecida como **colocar o código em linha**, o que aprimora o desempenho de programa.

Observação de engenharia de software 8.4

Ao implementar um método de uma classe, utilize os métodos set *e* get *da classe para acessar os dados* `private` *da classe. Isso simplifica a manutenção do código e reduz a probabilidade de erros.*

Utilizando construtores sobrecarregados da classe `Time2`

A classe `Time2Test` (Figura 8.6) invoca os construtores sobrecarregados `Time2` (linhas 8–13). A linha 8 invoca o construtor sem argumento (Figura 8.5, linhas 12–15). As linhas 9–13 do programa demonstram como passar argumentos para os outros construtores `Time2`. A linha 9 invoca o construtor nas linhas 18–21 da Figura 8.5. A linha 10 invoca o construtor nas linhas 24–27 da Figura 8.5. As linhas 11–12 invocam o construtor nas linhas 30–33 da Figura 8.5. A linha 13 invoca o construtor nas linhas 36–40 da Figura 8.5. O aplicativo exibe a representação `String` de cada objeto `Time2` para confirmar que eles foram foi inicializados corretamente.

```
1  // Figura 8.6: Time2Test.java
2  // Construtores sobrecarregados utilizados para inicializar objetos Time2.
3
4  public class Time2Test
5  {
6     public static void main( String[] args )
7     {
8        Time2 t1 = new Time2(); // 00:00:00
9        Time2 t2 = new Time2( 2 ); // 02:00:00
10       Time2 t3 = new Time2( 21, 34 ); // 21:34:00
11       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 ); // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
20       System.out.println(
21          "t2: hour specified; minute and second defaulted" );
22       System.out.printf( "   %s\n", t2.toUniversalString() );
23       System.out.printf( "   %s\n", t2.toString() );
24
25       System.out.println(
26          "t3: hour and minute specified; second defaulted" );
27       System.out.printf( "   %s\n", t3.toUniversalString() );
28       System.out.printf( "   %s\n", t3.toString() );
29
30       System.out.println( "t4: hour, minute and second specified" );
31       System.out.printf( "   %s\n", t4.toUniversalString() );
32       System.out.printf( "   %s\n", t4.toString() );
33
34       System.out.println( "t5: all invalid values specified" );
35       System.out.printf( "   %s\n", t5.toUniversalString() );
36       System.out.printf( "   %s\n", t5.toString() );
37
38       System.out.println( "t6: Time2 object t4 specified" );
39       System.out.printf( "   %s\n", t6.toUniversalString() );
40       System.out.printf( "   %s\n", t6.toString() );
41    } // fim de main
42 } // fim da classe Time2Test
```

```
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: all invalid values specified
   00:00:00
   12:00:00 AM
t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

Figura 8.6 | Construtores sobrecarregados utilizados para inicializar objetos `Time2`.

8.6 Construtores padrão e sem argumentos

Cada classe deve ter pelo menos um construtor. Se você não fornecer nenhum construtor na declaração de uma classe, o compilador cria um construtor padrão que não aceita argumentos ao ser invocado. O construtor padrão inicializa as variáveis de instância com os valores iniciais especificados nas suas declarações ou com seus valores padrão (zero para tipos numéricos primitivos, `false` para valores `boolean` e `null` para referências). Na Seção 9.4.1, você aprenderá que o construtor padrão também realiza outra tarefa.

Se a sua classe declarar construtores, o compilador não criará um construtor padrão. Nesse caso, você deve declarar um construtor sem argumentos se a inicialização padrão for requerida. Como ocorre com um construtor padrão, um construtor sem argumentos é invocado com parênteses vazios. Observe que o construtor `Time2` sem argumentos (as linhas 12–15 da Figura 8.5) inicializa explicitamente um objeto `Time2` passando ao construtor de três argumentos 0 para cada parâmetro. Uma vez que 0 é o valor padrão para variáveis de instância `int`, nesse exemplo o construtor sem argumentos na verdade poderia ser declarado com um corpo vazio. Nesse caso, cada variável de instância receberia seu valor padrão quando o construtor sem argumentos fosse chamado. Se omitíssemos o construtor sem argumentos, clientes dessa classe não seriam capazes de criar um objeto `Time2` com a expressão `new Time2()`.

Erro comum de programação 8.5

Um erro de compilação ocorre se um programa tentar inicializar um objeto de uma classe passando o número incorreto ou tipos de argumentos para o construtor da classe.

Observação de engenharia de software 8.5

O Java permite que outros métodos da classe além dos seus construtores tenham o mesmo nome da classe e especifiquem tipos de retorno. Esses métodos não são construtores e não serão chamados quando um objeto da classe for instanciado. O Java determina quais métodos são construtores localizando aqueles que têm o mesmo nome da classe e que não especificam um tipo de retorno.

8.7 Notas sobre os métodos *set* e *get*

Como você sabe, os campos `private` de uma classe podem ser manipulados somente por seus métodos. Uma manipulação típica talvez seja o ajuste do saldo bancário de um cliente (por exemplo, uma variável de instância `private` de uma classe `BankAccount`) por um método `computeInterest`. As classes costumam fornecer métodos `public` para permitir a clientes da classe *configurar* (*set*, isto é, atribuir valores a) ou *obter* (*get*, isto é, obter os valores de) variáveis de instância `private`.

Como um exemplo de atribuição de nome, geralmente um método que configura variável de instância `interestRate` seria chamado `setInterestRate` e em geral um método que obtém `interestRate` seria chamado `getInterestRate`. Os métodos *set* também são comumente chamados de **métodos modificadores**, porque em geral alteram o estado de um objeto — isto é, modificam os valores de variáveis de instância. Os métodos *get* também são comumente chamados de **métodos de acesso** ou **métodos de consulta**.

Métodos set *e* get versus *dados* `public`

Parece que fornecer as capacidades de *set* e *get* é essencialmente o mesmo que tornar `public` as variáveis de instância. Essa é uma das sutilezas que tornam o Java tão desejável para a engenharia de software. Uma variável de instância `public` pode ser lida ou gravada por qualquer método que tem uma referência que contém a variável. Se uma variável de instância for declarada `private`, um método *get* `public` certamente permitirá que outros métodos a acessem, mas o método *get* pode controlar como o cliente pode acessá-la. Por exemplo, um método *get* poderia controlar o formato dos dados que ele retorna e assim proteger o código do cliente na representação dos dados real. Um método `public` *set* pode — e deve — avaliar cuidadosamente as tentativas de modificar o valor da variável a fim de assegurar que o novo valor é consistente para esse item de dados. Por exemplo, uma tentativa de configurar (*set*) o dia do mês como 37 seria rejeitada, uma tentativa de configurar o peso de uma pessoa com um valor negativo seria rejeitada e assim por diante. Portanto, embora os métodos *set* e *get* possam fornecer acesso a dados `private`, o acesso é restrito pela implementação dos métodos. Isso ajuda a promover uma boa engenharia de software.

Teste de validade em métodos set

Os benefícios da integridade de dados não seguem automaticamente simplesmente porque as variáveis de instância são declaradas `private` — você deve fornecer a verificação de validade. O Java permite-lhe projetar programas melhores de uma maneira conveniente. Métodos *set* de uma classe poderiam retornar valores indicando que foram feitas tentativas de atribuir dados inválidos a objetos da classe. Um cliente da classe poderia testar o valor de retorno de um método *set* para determinar se a tentativa do cliente de modificar o objeto foi bem-sucedida e tomar uma ação apropriada. Mas, em geral, os métodos *set* têm o tipo de retorno `void` e utilizam o tratamento de exceções para indicar tentativas de atribuir dados inválidos. Discutimos o tratamento de exceções em detalhes no Capítulo 11.

Observação de engenharia de software 8.6

Se apropriado, forneça métodos `public` para alterar e recuperar os valores de variáveis de instância `private`. Essa arquitetura ajuda a ocultar a implementação de uma classe dos seus clientes, o que aprimora a modificabilidade do programa.

Dica de prevenção de erro 8.2

Os métodos set *e* get *ajudam a criar classes que são mais fáceis de depurar e manter. Se apenas um método realizar uma tarefa particular, como configurar a hora em um objeto `Time2`, é mais fácil depurar e manter a classe. Se a `hour` não for configurada adequadamente, o código que na verdade modifica a variável de instância `hour` estará localizado no corpo do método — `setHour`. Portanto, seus esforços de depuração podem se concentrar no método `setHour`.*

Métodos predicados

Uma outra utilização comum para métodos de acesso é testar se uma condição é verdadeira ou falsa — esses métodos costumam ser chamados de **métodos predicados**. Um exemplo seria a classe método `isEmpty` da classe `ArrayList`, que retorna `true` se `ArrayList` for vazia. Um programa poderia testar `isEmpty` antes de tentar ler outro item de um `ArrayList`.

8.8 Composição

Uma classe pode ter referências a objetos de outras classes como membros. Isso é chamado **composição** e, às vezes, é referido como um **relacionamento *tem um***. Por exemplo, um objeto `AlarmClock` precisa saber a hora atual e a hora em que o alarme deve soar, portanto, é razoável incluir duas referências a objetos `Time` em um objeto `AlarmClock`.

Nosso exemplo de composição contém três classes — `Date` (Figura 8.7), `Employee` (Figura 8.8) e `EmployeeTest` (Figura 8.9). `Date` (Figura 8.7) declara as variáveis de instância `month`, `day` e `year` (linhas 6–8) para representar uma data. O construtor recebe três parâmetros `int`. A linha 14 invoca o método utilitário `checkMonth` (linhas 23–33) para validar o mês — um valor fora do intervalo é configurado como `1` para manter um estado consistente. A linha 15 supõe que o valor para `year` esteja correto e não o valida. A linha 16 invoca o método utilitário `checkDay` (linhas 36–52) para validar o valor para `day` com base no `month` e `year` atual. As linhas 42–43 determinam se o dia é correto com base no número de dias no `month` particular. Se o dia não estiver correto, as linhas 46–47 determinam se o `month` é fevereiro, o dia é `29` e o `year` é um ano bissexto. Se as linhas 42–48 não retornarem um valor correto para `day`, a linha 51 retorna `1` para manter `Date` em um estado consistente. Observe que as linhas 18–19 no construtor geram a saída da referência `this` como uma `String`. Como `this` é uma referência ao objeto `Date` atual, o método `toString` do objeto (linhas 55–58) é chamado implicitamente para obter a representação `String` do objeto.

```
1  // Figura 8.7: Date.java
2  // Declaração da classe Date.
3
4  public class Date
5  {
```

```
6       private int month; // 1-12
7       private int day; // 1-31 conforme o mês
8       private int year; // qualquer ano
9
10      // construtor: chama checkMonth para confirmar o valor adequado para month;
11      // chama checkDay para confirmar o valor adequado para day
12      public Date( int theMonth, int theDay, int theYear )
13      {
14         month = checkMonth( theMonth ); // valida month
15         year = theYear; // poderia validar year
16         day = checkDay( theDay ); // valida day
17
18         System.out.printf(
19            "Date object constructor for date %s\n", this );
20      } // fim do construtor Date
21
22      // método utilitário para confirmar o valor adequado de month
23      private int checkMonth( int testMonth )
24      {
25         if ( testMonth > 0 && testMonth <= 12 ) // valida month
26            return testMonth;
27         else // month é inválido
28         {
29            System.out.printf(
30               "Invalid month (%d) set to 1.", testMonth );
31            return 1; // mantém objeto em estado consistente
32         } // fim de else
33      } // fim do método checkMonth
34
35      // método utilitário para confirmar o valor adequado de day com base em month e year
36      private int checkDay( int testDay )
37      {
38         int[] daysPerMonth =
39            { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
40
41         // verifica se day está no intervalo para month
42         if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43            return testDay;
44
45         // verifica ano bissexto
46         if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
47              ( year % 4 == 0 && year % 100 != 0 ) ) )
48            return testDay;
49
50         System.out.printf( "Invalid day (%d) set to 1.", testDay );
51         return 1;  // mantém objeto em estado consistente
52      } // fim do método checkDay
53
54      // retorna uma String no formato mês/dia/ano
55      public String toString()
56      {
57         return String.format( "%d/%d/%d", month, day, year );
58      } // fim do método toString
59   } // fim da classe Date
```

Figura 8.7 | Declaração da classe `data`.

A classe `Employee` (Figura 8.8) tem variáveis de instância `firstName`, `lastName`, `birthDate` e `hireDate`. Os membros `firstName` e `lastName` (linhas 6–7) são referências a objetos `String`. Os membros `birthDate` e `hireDate` (linhas 8–9) são referências a objetos `Date`. Isso demonstra que uma classe pode conter como variáveis de instância referências a objetos de outras classes. O construtor `Employee` (linhas 12–19) aceita quatro parâmetros — `first`, `last`, `dateOfBirth` e `dateOfHire`. As variáveis de instância do objeto `Employee` são atribuídas aos objetos referenciados pelos parâmetros. Observe que quando o método `toString` da classe `Employee` é chamado, ele retorna uma `String` contendo o nome do empregado e as representações de `String` dos dois objetos `Date`. Cada uma dessas `Strings` é obtida com uma chamada implícita ao método `toString` da classe `Date`.

```
1  // Figura 8.8: Employee.java
2  // Classe Employee com referência a outros objetos.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // construtor para inicializar nome, data de nascimento e
      // data de contratação
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // fim do construtor Employee
20
21    // converte Employee em formato de String
22    public String toString()
23    {
24       return String.format( "%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate );
26    } // fim do método toString
27 } // fim da classe Employee
```

Figura 8.8 | A classe `Employee` com referência a outros objetos.

A classe `EmployeeTest` (Figura 8.9) cria dois objetos `Date` (linhas 8–9) para representar o aniversário e a data de contratação, respectivamente, de um `Employee`. A linha 10 cria um `Employee` e inicializa suas variáveis de instância passando para o construtor duas `Strings` (representando o primeiro e último nomes do `Employee`) e dois objetos `Date` (representando o aniversário e a data de contratação). A linha 12 invoca implicitamente o método `toString` de `Employee` para exibir os valores das suas variáveis de instância e demonstrar que o objeto foi inicializado adequadamente.

```
1  // Figura 8.9: EmployeeTest.java
2  // Demonstração de composição.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // fim de main
14 } // fim da classe EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Figura 8.9 | A demonstração de composição.

8.9 Enumerações

Na Figura 6.9 (`Craps.java`), apresentamos o tipo `enum` básico que define um conjunto de constantes representadas como identificadores únicos. Nesse programa, as constantes `enum` representaram o status do jogo. Nesta seção, discutimos o relacionamento entre tipos e classes `enum`. Como as classes, todos os tipos `enum` são tipos de referência. Um tipo `enum` é declarado com uma **declaração `enum`**, uma lista separada por vírgulas de constantes `enum` — a declaração pode opcionalmente incluir outros componentes das classes tradicionais como construtores, campos e métodos. Cada declaração `enum` declara uma classe `enum` com as seguintes restrições:

1. As constantes `enum` são implicitamente `final`, porque declaram constantes que não devem ser modificadas.
2. Constantes `enum` são implicitamente `static`.
3. Qualquer tentativa de criar um objeto de um tipo `enum` com um operador `new` resulta em um erro de compilação.

As constantes `enum` podem ser utilizadas em qualquer lugar em que constantes podem ser utilizadas, como nos rótulos `case` das instruções `switch` e para controlar instruções `for` aprimoradas.

A Figura 8.10 ilustra como declarar variáveis de instância, um construtor e métodos em um tipo `enum`. A declaração `enum` (linhas 5–37) contém duas partes — as constantes `enum` e os outros membros do tipo `enum`. A primeira parte (linhas 8–13) declara seis constantes `enum`. Cada uma delas é opcionalmente seguida por argumentos que são passados para o **construtor `enum`** (linhas 20–24). Como os construtores que você viu nas classes, um construtor `enum` pode especificar qualquer número de parâmetros e pode ser sobrecarregado. Neste exemplo, o construtor `enum` tem dois parâmetros `String`, consequentemente cada constante `enum` é seguida por parênteses contendo dois argumentos `String`. A segunda parte (linhas 16–36) declara os outros membros do tipo `enum` — duas variáveis de instância (linhas 16–17), um construtor (linhas 20–24) e dois métodos (linhas 27–30 e 33–36).

```
1  // Figura 8.10: Book.java
2  // Declarando um tipo enum com um construtor e campos de
3  // instância explícitos e métodos de acesso para esses campos
4
5  public enum Book
6  {
7     // declara constantes do tipo enum
8     JHTP( "Java How to Program", "2010" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2008" ),
12    VBHTP( "Visual Basic 2008 How to Program", "2009" ),
13    CSHARPHTP( "Visual C# 2008 How to Program", "2009" );
14
15    // campos de instância
16    private final String title; // título de livro
17    private final String copyrightYear; // ano dos direitos autorais
18
19    // construtor enum
20    Book( String bookTitle, String year )
21    {
22       title = bookTitle;
23       copyrightYear = year;
24    } // fim do construtor enum Book
25
26    // método de acesso para título de campo
27    public String getTitle()
28    {
29       return title;
30    } // fim do método getTitle
31
32    // método de acesso para o campo copyrightYear
33    public String getCopyrightYear()
34    {
35       return copyrightYear;
36    } // fim do método getCopyrightYear
37 } // fim do enum Book
```

Figura 8.10 | Declarando um tipo `enum` com construtor e campos de instância explícitos e os métodos de acesso desses campos.

As linhas 16–17 declaram as variáveis de instância `title` e `copyrightYear`. Cada constante `enum` em `Book` é na verdade um objeto do tipo `Book` que tem sua própria cópia das variáveis de instância `title` e `copyrightYear`. O construtor (linhas 20–24) recebe dois parâmetros

String, um que especifica o título do livro e outro que especifica o ano dos direitos autorais. As linhas 22–23 atribuem esses parâmetros às variáveis de instância. As linhas 27–36 declaram dois métodos, que retornam o título de livro e o ano dos direitos autorais, respectivamente.

A Figura 8.11 testa o tipo enum Book e ilustra como iterar por um intervalo de constantes enum. Para cada enum, o compilador gera um método static chamado **values** (chamado na linha 12) que retorna um array das constantes do enum na ordem em que elas foram declaradas. As linhas 12–14 utilizam a instrução for aprimorada para exibir todas as constantes declaradas em enum Book. A linha 14 invoca os métodos getTitle e getCopyrightYear de enum Book para obter o título e o ano dos direitos autorais associado com a constante. Observe que quando uma constante enum é convertida em uma String (por exemplo, book na linha 13), o identificador da constante é utilizado como a representação de String (por exemplo, JHTP para a primeira constante enum).

```
1  // Figura 8.11: EnumTest.java
2  // testando o tipo enum Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "All books:\n" );
10
11       // imprime todos os livros em enum Book
12       for (Book book : Book.values() )
13          System.out.printf( "%-10s%-45s%s\n", book,
14             book.getTitle(),book.getCopyrightYear() );
15
16       System.out.println( "\nDisplay a range of enum constants:\n" );
17
18       // imprime os primeiros quatro livros
19       for ( Book book : EnumSet.range( Book.JHTP, Book.CPPHTP ) )
20          System.out.printf( "%-10s%-45s%s\n", book,
21             book.getTitle(),book.getCopyrightYear() );
22    } // fim de main
23 } // fim da classe EnumTest
```

```
All books:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
VBHTP     Visual Basic 2008 How to Program              2009
CSHARPHTP Visual C# 2008 How to Program                 2009

Display a range of enum constants:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
```

Figura 8.11 | Testando um tipo enum.

As linhas 19–21 utilizam o método static **range** da classe **EnumSet** (declarado no pacote java.util) para exibir um intervalo das constantes do enum Book. O método range leva dois parâmetros — a primeira e a última constantes enum no intervalo — e retorna um EnumSet que contém todas as constantes entre essas duas constantes, inclusive. Por exemplo, a expressão EnumSet.range(Book.JHTP, Book.CPPHTP) retorna um EnumSet contendo Book.JHTP, Book.CHTP, Book.IW3HTP e Book.CPPHTP. A instrução for aprimorada pode ser utilizada com um EnumSet, assim como com um array, portanto as linhas 12–14 utilizam-na para exibir o título e o ano dos direitos autorais de cada livro na EnumSet. A classe EnumSet fornece vários outros métodos static para criar conjuntos de constantes enum do mesmo tipo enum. Para obter mais informações sobre a classe EnumSet, visite java.sun.com/javase/6/docs/api/java/util/EnumSet.html.

Erro comum de programação 8.6

Em uma declaração enum, é um erro de sintaxe declarar constantes enum depois dos construtores campos e métodos do tipo enum.

8.10 Coleta de lixo e o método finalize

Toda classe no Java tem os métodos da classe `Object` (pacote `java.lang`), um dos quais é o método `finalize`. Esse método é raramente utilizado porque pode causar problemas de desempenho e há uma incerteza sobre se ele será chamado. No entanto, como `finalize` faz parte de cada classe, nós o discutimos aqui para ajudá-lo a entender o objetivo desse método. Os detalhes completos sobre o método `finalize` estão além do escopo deste livro e a maioria dos programadores não deve utilizá-lo — logo você verá por quê. Você aprenderá mais sobre a classe `Object` no Capítulo 9.

Todo objeto utiliza recursos do sistema, como memória. Precisamos de uma maneira disciplinada de retornar recursos ao sistema quando eles não são mais necessários; caso contrário, poderiam ocorrer "perdas de recursos" que impediriam sua reutilização pelo seu programa ou possivelmente por outros programas. A JVM realiza a **coleta de lixo** automática para reivindicar a memória ocupada por objetos que não são mais utilizados. Quando não houver mais referências a um objeto, o objeto está apto a ser coletado. Em geral, isso ocorre quando a JVM executa seu **coletor de lixo**. Assim, vazamentos de memória que são comuns em outras linguagens como C e C++ (porque a memória não é automaticamente reivindicada nessas linguagens) são menos prováveis em Java, mas alguns ainda podem acontecer de maneiras sutis. Outros tipos de vazamentos de recursos podem ocorrer. Por exemplo, um aplicativo pode abrir um arquivo no disco para modificar os seus conteúdos. Se ele não fechar o arquivo, o aplicativo deve terminar antes de qualquer outro aplicativo poder utilizá-lo.

O **método `finalize`** é chamado pelo coletor de lixo para realizar **limpeza de terminação** sobre um objeto um pouco antes de o coletor de lixo reivindicar a memória do objeto. O método `finalize` não aceita parâmetros e tem o tipo de retorno `void`. Um problema com relação ao método `finalize` é que não há garantias de o coletor de lixo executar em uma hora especificada. De fato, o coletor de lixo nunca pode executar antes de um programa terminar. Portanto, não fica claro se, ou quando, o método `finalize` será chamado. Por essa razão, a maioria dos programadores deve evitar o método `finalize`.

Observação de engenharia de software 8.7

Uma classe que usa recursos de sistema, como arquivos no disco, deve fornecer um método que os programadores podem chamar para liberar recursos quando não forem mais necessários em um programa. Muitas classes da Java API fornecem métodos `close` ou `dispose` para esse propósito. Por exemplo, a classe `Scanner` (`java.sun.com/javase/6/docs/api/java/util/Scanner.html`) tem um método `close`.

8.11 Membros da classe static

Cada objeto tem sua própria cópia de todas as variáveis de instância da classe. Em certos casos, apenas uma cópia de uma variável particular deve ser compartilhada por todos os objetos de uma classe. Um **campo `static`** — chamado de **variável de classe** — é utilizado nesses casos. Uma variável `static` representa **informações de escopo de classe** — todos os objetos da classe compartilham os mesmos dados. A declaração de uma variável `static` inicia com a palavra-chave `static`.

Vamos motivar a necessidade de dados `static` com um exemplo. Suponha que tivéssemos um videogame com `Martians` e outras criaturas de espaço. Cada `Martian` tende a ser corajoso e disposto a atacar outras criaturas espaciais quando o `Martian` está ciente de que pelo menos cinco `Martians` estão presentes. Se menos de cinco `Martians` estiverem presentes, cada um deles torna-se covarde. Assim, cada `Martian` precisa conhecer o `martianCount`. Poderíamos dotar a classe `Martian` com `martianCount` como uma variável de instância. Se fizermos isso, cada `Martian` terá uma cópia separada da variável de instância, e toda vez que criarmos um novo `Martian`, teremos de atualizar a variável de instância `martianCount` em cada objeto `Martian`. Isso desperdiça espaço com as cópias redundantes, desperdiça tempo com a atualização das cópias separadas e é propenso a erros. Em vez disso, declaramos `martianCount` como `static`, tornando `martianCount` dados de escopo de classe. Cada `Martian` pode ver a `martianCount` como se fosse uma variável de instância da classe `Martian`, mas apenas uma cópia da `static martianCount` é mantida. Isso economiza espaço. Poupamos tempo fazendo com que o construtor `Martian` incremente o `static martianCount` — há somente uma cópia, assim não temos de incrementar cópias separadas para cada objeto `Martian`.

Observação de engenharia de software 8.8

Use uma variável `static` quando todos os objetos de uma classe precisarem utilizar a mesma cópia da variável.

Variáveis estáticas têm escopo de classe. Podemos acessar membros `public static` de uma classe por meio de uma referência a qualquer objeto da classe ou qualificando o nome de membro com o nome de classe e um ponto (`.`), como em `Math.random()`. Membros da classe `private static` de uma classe podem ser acessados pelo código do cliente somente por métodos da classe. Realmente, os membros da classe `static` existem mesmo quando nenhum objeto da classe existe — eles estão disponíveis logo que a classe é carregada na memória em tempo de execução. Para acessar um membro `public static` quando não há nenhum objeto da classe (e mesmo se houver), prefixe o nome da classe e acrescente um ponto (`.`) ao membro `static`, como em `Math.PI`. Para acessar um membro `private static` quando não existir objetos da classe, forneça um método `public static` e chame-o qualificando seu nome com o nome de classe e um ponto.

Observação de engenharia de software 8.9

Variáveis e métodos de classe `static` existem e podem ser utilizados, mesmo se nenhum objeto dessa classe tiver sido instanciado.

Um método `static` não pode acessar membros de classe não `static`, porque um método `static` pode ser chamado mesmo quando nenhum objeto da classe foi instanciado. Pela mesma razão, a referência `this` não pode ser utilizada em um método `static`. A referência `this` deve referenciar um objeto específico da classe, e quando um método `static` for chamado, poderia não haver nenhum objeto de sua classe na memória.

Erro comum de programação 8.7

Um erro de compilação ocorre se um método `static` chamar um método de instância (não `static`) na mesma classe utilizando apenas o nome do método. De modo semelhante, um erro de compilação ocorre se um método `static` tentar acessar uma variável de instância na mesma classe utilizando apenas o nome da variável.

Erro comum de programação 8.8

Referenciar `this` em um método `static` é considerado um erro de sintaxe.

Monitorando o número de objetos de empregado que foram criados

Nosso próximo programa declara duas classes — `Employee` (Figura 8.12) e `EmployeeTest` (Figura 8.13). A classe `Employee` declara a variável `private static count` (Figura 8.12, linha 9) e o método `public static getCount` (linhas 36–39). A variável `count static` é inicializada como zero na linha 9. Se uma variável `static` não for inicializada, o compilador atribuirá um valor padrão — nesse caso 0, o valor padrão para o tipo `int`. A variável `count` mantém uma conta do número de objetos da classe `Employee` que foram criados até agora.

```
1  // Figura 8.12: Employee.java
2  // Variável estática utilizada para manter uma contagem do número
3  // de objetos Employee na memória.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // o número de Employees criados
10
11    // inicializa Employee, adiciona 1 a static count e
12    // gera a saída de String indicando que o construtor foi chamado
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       ++count;  // incrementa contagem estática de empregados
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // fim do construtor Employee
22
23    // obtém o primeiro nome
24    public String getFirstName()
25    {
26       return firstName;
27    } // fim do método getFirstName
28
29    // obtém o último nome
30    public String getLastName()
31    {
32       return lastName;
33    } // fim do método getLastName
34
35    // método estático para obter valor de contagem estática
36    public static int getCount()
37    {
38       return count;
39    } // fim do método getCount
40 } // fim da classe Employee
```

Figura 8.12 | Variável `static` utilizada para manter uma contagem do número de objetos `Employee` na memória.

Quando houver objetos `Employee`, a variável `count` pode ser utilizada em qualquer método de um objeto `Employee` — esse exemplo incrementa `count` no construtor (linha 18). O método `public static getCount` (linhas 36–39) retorna o número de objetos `Employee` que foram criados até agora. Quando não houver objetos da classe `Employee`, o código de cliente pode acessar a variável `count` chamando o método `getCount` via nome de classe, como em `Employee.getCount()`. Quando existirem objetos, o método `getCount` também pode ser chamado via qualquer referência a um objeto `Employee`.

Boa prática de programação 8.1

Invoque cada método `static` usando o nome de classe e um ponto (`.`) para enfatizar que o método sendo chamado é um método `static`.

O método `EmployeeTest main` (Figura 8.13) instancia dois objetos `Employee` (linhas 13–14). Quando cada construtor do objeto `Employee` é invocado, as linhas 15–16 da Figura 8.12 atribuem o nome e o sobrenome de `Employee` às variáveis de instância `firstName` e `lastName`. Note que essas duas instruções não fazem cópias dos argumentos `String` originais. Na verdade, objetos `String` em Java são **imutáveis** — eles não podem ser modificados depois de criados. Portanto, é seguro ter muitas referências a um objeto `String`. Em geral, esse não é o caso para os objetos da maioria das outras classes em Java. Se objetos `String` são imutáveis, talvez você se pergunte por que podemos usar operadores + e += para concatenar objetos `String`. Operações de concatenação de strings resultam em um novo objeto `String` contendo o valor concatenado. Os objetos `String` originais não são modificados.

```
1  // Figura 8.13: EmployeeTest.java
2  // Demonstração do membro static.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // mostra que a contagem é 0 antes de criar Employees
9        System.out.printf( "Employees before instantiation: %d\n",
10          Employee.getCount() );
11
12       // cria dois Employees; a contagem deve ser 2
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
16       // mostra que a contagem é 2 depois de criar dois Employees
17       System.out.println( "\nEmployees after instantiation: " );
18       System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19       System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20       System.out.printf( "via Employee.getCount(): %d\n",
21          Employee.getCount() );
22
23       // obtém nomes de Employees
24       System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25          e1.getFirstName(), e1.getLastName(),
26          e2.getFirstName(), e2.getLastName() );
27
28       // nesse exemplo, há somente uma referência a cada Employee,
29       // portanto as duas instruções seguintes indicam que
30       // esses objetos são elegíveis para a coleta de lixo
31       e1 = null;
32       e2 = null;
33    } // fim de main
34 } // fim da classe EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Figura 8.13 | A demonstração do membro `static`.

Quando `main` termina de usar os dois objetos `Employee`, as referências `e1` e `e2` são configuradas como `null` nas linhas 31–32. Nesse ponto, as referências `e1` e `e2` não mais referenciam os objetos que foram instanciados nas linhas 13–14. Os objetos tornam-se "elegíveis para a coleta de lixo" porque não há mais referências a eles no programa.

Por fim, o coletor de lixo talvez reivindique a memória para esse objeto (ou o sistema operacional reivindicará a memória quando o programa terminar). A JVM não garante quando, ou até se, o coletor de lixo executará. Quando o coletor de lixo realmente executar, é possível que nenhum objeto ou apenas um subconjunto de objetos elegíveis seja coletado.

8.12 Importação `static`

Na Seção 6.3, você aprendeu sobre os campos e métodos `static` da classe `Math`. Invocamos campos e métodos `static` da classe `Math` precedendo cada um com o nome da classe `Math` e um ponto (`.`). Uma declaração **`static import`** permite que você importe membros `static` de uma classe ou interface para que possa acessá-los via os nomes não qualificados dos membros na sua classe — o nome de classe e um ponto (`.`) não devem utilizar um membro `static` importado.

Uma declaração de importação `static` tem duas formas — uma que importa um membro `static` particular (conhecido como **importação `static` simples**) e outra que importa todos os membros `static` de uma classe (conhecido como **importação `static` sob demanda**). A sintaxe a seguir importa um membro `static` particular:

```
import static nomeDoPacote.NomeDaClasse.nomeDoMembroStatic;
```

onde *nomeDoPacote* é o pacote da classe (por exemplo, `java.lang`), *NomeDaClasse* é o nome da classe (por exemplo, `Math`) e *nomeDoMembroStatic* é o nome do campo ou método `static` (por exemplo, `PI` ou `abs`). A sintaxe a seguir importa todos os membros `static` de uma classe:

```
import static nomeDoPacote.NomeDaClasse.*;
```

onde *nomeDoPacote* é o pacote da classe (por exemplo, `java.lang`) e *NomeDaClasse* é o nome da classe (por exemplo, `Math`). O asterisco (`*`) indica que *todos* os membros `static` da classe especificada devem estar disponíveis para utilização na(s) classe(s) declarada(s) no arquivo. Observe que declarações de importação `static` importam somente membros de classes `static`. Instruções `import` regulares devem ser utilizadas para especificar as classes utilizadas em um programa.

A Figura 8.14 demonstra uma import `static`. A linha 3 é uma declaração de import `static` que importa todos os campos métodos `static` da classe `Math` no pacote `java.lang`. As linhas 9–12 acessam o campo `static` `E` da classe `Math` (linha 11) e os métodos `static` `sqrt` (linha 9), `ceil` (linha 10), `log` (linha 11) e `cos` (linha 12) sem preceder o nome do campo ou nomes dos métodos com o nome da classe `Math` e um ponto.

```
1  // Figura 8.14: StaticImportTest.java
2  // Importação estática dos métodos da classe Math.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String[] args )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // fim de main
14 } // fim da classe StaticImportTest
```

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Figura 8.14 | Importação estática dos métodos da classe `Math`.

Erro comum de programação 8.9

Um erro de compilação ocorre se um programa tentar importar métodos `static` que têm a mesma assinatura ou campos `static` que têm o mesmo nome proveniente de duas ou mais classes.

8.13 Variáveis de instância final

O **princípio do menor privilégio** é fundamental para uma boa engenharia de software. No contexto de um aplicativo, esse princípio afirma que deve ser concedido ao código somente a quantidade de privilégio e acesso de que ele precisa para realizar sua tarefa designada, mas não mais que isso. Isso torna os seus programas mais robustos evitando que o código modifique acidentalmente (ou maliciosamente) valores de variáveis e chame métodos que não devem ser acessíveis.

Vejamos como esse princípio se aplica a variáveis de instância. Algumas delas precisam ser modificáveis e algumas não. Você pode utilizar a palavra-chave `final` para especificar o fato de que uma variável não é modificável (isto é, é uma constante) e que qualquer tentativa de modificá-la é um erro. Por exemplo,

```
private final int INCREMENT;
```

declara uma variável de instância `final` `INCREMENT` (constante) do tipo `int`. Embora essas constantes possam ser inicializadas quando são declaradas, isso não é exigido. Elas podem ser inicializadas por cada um dos construtores da classe para que cada objeto da classe tenha um valor diferente.

Observação de engenharia de software 8.10

Declarar uma variável de instância como `final` ajuda a impor o princípio do menor privilégio. Se uma variável de instância não deve ser modificada, declare-a como sendo `final` para evitar modificação.

Nosso próximo exemplo contém duas classes — a classe `Increment` (Figura 8.15) e a `IncrementTest` (Figura 8.16). A classe `Increment` contém uma variável de instância `final int` chamada `INCREMENT` (Figura 8.15, linha 7). Observe que `INCREMENT` não é inicializada na sua declaração, dessa forma ela deve ser inicializada pelo construtor da classe (linhas 10–13). Se a classe fornecesse múltiplos construtores, cada um deles teria de inicializar a variável `final`. O construtor recebe um parâmetro `int incrementValue` e atribui seu valor a `INCREMENT` (linha 12). Uma variável `final` não pode ser modificada por atribuição depois que ela é inicializada. A classe de aplicativo `IncrementTest` cria um objeto da classe `Increment` (Figura 8.16, linha 8) e fornece como argumento ao construtor o valor `5` a ser atribuído à constante `INCREMENT`.

```
1  // Figura 8.15: Increment.java
2  // Variável de instância final em uma classe.
3
4  public class Increment
5  {
6     private int total = 0; // total de todos os incrementos
7     private final int INCREMENT; // variável constante (não inicializada)
8
9     // construtor inicializa variável de instância final INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // inicializa variável constante (uma vez)
13    } // fim do construtor Increment
14
15    // adiciona INCREMENT ao total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // fim do método addIncrementToTotal
20
21    // retorna representação de String dos dados de um objeto Increment
22    public String toString()
23    {
24       return String.format( "total = %d", total );
25    } // fim do método toString
26 } // fim da classe Increment
```

Figura 8.15 | Variável de instância `final` em uma classe.

```
1  // Figura 8.16: IncrementTest.java
2  // Variável final inicializada com um argumento de construtor.
3
4  public class IncrementTest
5  {
6     public static void main( String[] args )
7     {
8        Increment value = new Increment( 5 );
9
10       System.out.printf( "Before incrementing: %s\n\n", value );
11
12       for ( int i = 1; i <= 3; i++ )
13       {
14          value.addIncrementToTotal();
15          System.out.printf( "After increment %d: %s\n", i, value );
16       } // for final
17    } // fim de main
18 } // fim da classe IncrementTest
```

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Figura 8.16 | Variável `final` inicializada com um argumento de construtor.

Erro comum de programação 8.10

Tentar modificar uma variável de instância `final` depois que ela é inicializada é um erro de compilação.

Dica de prevenção de erro 8.3

Tentativas de modificar uma variável de instância `final` são capturadas em tempo de compilação em vez de causarem erros em tempo de execução. Sempre é preferível retirar bugs em tempo de compilação, se possível, em vez de permitir que passem para o tempo de execução (onde estudos descobriram que o reparo é frequentemente muito mais caro).

Observação de engenharia de software 8.11

Um campo `final` também deve ser declarado `static` se ele for inicializado em sua declaração com um valor que é o mesmo de todos os objetos da classe. Depois dessa inicialização, seu valor não pode nunca mudar. Portanto, não precisamos de uma cópia separada do campo para cada objeto da classe. Criar o campo `static` permite que todos os objetos da classe compartilhem o campo `final`.

Se uma variável `final` não for inicializada, ocorrerá um erro de compilação. Para demonstrar isso, colocamos a linha 12 da Figura 8.15 em um comentário e recompilamos a classe. A Figura 8.17 mostra a mensagem de erro produzida pelo compilador.

```
Increment.java:13: variable INCREMENT might not have been initialized
   } // fim do construtor de Increment
   ^
1 error
```

Figura 8.17 | A variável `final` `INCREMENT` deve ser inicializada.

Erro comum de programação 8.11

Não inicializar uma variável de instância `final` na sua declaração ou em cada construtor da classe produz um erro de compilação indicando que a variável talvez não tenha sido inicializada.

8.14 Estudo de caso da classe `Time`: criando pacotes

Vimos em quase todos os exemplos no texto que classes em bibliotecas preexistentes, como a Java API, podem ser importadas para um programa Java. Toda classe na Java API pertence a um pacote que contém um grupo de classes relacionadas. Esses pacotes são definidos uma vez, mas podem ser importados em muitos programas. À medida que os aplicativos tornam-se mais complexos, os pacotes ajudam os programadores a gerenciar a complexidade dos componentes de aplicativo. Os pacotes também facilitam a reutilização de software permitindo que programas importem classes de outros pacotes (como fizemos na maioria dos exemplos), antes de copiar as classes em cada programa que os utiliza. Outro benefício dos pacotes é que eles fornecem uma convenção para nomes únicos de classes, o que ajuda a evitar conflitos de nomes de classe (discutido mais adiante nesta seção). Esta seção introduz como criar seus próprios pacotes.

Passos para declarar uma classe reutilizável

Antes que uma classe possa ser importada para múltiplos aplicativos, ela deve ser colocada em um pacote a fim de torná-la reutilizável. A Figura 8.18 mostra como especificar o pacote em que uma classe deve ser colocada. A Figura 8.19 mostra como importar nossa classe empacotada de modo que ela possa ser utilizada em um aplicativo. Os passos para criar uma classe reutilizável são:

1. Declare uma classe `public`. Se a classe não for `public`, ela pode ser utilizada somente por outras classes no mesmo pacote.
2. Escolha um nome de pacote e adicione uma **declaração `package`** ao arquivo de código-fonte para a declaração de classe reutilizável. Em cada arquivo de código-fonte do Java pode haver apenas uma declaração `package`, e ela deve preceder todas as outras declarações e instruções. Observe que comentários não são instruções; assim, eles podem ser colocados antes de uma instrução `package` em um arquivo. [*Nota:* se nenhuma instrução `package` for fornecida, a classe é colocada no chamado pacote padrão e é acessível somente a outras classes no pacote padrão localizadas no mesmo diretório. Todos os programas anteriores neste livro que tinham duas ou mais classes utilizaram esse pacote padrão.]
3. Compile a classe de modo que ela seja colocada no diretório de pacotes apropriado.
4. Importe a classe reutilizável para um programa e use a classe.

```
1  // Figura 8.18: Time1.java
2  // Declaração de classe Time1 mantém a hora no formato de 24 horas.
3  package com.deitel.jhtp.ch08;
4
5  public class Time1
6  {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // configura um novo valor de hora usando a hora universal;
12    // assegura que os dados permaneçam consistentes configurando valores inválidos como zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida horas
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida minutos
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida segundos
18    } // fim do método setTime
19
20    // converte em String no formato de hora universal (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // fim do método do toUniversalString
25
26    // converte em String no formato padrão de hora (H:MM:SS AM ou PM)
27    public String toString()
28    {
29       return String.format( "%d:%02d:%02d %s",
30          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31          minute, second, ( hour < 12 ? "AM" : "PM" ) );
32    } // fim do método toString
33 } // fim da classe Time1
```

Figura 8.18 | Empacotando a classe `Time1` para reutilização.

Passos 1 e 2: criando uma classe public e adicionando a instrução package

Para o *Passo 1*, modificamos a classe `public` `Time1` declarado na Figura 8.1. A nova versão é mostrada na Figura 8.18. Nenhuma modificação foi feita na implementação da classe, portanto não discutiremos os detalhes de implementação novamente aqui.

Para o *Passo 2*, adicionamos uma declaração `package` (linha 3) que declara um `package` chamado `com.deitel.jhtp.ch08`. Colocar uma declaração `package` no início de um arquivo-fonte Java indica que a classe declarada no arquivo é parte do pacote especificado. Somente declarações `package`, declarações `import` e comentários podem aparecer fora das chaves de uma declaração de classe. Um arquivo de código-fonte Java deve ter a seguinte ordem:

1. uma declaração `package` (se houver alguma),
2. declarações `import` (se houver alguma) e então
3. declarações de classe.

Somente uma das declarações de classe em um arquivo particular pode ser `public`. Outras classes no arquivo são colocadas no pacote e podem ser utilizadas somente pelas outras classes no pacote. Classes não `public` estão em um pacote para suportar as classes reutilizáveis no pacote.

Em um esforço para fornecer nomes únicos a cada pacote, a Sun Microsystems especifica uma convenção que deveria ser seguida Todo nome de pacote deve iniciar com seu nome de domínio Internet na ordem inversa. Por exemplo, nosso nome de domínio é `deitel.com`, assim nossos nomes de pacotes iniciam com `com.deitel`. Para o nome de domínio *suafaculdade*`.edu`, o nome de pacote deve iniciar com `edu.`*suafaculdade*. Depois que o nome de domínio é invertido, você pode escolher qualquer outro nome para seu pacote. Se você faz parte de uma empresa com muitas divisões ou uma universidade com muitas faculdades, você pode querer utilizar o nome de sua divisão ou faculdade como o próximo nome no pacote. Escolhemos utilizar `jhtp` como o próximo nome em nosso nome de pacote para indicar que essa classe é do livro *Java, como programar*. O sobrenome em nosso nome de pacote especifica que esse pacote é para o Capítulo 8 (`ch08`).

Passos 3: compilando a classe empacotada

O *Passo 3* é compilar a classe de modo que ela seja armazenada no pacote apropriado. Quando um arquivo Java contendo uma declaração `package` é compilado, o arquivo de classe resultante é colocado no diretório especificado pela declaração. A declaração `package` na Figura 8.18 indica que a classe `Time1` deve ser colocada no diretório

```
com
   deitel
      jhtp
         ch08
```

Os nomes de diretório na declaração `package` especificam a posição exata das classes no pacote.

Ao compilar uma classe em um pacote, a opção de linha de comando `javac` **`-d`** faz com que o compilador crie diretórios apropriados com base na declaração `package` da classe. A opção também especifica onde os diretórios devem ser armazenados. Por exemplo, em uma janela de comando, utilizamos o comando de compilação

```
javac -d . Time1.java
```

para especificar que o primeiro diretório no nosso nome de pacote deve ser colocado no diretório atual. O ponto (`.`) depois de `-d` no comando anterior representa o diretório atual nos sistemas operacionais Windows, Unix, Linux e Mac OS X (e em vários outros também). Depois da execução do comando de compilação, o diretório atual contém um diretório chamado `com`, `com` contém um diretório chamado `deitel`, `deitel` contém um diretório chamado `jhtp` e `jhtp` contém um diretório chamado `ch08`. No diretório `ch08`, você pode localizar o arquivo `Time1.class`. [*Nota:* se não utilizar a opção `-d`, você deverá copiar ou mover o arquivo de classe para o diretório apropriado de pacote depois que compilá-lo.]

O nome `package` faz parte do **nome de classe completamente qualificado**, portanto o nome da classe `Time1` é de fato `com.deitel.jhtp.ch08.Time1`. Você pode utilizar esse nome completamente qualificado nos seus programas ou pode `import`ar a classe e utilizar seu **nome simples** (o nome sozinho — `Time1`) no programa. Se outro pacote também contiver uma classe `Time1`, os nomes de classe completamente qualificados podem ser utilizados para distinguir entre as classes no programa e impedir um **conflito de nomes** (também chamado de **colisão de nomes**).

Passo 4: importando a classe reutilizável

Uma vez que a classe é compilada e armazenada em seu pacote, ela pode ser importada para programas (*Passo 4*). No aplicativo `Time1PackageTest` da Figura 8.19, a linha 3 especifica que a classe `Time1` deve ser importada para uso na classe `Time1PackageTest`. A classe `Time1PackageTest` está no pacote padrão porque o arquivo `.java` da classe não contém uma declaração `package`. Como as duas classes estão em pacotes diferentes, o `import` na linha 3 é necessário para que a classe `Time1PackageTest` possa utilizar a classe `Time1`.

```
1  // Figura 8.19: Time1PackageTest.java
2  // objeto Time1 utilizado em um aplicativo.
3  import com.deitel.jhtp.ch08.Time1; // importa a classe Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String[] args )
8     {
9        // cria e inicializa um objeto Time1
10       Time1 time = new Time1(); // chama o construtor Time1
11
12       // gera saída de representações de string da hora
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // gera saída de uma linha em branco
18
19       // altera a hora e gera saída da hora atualizada
20       time.setTime( 13, 27, 6 );
21       System.out.print( "Universal time after setTime is: " );
22       System.out.println( time.toUniversalString() );
23       System.out.print( "Standard time after setTime is: " );
24       System.out.println( time.toString() );
25       System.out.println(); // gera saída de uma linha em branco
26
27       // configura hora com valores inválidos; gera saída da hora atualizada
28       time.setTime( 99, 99, 99 );
29       System.out.println( "After attempting invalid settings:" );
30       System.out.print( "Universal time: " );
31       System.out.println( time.toUniversalString() );
32       System.out.print( "Standard time: " );
33       System.out.println( time.toString() );
34    } // fim de main
35 } // fim da classe Time1PackageTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Figura 8.19 | Objeto `Time1` utilizado em um aplicativo.

A linha 3 é conhecida como uma **declaração import de tipo simples** — isto é, a declaração `import` especifica uma classe a importar. Se seu programa utilizar múltiplas classes no mesmo pacote, você poderá importar essas classes com uma única declaração `import`. Por exemplo, a declaração `import`

```
import java.util.*; // importa classes do pacote java.util
```

utiliza um asterisco (*) no fim da declaração `import` para informar o compilador de que todas as classes `public` no pacote `java.util` estão disponíveis para serem usadas no programa. Isso é conhecido como uma **declaração import de tipo por demanda**. Somente as classes no pacote `java.util` usadas no programa são carregadas pela JVM. A declaração `import` anterior permite utilizar o nome simples de qualquer classe no pacote `java.util` no programa. Por todo este livro, por clareza, utilizamos declarações de importação de tipo simples.

Erro comum de programação 8.12

Utilizar a declaração `import` `import java.*;` causa um erro de compilação. Você deve especificar o nome exato do pacote do qual você quer importar classes.

Especificando o classpath durante a compilação

Ao compilar `Time1PackageTest`, `javac` deve localizar o arquivo `.class` para `Time1` a fim de assegurar que a classe `Time1PackageTest` utilize a classe `Time1` corretamente. O compilador utiliza um objeto especial chamado **carregador de classe** para localizar as classes de que ele precisa. O carregador de classe começa pesquisando as classes Java padrão que são empacotadas no JDK. Em seguida, ele procura **pacotes opcionais**. O Java fornece um **mecanismo de extensão** que permite que novos (opcionais) pacotes sejam adicionados ao Java para propósitos de desenvolvimento e execução. [*Nota:* o mecanismo de extensão está além do escopo deste livro. Para mais informações, visite `java.sun.com/javase/6/docs/technotes/guides/extensions/`.] Se a classe não for localizada nas classes Java padrão, nem nas classes de extensão, o carregador de classe pesquisa o **classpath**, que contém uma lista de locais em que classes são armazenadas. O classpath consiste em uma lista de diretórios ou **repositórios de arquivos**, cada um separado por um **separador de diretório** — um ponto-e-vírgula (`;`) no Windows ou um sinal de dois-pontos (`:`) no Unix/Linux/Mac OS X. Os repositórios de arquivos são arquivos individuais que contêm diretórios de outros arquivos, em geral, um formato compacto. Por exemplo, as classes padrão utilizadas pelos seus programas estão contidas no repositório de arquivos `rt.jar`, instalado com o JDK. Os repositórios de arquivos normalmente terminam com as extensões de nome de arquivo `.jar` ou `.zip`. Os diretórios e repositórios de arquivos especificados no classpath contêm as classes que você deseja disponibilizar para o compilador Java e JVM.

Por padrão, o classpath consiste apenas no diretório atual. Entretanto, o classpath pode ser modificado

1. fornecendo a opção **`-classpath`** para o compilador `javac` ou
2. configurando a **variável de ambiente `CLASSPATH`** (uma variável especial que você define e o sistema operacional mantém de modo que os aplicativos possam procurar classes nos locais especificados).

Para obter mais informações sobre o classpath, visite `java.sun.com/javase/6/docs/technotes/tools/index.html#general`. A Seção intitulada "Informações gerais" contém informações sobre a configuração do classpath para Unix/Linux e Windows.

Erro comum de programação 8.13

Especificar um classpath explícito elimina o diretório atual do classpath. Isso impede que classes no diretório atual (incluindo pacotes no diretório atual) sejam carregadas adequadamente. Se classes precisarem ser carregadas do diretório atual, inclua um ponto (`.`) no classpath para especificar o diretório atual.

Observação de engenharia de software 8.12

Em geral, é mais adequado utilizar a opção `-classpath` do compilador, em vez da variável de ambiente `CLASSPATH`, para especificar o classpath de um programa. Isso permite que cada aplicativo tenha seu próprio classpath.

Dica de prevenção de erro 8.4

Especificar o classpath com a variável de ambiente `CLASSPATH` pode resultar em erros sutis e difíceis de localizar em programas que usam diferentes versões do mesmo pacote.

Para as figuras 8.18 e 8.19, não especificamos um classpath explícito. Assim, para localizar as classes no pacote `com.deitel.jhtp.ch08` desse exemplo, o carregador de classe procura o nome no pacote no diretório atual — `com` — depois navega pela estrutura de diretórios. O diretório `com` contém o subdiretório `deitel`, `deitel` contém o subdiretório `jhtp`, e `jhtp` contém o subdiretório `ch08`. No diretório `ch08`, o arquivo `Time1.class` é carregado pelo carregador de classe para assegurar que a classe seja utilizada adequadamente no nosso programa.

Especificando o classpath ao executar um aplicativo

Ao executar um aplicativo, a JVM deve ser capaz de localizar os arquivos `.class` utilizados nesse aplicativo. Como ocorre com o compilador, o comando `java` utiliza um carregador de classe que primeiro pesquisa as classes padrão e classes de extensão para então pesquisar o classpath (o diretório atual por padrão). O classpath pode ser especificado explicitamente utilizando-se qualquer uma das técnicas discutidas para o compilador. Como ocorre com o compilador, é melhor especificar um classpath individual do programa via opções de linha de comando da JVM. Você pode especificar o classpath no comando `java` via opções de linha de comando **`-classpath`** ou **`-cp`**, seguidas por uma lista de diretórios ou repositórios de arquivos separados por ponto-e-vírgula (`;`) no Microsoft Windows ou por dois-pontos (`:`) no Unix/Linux/Mac OS X. Mais uma vez, se classes precisarem ser carregadas do diretório atual, certifique-se de incluir um ponto (`.`) no classpath para especificar o diretório atual.

8.15 Acesso de pacote

Se nenhum modificador de acesso (`public`, `protected` ou `private` — `protected` é discutido no Capítulo 9) for especificado para um método ou variável quando esse método ou variável é declarado em uma classe, o método ou variável será considerado como tendo **acesso de pacote**. Em um programa que consiste em uma declaração de classe, isso não tem nenhum efeito específico. Entretanto, se um

programa utilizar múltiplas classes no mesmo pacote (isto é, um grupo de classes relacionadas), essas classes poderão acessar diretamente os membros de acesso de pacote de outras classes por meio de referências a objetos das classes apropriadas, ou no caso de membros `static` por meio do nome de classe. O acesso de pacote é raramente utilizado.

O aplicativo na Figura 8.20 demonstra o acesso de pacote. O aplicativo contém duas classes em um único arquivo de código-fonte — a classe de aplicativo `PackageDataTest` (linhas 5–21) e a classe `PackageData` (linhas 24–41). Quando você compilar esse programa, o compilador produzirá dois arquivos .class separados — `PackageDataTest.class` e `PackageData.class`. O compilador coloca os dois arquivos `.class` no mesmo diretório, assim as classes são consideradas parte do mesmo pacote. Consequentemente, a classe `PackageDataTest` tem permissão para modificar os dados de acesso de pacote dos objetos `PackageData`. Note que você também pode colocar a classe `PackageData` (linhas 24–41) em um arquivo de código-fonte separado. Contanto que ambas as classes sejam compiladas no mesmo diretório no disco, a relação pacote-acesso ainda funcionará.

Na declaração de classe `PackageData`, as linhas 26–27 declaram as variáveis de instância `number` e `string` sem modificadores de acesso — portanto, essas são variáveis de instância de acesso de pacote. O método `main` do aplicativo `PackageDataTest` cria uma instância da classe `PackageData` (linha 9) para demonstrar a capacidade de modificar as variáveis de instância `PackageData` diretamente (como mostrado nas linhas 15–16). Os resultados da modificação podem ser vistos na janela de saída.

```
1  // Figura 8.20: PackageDataTest.java
2  // Membros de acesso de pacote de uma classe permanecem
3  // acessíveis a outras classes no mesmo pacote.
4
5  public class PackageDataTest
6  {
7     public static void main( String[] args )
8     {
9        PackageData packageData = new PackageData();
10
11       // gera saída da representação String de packageData
12       System.out.printf( "After instantiation:\n%s\n", packageData );
13
14       // muda os dados de acesso de pacote no objeto packageData
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // gera saída da representação String de packageData
19       System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20    } // fim de main
21 } // fim da classe PackageDataTest
22
23 // classe com variáveis de instância de acesso de pacote
24 class PackageData
25 {
26    int number; // variável de instância de acesso de pacote
27    String string; // variável de instância de acesso de pacote
28
29    // construtor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    } // fim do construtor PackageData
35
36    // retorna a representação String do objeto PackageData
37    public String toString()
38    {
39       return String.format( "number: %d; string: %s", number, string );
40    } // fim do método toString
41 } // fim da classe PackageData
```

```
After instantiation:
number: 0; string: Hello
After changing values:
number: 77; string: Goodbye
```

Figura 8.20 | Os membros de acesso de pacote de uma classe permanecem acessíveis a outras classes no mesmo pacote.

8.16 (Opcional) Estudo de caso de GUI e imagens gráficas: utilizando objetos com imagens gráficas

A maioria dos elementos gráficos que você viu até agora não variou com cada execução de programa. O Exercício 6.2 da Seção 6.13 solicitou que você criasse um programa que gerasse formas e cores aleatoriamente. Nesse exercício, o desenho mudava sempre que o sistema chamava `paintComponent` para redesenhar o painel. Para criar um desenho mais consistente que permaneça idêntico todas as vezes que é desenhado, devemos armazenar informações sobre as formas exibidas de modo que possamos reproduzi-las toda vez que o sistema chamar `paintComponent`. Para fazer isso, criaremos um conjunto de classes de forma que armazena informações sobre cada forma. Tornaremos essas classes "inteligentes" permitindo que os objetos dessas classes se desenhem utilizando um objeto `Graphics`.

Classe `MyLine`

A Figura 8.21 declara a classe `MyLine`, que tem todas essas capacidades. A classe `MyLine` importa `Color` e `Graphics` (linhas 3–4). As linhas 8–11 declaram as variáveis de instância para as coordenadas necessárias para desenhar uma linha, e a linha 12 declara a variável de instância que armazena a cor da linha. O construtor nas linhas 15–22 recebe cinco parâmetros, um para cada variável de instância que ele inicializa. O método `draw` nas linhas 25–29 requer um objeto `Graphics` e o utiliza para desenhar a linha na cor e nas coordenadas apropriadas.

```
1  // Figura 8.21: MyLine.java
2  // Declaração da classe MyLine.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MyLine
7  {
8     private int x1; // coordenada x da 1a. extremidade
9     private int y1; // coordenada y da 1a. extremidade
10    private int x2; // coordenada x da 2a. extremidade
11    private int y2; // coordenada y da 2a. extremidade
12    private Color myColor; // cor dessa forma
13
14    // construtor com valores de entrada
15    public MyLine( int x1, int y1, int x2, int y2, Color color )
16    {
17       this.x1 = x1; // configura a coordenada x da 1a. extremidade
18       this.y1 = y1; // configura a coordenada y da 1a. extremidade
19       this.x2 = x2; // configura a coordenada x da 2a. extremidade
20       this.y2 = y2; // configura a coordenada y da 2a. extremidade
21       myColor = color; // configura a cor
22    } // fim do construtor MyLine
23
24    // Desenha a linha na cor especificada
25    public void draw( Graphics g )
26    {
27       g.setColor( myColor );
28       g.drawLine( x1, y1, x2, y2 );
29    } // fim do método draw
30 } // fim da classe MyLine
```

Figura 8.21 | A classe `MyLine` representa uma linha.

Classe `DrawPanel`

Na Figura 8.22, declaramos a classe `DrawPanel`, que irá gerar objetos aleatórios da classe `MyLine`. A linha 12 declara um array `MyLine` para armazenar as linhas a desenhar. Dentro do construtor (linhas 15–37), a linha 17 configura a cor de segundo plano como `Color.WHITE`. A linha 19 cria o array com um comprimento aleatório entre 5 e 9. O loop nas linhas 22–36 cria um novo `MyLine` para cada elemento no array. As linhas 25–28 geram coordenadas aleatórias para as extremidades finais da linha, e as linhas 31–32 geram uma cor aleatória para a linha. A linha 35 cria um novo objeto `MyLine` com os valores aleatoriamente gerados e o armazena no array.

```
1  // Figura 8.22: DrawPanel.java
2  // Programa que utiliza a classe MyLine
3  // para desenhar linhas aleatórias.
```

```
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.util.Random;
7  import javax.swing.JPanel;
8
9  public class DrawPanel extends JPanel
10 {
11    private Random randomNumbers = new Random();
12    private MyLine[] lines; // array de linhas
13
14    // construtor, cria um painel com formas aleatórias
15    public DrawPanel()
16    {
17       setBackground( Color.WHITE );
18
19       lines = new MyLine[ 5 + randomNumbers.nextInt( 5 ) ];
20
21       // cria linhas
22       for ( int count = 0; count < lines.length; count++ )
23       {
24          // gera coordenadas aleatórias
25          int x1 = randomNumbers.nextInt( 300 );
26          int y1 = randomNumbers.nextInt( 300 );
27          int x2 = randomNumbers.nextInt( 300 );
28          int y2 = randomNumbers.nextInt( 300 );
29
30          // gera uma cor aleatória
31          Color color = new Color( randomNumbers.nextInt( 256 ),
32          randomNumbers.nextInt( 256 ), randomNumbers.nextInt( 256 ) );
33
34          // adiciona a linha à lista de linhas a ser exibida
35          lines[ count ] = new MyLine( x1, y1, x2, y2, color );
36       } // for final
37    } // fim do construtor DrawPanel
38
39    // para cada array de forma, desenha as formas individuais
40    public void paintComponent( Graphics g )
41    {
42       super.paintComponent( g );
43
44       // desenha as linhas
45       for ( MyLine line : lines )
46          line.draw( g );
47    } // fim do método paintComponent
48 } // fim da classe DrawPanel
```

Figura 8.22 | Criando objetos `MyLine` aleatórios.

O método `paintComponent` itera pelos objetos `MyLine` no array `lines` utilizando uma instrução `for` aprimorada (linhas 45–46). Cada iteração chama o método `draw` do objeto `MyLine` atual e passa a ele o objeto `Graphics` para desenhar no painel.

Classe `TestDraw`

A classe `TestDraw` na Figura 8.23 configura uma nova janela para exibir nosso desenho. Como estamos configurando as coordenadas para as linhas somente uma vez no construtor, o desenho não muda se `paintComponent` for chamado para atualizar o desenho na tela.

```
1  // Figura 8.23: TestDraw.java
2  // Criando um JFrame para exibir um DrawPanel.
3  import javax.swing.JFrame;
4
5  public class TestDraw
6  {
7     public static void main( String[] args )
8     {
9        DrawPanel panel = new DrawPanel();
10       JFrame application = new JFrame();
11
```

```
12          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13          application.add( panel );
14          application.setSize( 300, 300 );
15          application.setVisible( true );
16       } // fim de main
17    } // fim da classe TestDraw
```

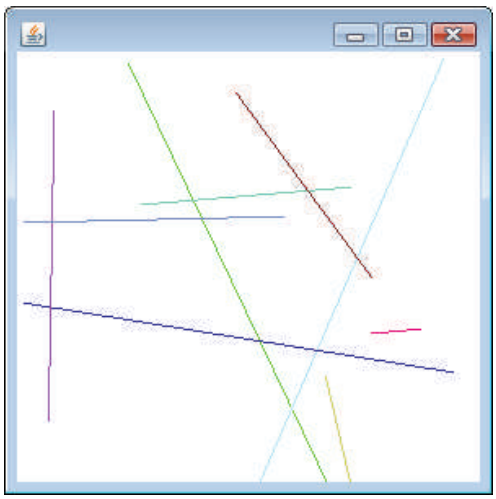


Figura 8.23 | Criando um `JFrame` para exibir um `DrawPanel`.

Exercício do Estudo de caso GUI e imagens gráficas

8.1 Estenda o programa das figuras 8.21 a 8.23 para desenhar aleatoriamente retângulos e ovais. Crie as classes `MyRectangle` e `MyOval`. Essas duas classes devem incluir as coordenadas *x1*, *y1*, *x2*, *y2*, uma cor e um flag `boolean` para determinar se a forma é uma forma preenchida. Declare um construtor em cada classe com argumentos para inicializar todas as variáveis de instância. Para ajudar a desenhar retângulos e ovais, cada classe deve fornecer os métodos `getUpperLeftX`, `getUpperLeftY`, `getWidth` e `getHeight` que calculam as coordenadas *x* e *y* do canto superior esquerdo e a largura e altura, respectivamente. A coordenada *x* superior esquerda é o menor dos dois valores da coordenada *x*, a coordenada *y* superior esquerda é o menor dos dois valores da coordenada *y*, a largura é o valor absoluto da diferença entre os dois valores da coordenada *x*, e a altura é o valor absoluto da diferença entre os dois valores das coordenadas *y*.

A classe `DrawPanel`, que estende `JPanel` e trata a criação das formas, deve declarar três arrays, um para cada tipo de forma. O comprimento de cada array deve ser um número aleatório entre 1 e 5. O construtor da classe `DrawPanel` preencherá cada um dos arrays com formas de posição aleatória, tamanho, cor e preenchimento.

Além disso, modifique todas as três classes de forma a incluir o seguinte:

a) Um construtor sem argumentos que configura todas as coordenadas da forma como `0`, a cor da forma como `Color.BLACK` e a propriedade preenchida como `false` (`MyRectangle` e `MyOval` somente).

b) Métodos *set* para as variáveis de instância em cada classe. Os métodos que configuram um valor de coordenada devem verificar se o argumento é maior ou igual a zero antes de configurar a coordenada — se não for, devem configurar a coordenada como zero. O construtor deve chamar os métodos *set* em vez de inicializar as variáveis locais diretamente.

c) Os métodos *get* para as variáveis de instância em cada classe. O método `draw` deve referenciar as coordenadas pelos métodos *get* em vez de acessá-las diretamente.

8.17 Conclusão

Neste capítulo, apresentamos conceitos adicionais sobre classes. O estudo de caso da classe `Time` apresentou uma declaração de classe completa que consiste em dados `private`, construtores `public` sobrecarregados para flexibilidade da inicialização, métodos *set* e *get* para manipular os dados da classe e métodos que retornaram representações de `String` de um objeto `Time` em duas formas diferentes. Você também aprendeu que cada classe pode declarar um método `toString` que retorna uma representação `String` de um objeto da classe e que o método `toString` pode ser chamado implicitamente sempre que um objeto de uma classe aparece no código onde se espera uma `String`.

Você aprendeu que a referência `this` é utilizada implicitamente nos métodos não `static` de uma classe para obter acesso às variáveis de instância da classe e outros métodos não `static`. Você também viu utilizações explícitas da referência `this` para acessar os membros da classe (incluindo campos sombreados) e como utilizar a palavra-chave `this` em um construtor para chamar um outro construtor da classe.

Discutimos as diferenças entre construtores padrão fornecidos pelo compilador e construtores sem argumentos fornecidos pelo programador. Você aprendeu que uma classe pode ter referências a objetos de outras classes como membros — um conceito conhecido como composição. Você viu o tipo de classe `enum` e aprendeu como essa classe pode ser utilizada para criar o conjunto de constantes para uso em

um programa. Discutimos a capacidade da coleta de lixo do Java e como ela reivindica (inesperadamente) a memória de objetos que não são mais utilizados. O capítulo explicou a motivação da utilização de campos `static` em uma classe e demonstrou como declarar e utilizar campos e métodos `static` nas suas próprias classes. Você também aprendeu a declarar e inicializar variáveis `final`.

Aprendeu a empacotar suas próprias classes para reutilização e como importar essas classes para um aplicativo. Por fim, você aprendeu que campos declarados sem um modificador de acesso têm acesso de pacote por padrão. Você viu o relacionamento entre classes no mesmo pacote que permite a cada classe em um pacote acessar os membros de acesso de pacote de outras classes no pacote.

No próximo capítulo, você aprenderá um aspecto importante da programação orientada a objetos em Java — a herança. Veremos que todas as classes em Java são relacionadas direta ou indiretamente à classe chamada `Object`. Você também começará a entender como os relacionamentos entre classes permitem construir aplicativos mais poderosos.

Resumo

Seção 8.2 Estudo de caso da classe `Time`

- Os métodos `public` de uma classe também são conhecidos como os serviços `public` ou interface `public` da classe. Eles apresentam aos clientes da classe uma visão dos serviços que a classe fornece.
- Os membros `private` de uma classe não são acessíveis aos clientes da classe.
- Um objeto que contém dados consistentes tem valores de dados que sempre são mantidos no intervalo.
- Um valor passado para um método a fim de modificar uma variável de instância é um valor consistente se ele estiver no intervalo aceitável da variável de instância. Um valor correto sempre é um valor consistente, mas um valor consistente não é correto se um método receber um valor fora do intervalo e configurá-lo como um valor consistente para manter o objeto em um estado consistente.
- O método `static format` da classe `String` é semelhante ao método `System.out.printf` exceto que `format` retorna uma `String` formatada em vez de exibi-la em uma janela de comando.
- Todos os objetos no Java têm um método `toString` que retorna uma representação `String` do objeto. O método `toString` é chamado implicitamente quando um objeto aparece no código onde uma `String` é necessária.

Seção 8.3 Controlando o acesso a membros

- Os modificadores de acesso `public` e `private` controlam o acesso às variáveis e métodos de uma classe.
- O principal propósito dos métodos `public` é apresentar para os clientes da classe uma visualização dos serviços que a classe fornece. Os clientes não precisam se preocupar com a maneira como a classe realiza suas tarefas.
- As variáveis `private` e os métodos `private` de uma classe (isto é, os detalhes da sua implementação) não são acessíveis aos clientes dela.

Seção 8.4 Referenciando membros do objeto atual com a referência `this`

- Um método não `static` de um objeto utiliza implicitamente a palavra-chave `this` para referenciar as variáveis de instância do objeto e outros métodos. A palavra-chave `this` também pode ser usada explicitamente.
- O compilador produz um arquivo separado com a extensão `.class` para cada classe compilada.
- Se uma variável local tiver o mesmo nome do campo de uma classe, a variável local sombreará o campo. Você pode utilizar a referência `this` para referenciar o campo sombreado explicitamente.

Seção 8.5 Estudo de caso da classe `Time`: construtores sobrecarregados

- Construtores sobrecarregados permitem que objetos de uma classe sejam inicializados de diferentes maneiras. O compilador diferencia construtores sobrecarregados pelas suas assinaturas.
- Para chamar um construtor de uma classe a partir de outro da mesma classe, você pode utilizar a palavra-chave `this` seguida por parênteses contendo os argumentos de construtor. Tal chamada de construtor deve aparecer como a primeira instrução no corpo do construtor.

Seção 8.6 Construtores padrão e sem argumentos

- Se nenhum construtor for fornecido em uma classe, o compilador criará um construtor padrão.
- Se uma classe declarar construtores, o compilador não criará um construtor padrão. Nesse caso, você deve declarar um construtor sem argumentos se a inicialização padrão for requerida.

Seção 8.7 Notas sobre os métodos set *e* get

- Os métodos *set* são comumente chamados de métodos modificadores porque geralmente alteram um valor. Os métodos *get* são comumente chamados de métodos de acesso ou métodos de consulta. Um método predicado testa se uma condição é verdadeira ou falsa.

Seção 8.8 Composição

- Uma classe pode ter referências a objetos de outras classes como membros. Isso é chamado composição e, às vezes, é referido como um relacionamento *tem um*.

Seção 8.9 Enumerações

- Todos os tipos `enum` são tipos por referência. Um tipo `enum` é declarado com uma declaração `enum`, que é uma lista separada por vírgulas de constantes `enum`. A declaração pode incluir opcionalmente outros componentes das classes tradicionais, como construtores, campos e métodos.
- As constantes `enum` são implicitamente `final`, porque declaram constantes que não devem ser modificadas.
- Constantes `enum` são implicitamente `static`.
- Qualquer tentativa de criar um objeto de um tipo `enum` com um operador `new` resulta em um erro de compilação.
- Constantes `enum` podem ser utilizadas em qualquer lugar em que constantes podem ser utilizadas, como nos rótulos `case` das instruções `switch` e para controlar instruções `for` aprimoradas.
- Cada constante `enum` em uma declaração `enum` é opcionalmente seguida por argumentos que são passados para o construtor `enum`.
- Para cada `enum`, o compilador gera um método `static` chamado `values` que retorna um array das constantes do `enum` na ordem em que elas foram declaradas.
- O método `EnumSet static range` recebe a primeira e a última constantes `enum` em um intervalo e retorna um `EnumSet` que contém todas as constantes entre essas duas constantes, inclusive.

Seção 8.10 Coleta de lixo e o método `finalize`

- Toda classe no Java tem os métodos da classe `Object`, um do quais é o método `finalize`.
- A Java Virtual Machine (JVM) realiza a coleta de lixo automaticamente para reivindicar a memória ocupada pelos objetos que não estão mais em uso. Quando não houver mais referências a um objeto, o objeto é elegível para a coleta de lixo. A memória desse objeto pode ser reivindicada quando a JVM executa seu coletor de lixo.
- O método `finalize` é chamado pelo coletor de lixo um pouco antes de ele reivindicar a memória do objeto. O método `finalize` não aceita parâmetros e tem o tipo de retorno `void`.
- O coletor de lixo nunca pode executar antes de um programa terminar. Portanto, não fica claro se, ou quando, o método `finalize` será chamado.

Seção 8.11 Membros da classe `static`

- Uma variável `static` representa as informações de escopo de classe que são compartilhadas entre todos os objetos da classe.
- Variáveis estáticas têm escopo de classe. Os membros `public static` de uma classe podem ser acessados por meio de uma referência a qualquer objeto da classe ou podem ser acessados qualificando o nome de membro com o nome de classe e um ponto (`.`). O código de cliente pode acessar os membros `private static` de uma classe apenas pelos métodos da classe.
- Os membros de classe `static` existem logo que a classe estiver carregada na memória.
- Um método declarado `static` não pode acessar membros de classe não `static`, porque um método `static` pode ser chamado mesmo quando nenhum objeto da classe foi instanciado.
- A referência `this` não pode ser utilizada em um método `static`.

Seção 8.12 Importação `static`

- Uma declaração import `static` permite aos programadores referenciar membros `static` importados sem o nome de classe e um ponto (`.`). Uma única declaração import `static` importa um membro `static`, e uma import `static` por demanda importa todos os membros `static` de uma classe.

Seção 8.13 Variáveis de instância `final`

- No contexto de um aplicativo, o princípio do menor privilégio afirma que deve ser concedida ao código somente a quantidade de privilégio e acesso de que ele precisa para realizar sua tarefa designada.
- A palavra-chave `final` especifica que uma variável não é modificável. Essas variáveis podem ser inicializadas quando são declaradas ou pelos construtores de uma classe.

Seção 8.14 Estudo de caso da classe `Time`: criando pacotes

- Cada classe na Java API pertence a um pacote que contém um grupo de classes relacionadas. Os pacotes ajudam a gerenciar a complexidade de componentes de um aplicativo e facilitam a reutilização de software.
- Os pacotes fornecem uma convenção para nomes únicos de classes que ajuda a evitar conflitos entre nomes de classe.
- Antes que uma classe possa ser importada para múltiplos aplicativos, ela deve ser colocada em um pacote. Há somente uma declaração `package` em cada arquivo de código-fonte Java e ela deve preceder todas as outras declarações e instruções no arquivo.
- Cada nome de pacote deve iniciar com seu nome de domínio Internet na ordem inversa. Depois que o nome de domínio é invertido, você pode escolher qualquer outro nome para seu pacote.

- Ao compilar uma classe em um pacote, a opção de linha de comando `javac -d` especifica onde armazenar o pacote e faz com que o compilador crie os diretórios do pacote, se eles não existirem.
- O nome `package` é parte do nome completamente qualificado de classe. Isso ajuda a evitar conflitos de nomes.
- Uma declaração import de tipo simples especifica uma classe a importar. Uma declaração import de tipo por demanda importa somente as classes que o programa utiliza a partir de um pacote particular.
- O compilador utiliza um carregador de classe para localizar as classes de que ele precisa no classpath. O classpath consiste em uma lista de diretórios ou repositórios de arquivos, cada um separado por um separador de diretório.
- O classpath para o compilador e a JVM pode ser especificado fornecendo a opção `-classpath` para o comando `javac` ou `java`, ou configurando a variável de ambiente `CLASSPATH`. Se for necessário carregar classes do diretório atual, inclua um ponto (`.`) no classpath.

Seção 8.15 Acesso de pacote

- Se nenhum modificador de acesso for especificado para um método ou variável quando esse método ou variável é declarado em uma classe, o método ou variável é considerado como tendo acesso de pacote.

Terminologia

acesso a pacotes, 267
carregador de classe, 267
classpath, 267
`-classpath`, opção de linha de comando para `javac`, 267
coleta de lixo, 258
coletor de lixo, 258
colisão de nomes, 265
colocando chamadas de método em linha, 251
composição, 253
conflito de nomes, 265
construtor sem argumento, 250
construtores sobrecarregados, 248
`-cp`, opção de linha de comando para `java`, 267
`-d`, opção de compilador, 265
declaração import do tipo por demanda, 266
declaração import do tipo simples, 266
`enum`, construtor, 256
`enum`, declaração, 256
`EnumSet`, classe, 257
`finalize`, método, 258
`format`, método da classe `String`, 244
informações de nível de classe, 258
mecanismo de extensão, 267
método de acesso, 252
método de consulta, 252
método modificador, 252
método predicado, 253
nome de classe completamente qualificado, 265
nome simples de uma classe, 265
objeto imutável, 260
`package`, declaração, 264
pacote opcional, 267
princípio do menor privilégio, 262
`public`, interface, 242
`public`, serviços (de uma classe), 242
`range`, método da classe `EnumSet`, 257
relacionamento *tem um*, 253
repositórios de arquivos, 267
separador de diretório, 267
`static`, campo, 258
`static`, importação, 261
`static`, importação simples, 261
`static`, importação sob demanda, 261
terminação, preparação, 258
`this`, palavra-chave, 246
`this`, referência, 246
`values`, método de um `enum`, 257
variável de classe, 258

Exercícios de autorrevisão

8.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Ao compilar uma classe em um pacote, a opção de linha de comando `javac` _____ especifica onde armazenar o pacote e faz com que o compilador crie os diretórios do pacote, se eles não existirem.

b) O método `static` da classe `String` _____ é semelhante ao método `System.out.printf`, mas retorna uma `String` formatada em vez de exibir uma `String` em uma janela de comando.

c) Se um método contiver uma variável local com o mesmo nome de um dos campos da sua classe, a variável local _____ o campo no escopo desse método.

d) O método _____ é chamado pelo coletor de lixo um pouco antes de reivindicar a memória de um objeto.

e) Uma declaração de _____ especifica uma classe a ser importada.

f) Se uma classe declarar construtores, o compilador não criará um(a) _____.

g) O método _____ de um objeto é chamado implicitamente quando um objeto aparece no código em que uma `String` é necessária.

h) Métodos *get* são comumente chamados de _____ ou _____.

i) Um método _____ testa se uma condição é verdadeira ou falsa.

j) Para cada `enum`, o compilador gera um método `static` chamado _____ que retorna um array das constantes do `enum` na ordem em que elas foram declaradas.

k) A composição às vezes é referida como um relacionamento _____.

l) Uma declaração de _____contém uma lista separada por vírgulas de constantes.

m) Uma variável _____ representa as informações de escopo de classe que são compartilhadas por todos os objetos da classe.

n) Uma declaração _____ importa um membro `static`.

o) O _____ declara que só deve ser concedida ao código a quantidade de privilégio e acesso de que ele precisa para realizar a tarefa designada.

p) A palavra-chave _____ especifica que uma variável não é modificável.

q) Há somente um(a) _____ em um arquivo de código-fonte Java e deve preceder todas as outras declarações e instruções no arquivo.

r) Um(a) declaração _____ importa somente as classes que o programa utiliza em um pacote em particular.

s) O compilador utiliza um(a) _____ para localizar as classes de que ele precisa no classpath.

t) O classpath para o compilador e para a JVM pode ser especificado com a opção _____ para o comando `javac` ou `java`, ou configurando a variável de ambiente _____.

u) Métodos *set* são comumente chamados _____ porque eles geralmente alteram um valor.

v) Um(a) _____ importa todos os membros `static` de uma classe.

w) Os métodos `public` de uma classe também são conhecidos como _____ ou _____ da classe.

x) Em um objeto, os dados_____ são valores dos dados que são sempre mantidos no intervalo.

Respostas dos exercícios de autorrevisão

8.1 a) `-d`. b) `format`. c) sombras. d) `finalize`. e) declaração de importação simples. f) construtor padrão. g) `toString`. h) métodos de acesso, métodos de consulta. i) predicado. j) `values`. k) *tem um*. l) `enum`. m) `static`. n) import `static` simples. o) princípio do menor privilégio. p) `final`. q) declaração `package`. r) importação por demanda. s) carregador de classe. t) `-classpath`, `CLASSPATH`. u) métodos modificáveis. v) importação `static` por demanda. w) serviços `public`, interface `public`. x) consistente.

Exercícios

8.2 Explique a noção de acesso de pacote em Java. Explique os aspectos negativos do acesso de pacote.

8.3 O que acontece quando um tipo de retorno, mesmo `void`, é especificado para um construtor?

8.4 (***Classe Rectangle***) Crie uma classe `Rectangle` com atributos `length` e `width`, cada um dos quais assume o padrão de 1. Forneça métodos que calculem o perímetro e área do retângulo. A classe tem métodos *set* e *get* para o comprimento (`length`) e a largura (`width`). Os métodos *set* devem verificar se `length` e `width` são, cada um, números de ponto flutuante maiores que 0,0 e menores que 20,0. Escreva um programa para testar a classe `Rectangle`.

8.5 (***Modificando a representação interna de dados de uma classe***) Seria perfeitamente razoável que a classe `Time2` da Figura 8.5 represente a hora internamente como o número de segundos a partir da meia-noite em vez dos três valores inteiros `hour`, `minute` e `second`. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados. Modifique a classe `Time2` da Figura 8.5 para implementar `Time2` como o número de segundos desde a meia-noite e mostrar que não há alteração visível para os clientes da classe.

8.6 (***Classe SavingsAccount***) Crie uma classe `SavingsAccount`. Utilize uma variável `static annualInterestRate` para armazenar a taxa de juros anual para todos os correntistas. Cada objeto da classe contém uma variável de instância `private savingsBalance` para indicar a quantidade que o cliente atualmente tem em depósito. Forneça método `calculateMonthlyInterest` para calcular os juros mensais multiplicando o `savingsBalance` por `annualInterestRate` dividido por 12 — esses juros devem ser adicionados ao `savingsBalance`. Forneça um método `static modifyInterestRate` que configure `annualInterestRate` como um novo valor. Escreva um programa para testar a classe `SavingsAccount`. Instancie dois objetos `savingsAccount`, `saver1` e `saver2`, com saldos de R$ 2.000,00 e R$ 3.000,00, respectivamente. Configure `annualInterestRate` como 4% e então calcule o juro mensal de cada um dos 12 meses e imprima os novos saldos para os dois poupadores. Em seguida, configure `annualInterestRate` para 5%, calcule a taxa do próximo mês e imprima os novos saldos para os dois poupadores.

8.7 (***Aprimorando a classe Time2***) Modifique a classe `Time2` da Figura 8.5 para incluir um método `tick` que incrementa a hora armazenada em um objeto `Time2` em um segundo. Forneça um método `incrementMinute` para incrementar o minuto por um e o método `incrementHour` para incrementar a hora por uma. O objeto `Time2` sempre deve permanecer em um estado consistente. Escreva um programa que testa o método `tick`, o método `incrementMinute` e o método `incrementHour` para assegurar que eles funcionam corretamente. Certifique-se de testar os seguintes casos:

a) incrementar para o próximo minuto,

b) incrementar para a próxima hora e

c) incrementar para o próximo dia (isto é, 11:59:59 PM para 12:00:00 AM).

8.8 (***Aprimorando a classe Date***) Modifique a classe `Date` da Figura 8.7 para realizar uma verificação de erros nos valores inicializadores das variáveis de instância `month`, `day` e `year` (atualmente ela valida somente o mês e dia). Forneça um método `nextDay` para incrementar o dia por um. O objeto `Date` deve permanecer em um estado consistente. Escreva um programa que testa o método `nextDay` em um loop que imprime a data durante cada iteração para ilustrar que esse método funciona corretamente. Teste os seguintes casos:

a) incrementar para o próximo mês e

b) incrementar para o próximo ano.

8.9 (***Retornando indicadores de erros de métodos***) Modifique os métodos *set* na classe `Time2` da Figura 8.5 para retornar valores de erros apropriados se ocorrer uma tentativa de configurar uma das variáveis de instância `hour`, `minute` ou `second` de um objeto da classe `Time2` como um valor inválido. [*Dica:* utilize tipos de retorno `boolean` em cada método.] Escreva um programa que teste esses novos métodos *set* e gere a saída de mensagens de erro quando valores incorretos são fornecidos.

8.10 Reescreva a Figura 8.14 para utilizar uma declaração `import` separada para cada membro `static` da classe `Math` que é utilizado no exemplo.

8.11 Escreva um tipo `enum TrafficLight`, cuja constante (`RED`, `GREEN`, `YELLOW`) aceite um parâmetro — a duração da luz. Escreva um programa para testar o `enum TrafficLight` de modo que ele exiba a constante `enum` e suas durações.

8.12 (***Números complexos***) Crie uma classe chamada `Complex` para realizar aritmética com números complexos. Os números complexos têm a forma

parteReal + *parteImaginária* * *i*

onde *i* é

$\sqrt{-1}$

Escreva um programa para testar sua classe. Utilize variáveis de ponto flutuantes para representar os dados `private` da classe. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos `public` que realizam as seguintes operações:

a) Somar dois números `Complex`: as partes reais são somadas de um lado e as partes imaginárias são somadas de outro.

b) Subtrair dois números `Complex`: a parte real do operando direito é subtraída da parte real do operando esquerdo e a parte imaginária do operando direito é subtraída da parte imaginária do operando esquerdo.

c) Imprimir números `Complex` na forma `(a, b)`, onde `a` é a parte real e `b` é a parte imaginária.

8.13 (***Classe DateAndTime***) Crie uma classe `DateAndTime` que combina a classe `Time2` modificada do Exercício 8.7 e a classe `Date` modificada do Exercício 8.8. Modifique o método `incrementHour` para chamar o método `nextDay` se a hora for incrementada para o próximo dia. Modifique os métodos `toString` e `toUniversalString` para gerar saída da data além da hora. Escreva um programa para testar a nova classe `DateAndTime`. Especificamente, teste o incremento de tempo para o próximo dia.

8.14 (***Conjunto de inteiros***) Crie a classe `IntegerSet`. Cada objeto `IntegerSet` pode armazenar inteiros no intervalo de 0 a 100. O conjunto é representado por um array de `boolean`s. O elemento do array `a[i]` é `true` se o inteiro *i* estiver no conjunto. O elemento do array `a[j]` é `false` se o inteiro *j* não estiver no conjunto. O construtor sem argumentos inicializa o array como um "conjunto vazio" (isto é, todos os valores `false`).

Forneça os seguintes métodos: o método `union` cria um conjunto que é a união teórica de dois conjuntos existentes (isto é, um elemento do array do novo conjunto é configurado como `true` se esse elemento for `true` em qualquer um dos conjuntos existentes ou em ambos; caso contrário, esse elemento é configurado como `false`). O método `intersection` cria um conjunto que é a interseção teórica de dois conjuntos existentes (isto é, um elemento do array do novo conjunto é configurado como `false` se esse elemento for `false` em qualquer um ou em ambos os conjuntos existentes — caso contrário, esse elemento é configurado como `true`). O método `insertElement` insere um novo inteiro *k* em um conjunto (configurando `a[k]` como `true`). O método `deleteElement` exclui o inteiro *m* (configurando `a[m]` como `false`). O método `toString` retorna uma `String` contendo um conjunto como uma lista de números separados por espaços. Inclua somente os elementos que estão presentes no conjunto. Utilize `---` para representar um conjunto vazio. O método `isEqualTo` determina se dois conjuntos são iguais. Escreva um programa para testar a classe `IntegerSet`. Instancie vários objetos `IntegerSet`. Teste se todos os seus métodos funcionam adequadamente.

8.15 (***Classe Date***) Crie uma classe `Date` com as seguintes capacidades:

a) Gerar saída da data em múltiplos formatos, como

```
MM/DD/YYYY
June 14, 1992
DDD YYYY
```

b) Utilizar construtores sobrecarregados para criar objetos `Date` inicializados com datas dos formatos na parte (a). No primeiro caso o construtor deve receber três valores inteiros. No segundo caso deve receber uma `String` e dois valores inteiros. No terceiro caso deve receber dois valores inteiros, o primeiro representando o número de dias no ano. [*Dica:* para converter a representação de `String` do mês em um valor numérico, compare as `String`s utilizando o método `equals`. Por exemplo, se `s1` e `s2` forem `String`s, a chamada de método `s1.equals(s2)` retorna `true` se as `String`s forem idênticas e, caso contrário, retorna `false`.]

8.16 (***Números racionais***) Crie uma classe chamada `Rational` para realizar aritmética com frações. Escreva um programa para testar sua classe. Utilize variáveis do tipo inteiro para representar as variáveis de instância `private` da classe — `numerator` e `denominator`. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve armazenar a fração em uma forma reduzida. A fração

`2/4`

é equivalente a `1/2` e seria armazenada no objeto como 1 no `numerator` e 2 no `denominator`. Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos `public` que realizam cada uma das operações a seguir:

a) Somar dois números `Rational`: o resultado da adição deve ser armazenado na forma reduzida.

b) Subtrair dois números `Rational`: o resultado da subtração deve ser armazenado na forma reduzida.

c) Multiplicar dois números `Rational`: o resultado da multiplicação deve ser armazenado na forma reduzida.

d) Dividir dois números `Rational`: o resultado da divisão deve ser armazenado na forma reduzida.

e) Retornar uma representação `String` de um número `Rational` na forma `a/b`, onde `a` é o `numerator` e `b` é o `denominator`.

f) Retornar uma representação `String` de um número `Rational` no formato de ponto flutuante. (Considere a possibilidade de fornecer capacidades de formatação que permitam que o usuário da classe especifique o número de dígitos de precisão à direita do ponto de fração decimal.)

8.17 (***Classe HugeInteger***) Crie uma classe `HugeInteger` que utiliza um array de 40 elementos de dígitos para armazenar inteiros com até 40 dígitos. Forneça os métodos `parse`, `toString`, `add` e `subtract`. O método `parse` deve receber uma `String`, extrair cada dígito utilizando o método `charAt` e colocar o equivalente inteiro de cada dígito no array de inteiros. Para comparar objetos `HugeInteger`, forneça os métodos a seguir: `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` e `isLessThanOrEqualTo`. Cada um destes é um mé-

todo predicado que retorna `true` se o relacionamento estiver contido entre os dois objetos `HugeInteger` e retorna `false` se o relacionamento não estiver contido. Forneça um método predicado `isZero`. Se você se sentir ambicioso, também forneça os métodos `multiply`, `divide` e `remainder`. [*Nota:* valores `boolean` primitivos podem ser gerados como as palavras "true" ou "false" com o especificador de formato `%b`.]

8.18 **(*Jogo da velha*)** Crie uma classe `TicTacToe` que permitirá escrever um programa para reproduzir o jogo da velha. A classe contém um array bidimensional privado 3 por 3. Utilize uma lista para representar o valor em cada célula do array. As constantes da enumeração devem ser chamadas de `X`, `O` e `EMPTY` (para uma posição que não contém um `X` ou um `O`). O construtor deve inicializar os elementos board (tabuleiro) como `EMPTY`. Permita dois jogadores humanos. Para onde quer que o primeiro jogador se mova, coloque um `X` no quadrado especificado; coloque um `O` no local para o qual o segundo jogador se mover. Todo movimento deve ocorrer em um quadrado vazio. Depois de cada jogada, determine se o jogo foi ganho ou se aconteceu um empate. Se você se sentir motivado, modifique seu programa de modo que o computador faça o movimento para um dos jogadores. Além disso, permita que o jogador especifique se quer ser o primeiro ou o segundo. Se você se sentir excepcionalmente motivado, desenvolva um programa que jogue o jogo da velha tridimensional em uma grade 4 por 4 por 4 [*Nota:* esse é um projeto extremamente desafiador!].

Fazendo a diferença

8.19 **(*Controle de tráfego aéreo*)** Todos os dias, de acordo com a National Air Traffic Controllers Association (`www.natca.org/mediacenter/bythenumbers.msp`), há mais de 87.000 voos nos Estados Unidos, incluindo voos comerciais, voos de carga, e assim por diante, e a tendência de longo prazo é que a atividade de tráfego aéreo cresça junto com a população. Assim como cresce o tráfego aéreo, também crescem os desafios para os controladores de tráfego aéreo, que controlam os voos e fornecem instruções aos pilotos para garantir um fluxo de tráfego seguro no espaço aéreo.

Neste exercício, você criará uma classe `Flight` que pode ser utilizada em um simulador de controle do tráfego aéreo simples. O método `main` do aplicativo atuará como controle de tráfego aéreo. Visite sites como

`www.howstuffworks.com/air-traffic-control.htm`

para pesquisar como o sistema de controle do tráfego aéreo funciona. Então identifique alguns atributos-chave de um `Flight` em um sistema de controle de tráfego aéreo. Pense nos diferentes estados nos quais um avião pode estar desde a hora em que ele estaciona no portão de um aeroporto até chegar ao seu destino — estacionado, taxeando antes de decolar ou após pousar, esperando para decolar, decolando, subindo, e assim por diante. Utilize uma enumeração `FlightStatus` para representar esses estados. Os atributos poderiam incluir o modelo e a fabricação do avião, a velocidade aérea atual, altitude atual, direção, a transportadora, hora de partida, hora de chegada estimada, a origem e o destino. A origem e o destino devem ser especificados usando códigos de aeroporto de três letras padrão, como `BOS` para Boston e `LAX` para Los Angeles (esses códigos estão disponíveis em `world-airport-codes.com`). Forneça os métodos *set* e *get* para manipular esses e qualquer outro atributo que for identificado. Em seguida, identifique os comportamentos da classe e os implemente como os métodos da classe. Inclua comportamentos como `changeAltitude`, `reduceSpeed` e `beginLandingApproach`. O construtor `Flight` deve inicializar atributos de um `Flight`. Você também deve fornecer um método `toString` que retorna uma a representação `String` do status atual de um `Flight` (por exemplo, estacionado no portão, taxeando, decolando, mudando de altitude). Essa `String` deve incluir todos os valores de variável de instância do objeto.

Quando o aplicativo executar, `main` exibirá a mensagem, `"Air Traffic Control Simulator"`, então irá criar e interagir com três objetos `Flight` que representam aviões que estão atualmente voando ou se preparando para voar. Por uma questão de simplicidade, a confirmação do `Flight` de cada ação será uma mensagem exibida na tela quando o método apropriado for chamado no objeto. Por exemplo, se você chamar o método `changeAltitude` de um voo, o método deve:

a) Exibir uma mensagem contendo a companhia aérea, o número do voo, a `"changing altitude"`, a altitude atual e a nova altitude.

b) Mude o estado da variável de instância `status` para `CHANGING_ALTITUDE`.

c) Mude o valor da variável de instância `newAltitude`.

Em `main`, crie e inicialize três objetos `Flight` que estão em estados diferentes — por exemplo, um poderia estar no portão, um poderia estar preparando-se para a decolagem e o outro preparando-se para aterrissar. O método `main` deve enviar mensagens para (invocar métodos sobre) objetos `Flight`. À medida que um objeto `Flight` recebe cada mensagem, ele deve exibir uma mensagem de confirmação do método sendo chamado — como "[nome da companhia aérea] [Número do voo] mudar de altitude de 20.000 para 25.000 pés". O método também deve atualizar informações de estado apropriadas no objeto `Flight`. Por exemplo, se o Controle de Tráfego Aéreo enviar uma mensagem como "[Companhia Aérea] [número do voo] descendo a 12.000 pés", o programa deve executar uma chamada de método como `flight1.changeAltitude(12000)`, que exibiria uma mensagem de confirmação e estabeleceria a variável de instância `newAltitude` para `12000`. [*Nota:* suponha que a variável de instância `currentAltitude` está sendo estabelecida automaticamente pelo altímetro do avião.]

9

Nunca diga que você conhece inteiramente uma pessoa, até dividir uma herança com ela.
— **Johann Kasper Lavater**

Esse método é para definir como o número de uma classe a classe de todas as classes similares à classe fornecida.
— **Bertrand Russell**

Melhor que herdar uma biblioteca, é colecionar uma.
— **Augustine Birrell**

Save base authority from others' books. (Sempre foi despiciendo o lucro avaro que nos vem de alfarrábio ou pergaminho.)
— **William Shakespeare**

Programação orientada a objetos: herança

Objetivos

Neste capítulo, você aprenderá:

- Como a herança promove a capacidade de reutilização de software.
- As noções de superclasses e subclasses.
- A utilizar a palavra-chave `extends` para criar uma classe que herda atributos e comportamentos de outra classe.
- A utilizar o modificador de acesso `protected` para fornecer acesso de métodos de subclasse a membros de superclasse.
- A acessar membros de superclasse com `super`.
- Como os construtores são utilizados em hierarquias de herança.
- Os métodos da classe `Object`, a superclasse direta ou indireta de todas as classes em Java.

Sumário

9.1 Introdução
9.2 Superclasses e subclasses
9.3 Membros `protected`
9.4 Relacionamento entre superclasses e subclasses
9.4.1 Criando e utilizando uma classe `CommissionEmployee`
9.4.2 Criando e utilizando uma classe `BasePlusCommissionEmployee`
9.4.3 Criando uma hierarquia de herança `CommissionEmployee-BasePlus-CommissionEmployee`
9.4.4 Hierarquia de herança `CommissionEmployee-BasePlusCommissionEmployee` utilizando variáveis de instância `protected`
9.4.5 Hierarquia de herança `CommissionEmployee-BasePlusCommissionEmployee` utilizando variáveis de instância `private`
9.5 Construtores em subclasses
9.6 Engenharia de software com herança
9.7 Classe `Object`
9.8 (Opcional) Estudo de caso de GUIs e imagens gráficas: exibindo texto e imagens com rótulos
9.9 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

9.1 Introdução

Este capítulo continua nossa discussão sobre programação orientada a objetos (*Object-Oriented Programming* — OOP) introduzindo uma de suas principais capacidades — **herança**, que é uma forma de reutilização de software em que uma nova classe é criada absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas. Com a herança, você pode economizar tempo durante o desenvolvimento de um programa baseando novas classes no software existente testado, depurado e de alta qualidade. Isso também aumenta a probabilidade de que um sistema será implementado e mantido efetivamente.

Ao criar uma classe, em vez de declarar membros completamente novos, você pode designar que a nova classe deve herdar membros de uma classe existente. A classe existente é chamada de **superclasse** e a nova classe, de **subclasse**. (A linguagem de programação C++ refere-se à superclasse como a **classe básica** e à subclasse como a **classe derivada**.) Cada subclasse pode tornar-se uma superclasse para futuras subclasses.

Uma subclasse pode adicionar seus próprios campos e métodos. Portanto, uma subclasse é mais específica que sua superclasse e representa um grupo mais especializado de objetos. A subclasse expõe os comportamentos da sua superclasse e pode adicionar comportamentos que são específicos à subclasse. É por isso que a herança é às vezes conhecida como **especialização**.

A **superclasse direta** é a superclasse a partir da qual a subclasse herda explicitamente. Uma **superclasse indireta** é qualquer superclasse acima da classe direta na **hierarquia de classes**, que define os relacionamentos de herança entre as classes. No Java, a hierarquia de classes inicia com a classe `Object` (no pacote `java.lang`), que *toda* classe em Java direta ou indiretamente **estende** (ou "herda de"). A Seção 9.7 lista os métodos da classe `Object` que são herdados por todas as outras classes Java. O Java só suporta **herança única**, na qual cada classe é derivada exatamente de uma superclasse direta. Ao contrário de C++, o Java não suporta herança múltipla (que ocorre quando uma classe é derivada de mais de uma superclasse direta). O Capítulo 10, "Programação orientada a objetos: polimorfismo", explica como os programadores em Java podem utilizar interfaces para obter muitos dos benefícios da herança múltipla e, ao mesmo tempo, evitar os problemas associados.

Distinguimos entre o **relacionamento *é um*** e o **relacionamento *tem um***. *É um* representa a herança. Em um relacionamento *é um*, um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse. Por exemplo, um carro *é um* veículo. Por contraste, *tem um* representa a composição (ver Capítulo 8). Em um relacionamento *tem um*, um objeto contém como membros referências a outros objetos. Por exemplo, um carro *tem uma* direção (e um objeto carro tem uma referência a um objeto direção).

Novas classes podem herdar de classes em **bibliotecas de classes**. As organizações desenvolvem suas próprias bibliotecas de classes e tiram proveito de outras disponíveis no mundo. Algum dia, a maioria dos softwares novos provavelmente será construída a partir de **componentes reutilizáveis padronizados**, assim como automóveis e a maioria dos hardwares de computadores são construídos hoje. Isso facilitará o desenvolvimento de softwares mais poderosos, em maior número e mais baratos.

9.2 Superclasses e subclasses

Frequentemente um objeto de uma classe também *é um* objeto de outra classe. Por exemplo, em geometria, um retângulo *é um* quadrilátero — uma forma de quatro lados — assim como o são os quadrados, paralelogramos e trapézios. Portanto, em Java, pode-se dizer que a classe `Retângulo` herda da classe `Quadrilátero`. Nesse contexto, a classe `Quadrilátero` é uma superclasse e a classe `Retângulo` é uma subclasse. Um retângulo *é um* tipo específico de quadrilátero, mas é incorreto afirmar que cada quadrilátero *é um* retângulo — o

quadrilátero poderia ser um paralelogramo ou alguma outra forma. A Figura 9.1 lista vários exemplos simples de superclasses e subclasses — observe que as superclasses tendem a ser "mais gerais" e as subclasses, "mais específicas".

Como cada objeto de subclasse *é um* objeto de sua superclasse e uma superclasse pode ter muitas subclasses, o conjunto de objetos representado por uma superclasse é, em geral, maior que o conjunto de objetos representado por qualquer uma de suas subclasses. Por exemplo, a superclasse `Vehicle` representa todos os veículos, incluindo carros, caminhões, barcos, bicicletas e assim por diante. Por contraste, a subclasse `Car` representa um subconjunto de veículos menor mais específico.

Superclasse	Subclasses
`Aluno`	`AlunoDeGraduação, AlunoDePósGraduação`
`Forma`	`Círculo, Triângulo, Retângulo, Esfera, Cubo`
`Financiamento`	`FinanciamentoDeCarro, FinanciamentoDeCasa`
`Empregado`	`CorpoDocente, Funcionário`
`ContaBancária`	`ContaCorrente, ContaPoupança`

Figura 9.1 | Exemplos de herança.

As relações de herança formam estruturas hierárquicas parecidas com uma árvore. Uma superclasse existe em um relacionamento hierárquico com suas subclasses. Vamos desenvolver uma hierarquia de classe de exemplo (Figura 9.2), também chamada de **hierarquia de herança**. Uma comunidade universitária tem milhares de membros, incluindo empregados, alunos e graduados. Os empregados incluem o corpo docente e os funcionários operacionais. Os membros da faculdade são administradores (por exemplo, diretores e chefes de departamento) ou professores. Observe que a hierarquia poderia conter muitas outras classes. Por exemplo, alunos podem ser graduados ou graduandos. Os graduandos podem ser primeiranistas, segundanistas, terceiranistas ou quartanistas.

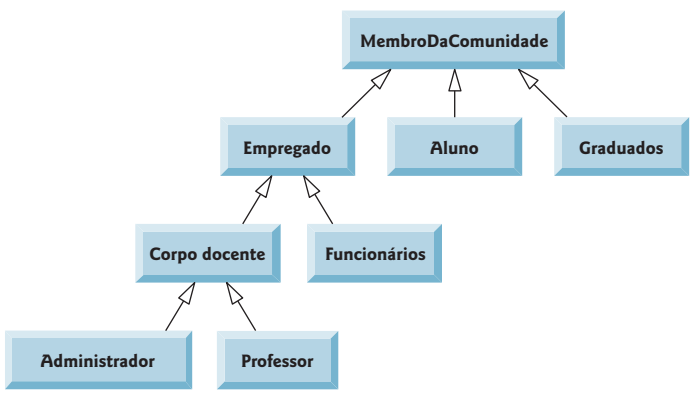


Figura 9.2 | Hierarquia de herança `MembrosDaComunidade` da universidade.

Cada seta na hierarquia representa um relacionamento *é um*. À medida que seguimos as setas para cima nessa hierarquia de classes, podemos declarar, por exemplo, que "um `Empregado` *é um* `MembroDaComunidade`" e "um `Professor` *é um* membro do `CorpoDocente`". `MembroDaComunidade` é a superclasse direta de `Empregado`, `Aluno` e `Graduados` e é uma superclasse indireta de todas as outras classes no diagrama. A partir da parte inferior, você pode seguir as setas e aplicar um relacionamento *é um* até a superclasse na parte superior. Por exemplo, um `Administrador` *é um* membro `CorpoDocente`, *é um* `Empregado`, *é um* `MembroDaComunidade` e, naturalmente, *é um* `Objeto`.

Agora considere a hierarquia de herança `Shape` na Figura 9.3. Essa hierarquia inicia com a superclasse `Forma`, que é estendida pelas subclasses `FormaBiDimensional` e `FormaTriDimensional` — `Forma` é `FormaBiDimensional` ou `FormaBiDimensional`. O terceiro nível dessa hierarquia contém tipos específicos de `FormaBiDimensional` e `FormaTriDimensional`. Como na Figura 9.2, podemos seguir as setas a partir da parte inferior do diagrama até a superclasse na parte superior nessa hierarquia de classes para identificar vários relacionamentos *é um*. Por exemplo, um `Triângulo` *é uma* `FormaBiDimensional` e *é uma* `Forma`, enquanto uma `Esfera` *é uma* `FormaBiDimensional` e *é uma* `Forma`. Observe que essa hierarquia poderia conter muitas outras classes. Por exemplo, elipses e trapezoides são `FormaBidimensionais`.

Nem todo relacionamento de classe é um relacionamento de herança. No Capítulo 8, discutimos o relacionamento *tem um*, em que as classes têm membros que são referências a objetos de outras classes. Esses relacionamentos criam classes compondo classes existentes. Por exemplo, considerando as classes `Empregado`, `DataDeNascimento` e `NúmeroDeTelefone`, não é correto dizer que um `Empregado` *é uma* `DataDeNascimento` ou que um `Empregado` *é um* `NúmeroDeTelefone`. Entretanto, um `Empregado` *tem uma* `DataDeNascimento` e um `Empregado` *tem um* `NúmeroDeTelefone`.

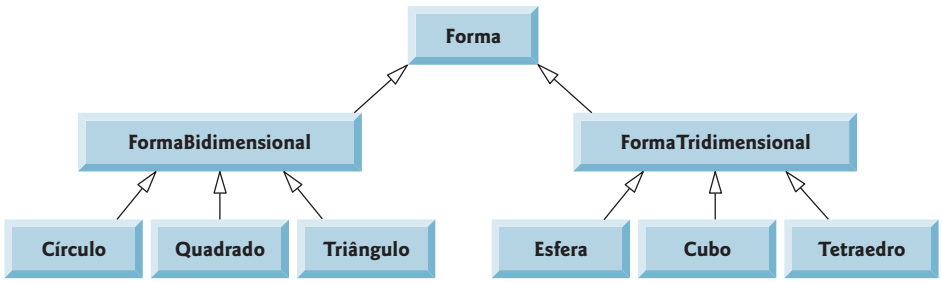


Figura 9.3 | Hierarquia de herança para `Formas`.

É possível tratar objetos de superclasse e objetos de subclasse de maneira semelhante — seus aspectos comuns são expressos nos membros da superclasse. Os objetos de todas as classes que estendem uma superclasse comum podem ser tratados como objetos dessa superclasse (isto é, esses objetos têm um relacionamento *é um* com a superclasse). Mais adiante neste capítulo e no Capítulo 10, "Programação orientada a objetos: polimorfismo", consideramos muitos exemplos que tiram proveito do relacionamento *é um*.

Um problema com herança é que uma subclasse pode herdar métodos de que não necessita ou que não deveria ter. Mesmo quando um método de superclasse é adequado a uma subclasse, essa subclasse precisa frequentemente de uma versão personalizada do método. Nesses casos, a subclasse pode **sobrescrever** (redefinir) o método de superclasse com uma implementação adequada, como veremos com frequência nos exemplos de código do capítulo.

9.3 Membros `protected`

O Capítulo 8 discutiu modificadores de acesso `public` e `private`. Os membros `public` de uma classe são acessíveis onde quer que o programa tenha uma referência a um objeto dessa classe ou uma de suas subclasses. Os membros `private` de uma classe só são acessíveis dentro da própria classe. Nesta seção, introduzimos o modificador de acesso **`protected`**. Utilizar acesso `protected` oferece um nível intermediário de acesso entre `public` e `private`. Os membros `protected` de uma superclasse podem ser acessados por membros dessa superclasse, por membros de suas subclasses e por membros de outras classes no mesmo pacote — membros `protected` também têm acesso de pacote.

Todos os membros de superclasse `public` e `protected` retêm seu modificador de acesso original quando se tornam membros da subclasse — membros `public` da superclasse tornam-se membros `public` da subclasse e membros `protected` da superclasse tornam-se membros `protected` da subclasse. Os membros da superclasse `private` não são acessíveis fora da própria classe. Em vez disso, eles permanecem ocultos nas suas subclasses e só podem ser acessados pelos métodos `public` ou `protected` herdados da superclasse.

Os métodos de subclasse podem referir-se a membros `public` e `protected` herdados da superclasse simplesmente utilizando os nomes de membro. Quando um método de subclasse sobrescrever um método de superclasse herdado, o método de superclasse pode ser acessado a partir da subclasse precedendo o nome do método de superclasse com a palavra-chave **`super`** e um separador de ponto (`.`). Discutimos o acesso a membros sobrescritos da superclasse na Seção 9.4.

Observação de engenharia de software 9.1

Os métodos de uma subclasse não acessam membros `private` diretamente de sua superclasse. Uma subclasse pode alterar o estado de variáveis de instância `private` da superclasse somente por meio de métodos não `private` fornecidos na superclasse e herdados pela subclasse.

Observação de engenharia de software 9.2

Declarar variáveis de instância `private` ajuda-lhe a testar, depurar e modificar sistemas corretamente. Se uma subclasse pudesse acessar variáveis de instância `private` da sua superclasse, classes que herdam dessa subclasse também poderiam acessar as variáveis de instância. Isso propagaria acesso ao que devem ser variáveis de instância `private` e os benefícios do ocultamento de informações seriam perdidos.

9.4 Relacionamento entre superclasses e subclasses

Agora utilizamos uma hierarquia de herança que contém tipos de empregados no aplicativo de folha de pagamento de uma empresa para discutir o relacionamento entre uma superclasse e sua subclasse. Nessa empresa, os empregados comissionados (que serão representados como objetos de uma superclasse) recebem uma porcentagem de suas vendas, enquanto empregados comissionados com salário-base (que serão representados como objetos de uma subclasse) recebem um salário-base *mais* uma porcentagem de suas vendas.

Dividimos nossa discussão do relacionamento entre essas classes em cinco exemplos. O primeiro exemplo declara a classe `CommissionEmployee`, que herda diretamente da classe `Object` e declara como variáveis de instância `private` o nome, sobrenome, CIC, taxa de comissão e quantidade de vendas brutas (isto é, o total).

O segundo exemplo declara a classe `BasePlusCommissionEmployee`, que também herda diretamente da classe `Object` e declara como variáveis de instância `private` um nome, sobrenome, CIC, taxa de comissão, quantidade bruta de vendas e salário-base. Criamos essa classe escrevendo cada linha de código que a classe exige — logo veremos que é muito mais eficiente criá-la herdando da classe `CommissionEmployee`.

O terceiro exemplo declara uma nova classe `BasePlusCommissionEmployee` que estende a classe `CommissionEmployee` (isto é, uma `BasePlusCommissionEmployee` *é uma* `CommissionEmployee` quem também tem um salário-base). Essa reutilização de software permite *escrever menos código* ao desenvolver a nova subclasse. Nesse exemplo, a classe `BasePlusCommissionEmployee` tenta acessar os membros `private` da classe `CommissionEmployee` — isso resulta em erros de compilação, porque a subclasse não pode acessar as variáveis de instância da superclasse `private`.

O quarto exemplo mostra que, se as variáveis de instância `CommissionEmployee` forem declaradas como `protected`, a subclasse `BasePlusCommissionEmployee` poderá acessar esses dados diretamente. As classes `BasePlusCommissionEmployee` contêm funcionalidades idênticas, mas mostramos como a versão herdada é mais fácil de criar e gerenciar.

Depois de discutirmos a conveniência de utilizar as variáveis de instância `protected`, criamos o quinto exemplo, que configura as variáveis de instância `CommissionEmployee` novamente como `private` para impor boa engenharia de software. Em seguida, mostramos como a subclasse `BasePlusCommissionEmployee` pode utilizar métodos `public` de `CommissionEmployee` para manipular (de uma maneira controlada) as variáveis de instância `private` herdadas de `CommissionEmployee`.

9.4.1 Criando e utilizando uma classe `CommissionEmployee`

Começamos declarando a classe `CommissionEmployee` (Figura 9.4). A linha 4 inicia a declaração de classe e indica que a classe `CommissionEmployee` **`estende`** (isto é, herda) a classe **`Object`** (do pacote `java.lang`). Isso faz a classe `CommissionEmployee` herdar os métodos `Object` da classe — a classe `Object` não tem nenhum campo. Se você não especificar explicitamente qual classe uma nova classe estende, a classe estenderá `Object` implicitamente. Por essa razão, normalmente você não incluirá "`extends Object`" em seu código — fazemos isso nesse exemplo somente para propósitos de demonstração.

```
1  // Figura 9.4: CommissionEmployee.java
2  // Classe CommissionEmployee representa um funcionário que recebeu uma
3  // porcentagem das vendas brutas.
4  public class CommissionEmployee extends Object
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // vendas brutas semanais
10    private double commissionRate; // porcentagem da comissão
11
12    // construtor de cinco argumentos
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // chamada implícita para o construtor Object ocorre aqui
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // valida e armazena as vendas brutas
21       setCommissionRate( rate ); // valida e armazena a taxa de comissão
22    } // fim do construtor CommissionEmployee de cinco argumentos
23
24    // configura o nome
25    public void setFirstName( String first )
26    {
27       firstName = first; // deve validar
28    } // fim do método setFirstName
29
30    // retorna o nome
31    public String getFirstName()
32    {
33       return firstName;
34    } // fim do método getFirstName
35
36    // configura o sobrenome
```

```
37       public void setLastName( String last )
38       {
39          lastName = last; // deve validar
40       } // fim do método setLastName
41
42       // retorna o sobrenome
43       public String getLastName()
44       {
45          return lastName;
46       } // fim do método getLastName
47
48       // configura o CIC
49       public void setSocialSecurityNumber( String ssn )
50       {
51          socialSecurityNumber = ssn; // deve validar
52       } // fim do método setSocialSecurityNumber
53
54       // retorna o CIC
55       public String getSocialSecurityNumber()
56       {
57          return socialSecurityNumber;
58       } // fim do método getSocialSecurityNumber
59
60       // configura a quantidade de vendas brutas
61       public void setGrossSales( double sales )
62       {
63          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64       } // fim do método setGrossSales
65
66       // retorna a quantidade de vendas brutas
67       public double getGrossSales()
68       {
69          return grossSales;
70       } // fim do método getGrossSales
71
72       // configura a taxa de comissão
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // fim do método setCommissionRate
77
78       // retorna a taxa de comissão
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // fim do método getCommissionRate
83
84       // calcula os lucros
85       public double earnings()
86       {
87          return commissionRate * grossSales;
88       } // fim do método earnings
89
90       // retorna a representação String do objeto CommissionEmployee
91       @Override // indica que esse método sobrescreve um método de superclasse
92       public String toString()
93       {
94          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95             "commission employee", firstName, lastName,
96             "social security number", socialSecurityNumber,
97             "gross sales", grossSales,
98             "commission rate", commissionRate );
99       } // fim do método toString
100   } // fim da classe CommissionEmployee
```

Figura 9.4 | A classe `CommissionEmployee` representa um empregado pago com uma porcentagem das vendas brutas.

Visão geral dos métodos e variáveis de instância da classe `CommissionEmployee`

Os serviços `public` da classe `CommissionEmployee` incluem um construtor (linhas 13–22) e métodos `earnings` (linhas 85–88) e `toString` (linhas 91–99). As linhas 25–82 declaram métodos *get* e *set* `public` para as variáveis de instância da classe (declaradas nas linhas 6–10) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`. A classe declara suas variáveis de instância como `private`, então os objetos de outras classes não podem acessar essas variáveis diretamente. Declarar variáveis de instância como `private` e fornecer os métodos *get* e *set* para as manipular e validar ajuda a impor boa engenharia de software. Os métodos `setGrossSales` e `setCommissionRate`, por exemplo, validam seus argumentos antes de atribuir os valores às variáveis de instância `grossSales` e `commissionRate`. Em um aplicativo de negócios crítico do mundo real, também realizaríamos a validação nos outros métodos *set* da classe.

O construtor da classe `CommissionEmployee`

Os construtores não são herdados, então a classe `CommissionEmployee` não herda o construtor da classe `Object`. Mas os construtores de uma superclasse ainda estão disponíveis para subclasses. De fato, a primeira tarefa de qualquer construtor de subclasse é chamar o construtor de sua superclasse direta, explícita ou implicitamente (se nenhuma chamada de construtor for especificada), para assegurar que as variáveis de instância herdadas da superclasse são inicializadas adequadamente. Nesse exemplo, o construtor da classe `CommissionEmployee` chama o construtor da classe `Object` implicitamente. A sintaxe para chamar um construtor de superclasse é explicitamente discutida na Seção 9.4.3. Se o código não incluir uma chamada explícita para o construtor de superclasse, o Java chama implicitamente o construtor padrão ou sem argumento da superclasse. O comentário na linha 16 da Figura 9.4 indica onde a chamada implícita para o construtor padrão da superclasse `Object` é feita (você não escreve o código dessa chamada). O construtor padrão de `Object` (vazio) não faz nada. Observe que mesmo se uma classe não tiver construtores, o construtor padrão que o compilador declara implicitamente para a classe chamará o construtor padrão ou sem argumento da superclasse.

Depois da chamada implícita para o construtor de `Object`, as linhas 17–21 do construtor de `CommissionEmployee` atribui valores às variáveis de instância da classe. Observe que não validamos os valores de argumentos `first`, `last` e `ssn` antes de atribuí-los às variáveis de instância correspondentes. Poderíamos validar o nome e o sobrenome — talvez para assegurar que eles tenham um comprimento razoável. De maneira semelhante, um CIC poderia ser validado para assegurar que ele contém nove dígitos, com ou sem traços (por exemplo, `123-45-6789` ou `123456789`).

Método `earnings` da classe `CommissionEmployee`

O método `earnings` (linhas 85–88) calcula os ganhos de um `CommissionEmployee`. A linha 87 multiplica `commissionRate` pelo `grossSales` e retorna o resultado.

A anotação `@Override` e o método `toString` da classe `CommissionEmployee`

O método `toString` (linhas 91–99) é especial — é um dos métodos que toda classe herda direta ou indiretamente da classe `Object` (resumida na Seção 9.7). O método `toString` retorna uma `String` representando um objeto. Ele é chamado implicitamente sempre que um objeto precisa ser convertido em uma representação `String` — por exemplo, quando um objeto é impresso pelo método `printf` ou `String format` utilizando o especificador de formato `%s`. O método `toString` da classe `Object` retorna uma `String` que inclui o nome da classe do objeto. Ele é principalmente um marcador de lugar que pode ser sobrescrito por uma subclasse para especificar uma adequada representação `String` dos dados em um objeto de subclasse. O método `toString` da classe `CommissionEmployee` sobrescreve (redefine) o método `toString` da classe `Object`. Quando invocado, o método `toString` de `CommissionEmployee` utiliza o método `String format` para retornar uma `String` que contém informações sobre o `CommissionEmployee`. Para sobrescrever um método de superclasse, uma subclasse deve declarar um método com a mesma assinatura (nome de método, número de parâmetros, tipos de parâmetro e ordem dos tipos de parâmetro) como o método de superclasse — o método `toString` de `Object` não aceita nenhum parâmetro, portanto `CommissionEmployee` declara `toString` sem parâmetros.

A linha 91 utiliza a **notação `@Override`** para indicar se o método `toString` deve sobrescrever um método de superclasse. Anotações têm várias finalidades. Por exemplo, ao tentar sobrescrever um método de superclasse, erros comuns incluem nomear o método da subclasse incorretamente, ou utilizar o número ou tipos incorretos dos parâmetros na lista de parâmetros. Cada um desses problemas cria uma sobrecarga não intencional do método de uma superclasse. Se você então tentar chamar o método em um objeto de uma subclasse, a versão da superclasse será invocada e a versão da subclasse não é ignorada — potencialmente levando a erros de lógica sutis. Quando o compilador encontra um método declarado com `@Override`, ele compara a assinatura do método com as assinaturas de método da superclasse. Se não houver uma correspondência exata, o compilador emite uma mensagem de erro, como "method does not override or implement a method from a supertype" (método não sobrescreve ou implementa um método a partir de um supertipo). Isso indica que você sobrecarregou acidentalmente um método de superclasse. Você pode então corrigir a assinatura do método para que ela corresponda àquela na superclasse.

Como veremos ao discutir aplicativos Web e serviços Web nos capítulos 29 a 31, anotações também podem adicionar código de suporte complexo às suas classes para simplificar o processo de desenvolvimento e podem ser utilizadas pelos servidores para configurar certos aspectos dos aplicativos Web.

Erro comum de programação 9.1

Utilizar uma assinatura de método incorreta ao tentar sobrescrever um método de superclasse resulta em uma sobrecarga de método não intencional que pode levar a erros de lógica sutis.

Erro comum de programação 9.2

É um erro de sintaxe sobrescrever um método com um modificador de acesso mais restrito — um método `public` da superclasse não pode tornar-se um método `protected` ou `private` na subclasse; um método `protected` da superclasse não pode tornar-se um método `private` na subclasse. Fazer isso quebraria o relacionamento é um *em que se exige que todos os objetos de subclasse sejam capazes de responder a chamadas de método que são feitas para os métodos `public` declarados na superclasse. Se um método `public`, por exemplo, pudesse ser sobrescrito como um método `protected` ou `private`, os objetos de subclasse não seriam capazes de responder às mesmas chamadas de método como objetos de superclasse. Uma vez que um método é declarado `public` em uma superclasse, o método permanece `public` para todas as subclasses diretas e indiretas da classe.*

Dica de prevenção de erro 9.1

Declare métodos sobrescritos com a notação `@Override` para assegurar em tempo de compilação que você definiu as assinaturas corretamente. Sempre é melhor localizar erros em tempo de compilação em vez de em tempo de execução.

Classe `CommissionEmployeeTest`

A Figura 9.5 testa a classe `CommissionEmployee`. As linhas 9–10 instanciam um objeto `CommissionEmployee` e invocam o construtor de `CommissionEmployee` (linhas 13–22 da Figura 9.4) para inicializá-lo com `"Sue"` como o nome, `"Jones"` como o sobrenome, `"222-22-2222"` como o cadastro de pessoas físicas, `10000` como o valor das vendas brutas e `.06` como a porcentagem de comissão. As linhas 15–24 usam métodos *get* de `CommissionEmployee` para recuperar os valores de variável de instância do objeto para saída. As linhas 26–27 invocam os métodos `setGrossSales` e `setCommissionRate` do objeto para alterar os valores de variáveis de instância `grossSales` e `commissionRate`. As linhas 29–30 geram saída da representação de `String` da `CommissionEmployee` atualizada. Observe que quando um objeto é enviado para a saída utilizando o especificador de formato `%s`, o método `toString` do objeto é invocado implicitamente para obter a representação da `String` do objeto. [*Nota:* neste capítulo, não utilizamos os métodos `earnings` das nossas classes; mas esses métodos serão utilizados extensivamente no Capítulo 10.]

```
1  // Figura 9.5: CommissionEmployeeTest.java
2  // Programa de teste da classe CommissionEmployee.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instancia o objeto CommissionEmployee
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // obtém os dados de empregado comissionado
13       System.out.println(
14          "Employee information obtained by get methods: \n" );
15       System.out.printf( "%s %s\n", "First name is",
16          employee.getFirstName() );
17       System.out.printf( "%s %s\n", "Last name is",
18          employee.getLastName() );
19       System.out.printf( "%s %s\n", "Social security number is",
20          employee.getSocialSecurityNumber() );
21       System.out.printf( "%s %.2f\n", "Gross sales is",
22          employee.getGrossSales() );
23       System.out.printf( "%s %.2f\n", "Commission rate is",
24          employee.getCommissionRate() );
25
26       employee.setGrossSales( 500 ); // configura vendas brutas
27       employee.setCommissionRate( .1 ); // configura a taxa de comissão
28
29       System.out.printf( "\n%s:\n\n%s\n",
30          "Updated employee information obtained by toString", employee );
31    } // fim de main
32 } // fim da classe CommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Figura 9.5 | Programa de teste da classe `CommissionEmployee`.

9.4.2 Criando e utilizando uma classe `BasePlusCommissionEmployee`

Agora discutimos a segunda parte de nossa introdução à herança, declarando e testando a classe `BasePlusCommissionEmployee` (uma completamente nova e independente) (Figura 9.6), que contém o nome, sobrenome, CIC, quantidade de vendas brutas, taxa de comissão *e* salário-base. Os serviços `public` da classe `BasePlusCommissionEmployee` incluem um construtor `BasePlusCommissionEmployee` (linhas 15–25) e os métodos `earnings` (linhas 100–103) e `toString` (linhas 106–115). As linhas 28–97 declaram os métodos `public` *get* e *set* para as variáveis de instância da `private` classe (declaradas nas linhas 7–12) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` *e* `baseSalary`. Essas variáveis e métodos encapsulam todos os recursos necessários de um empregado comissionado com salário-base. Observe a semelhança entre essa classe e a classe `CommissionEmployee` (Figura 9.4) — nesse exemplo, ainda não exploraremos essa semelhança.

```
1  // Figura 9.6: BaseplusCommissionEmployee.java
2  // A classe BasePlusCommissionEmployee representa um empregado que
3  // recebe um salário-base além da comissão.
4
5  public class BasePlusCommissionEmployee
6  {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // vendas brutas semanais
11    private double commissionRate; // porcentagem da comissão
12    private double baseSalary; // salário-base por semana
13
14    // construtor de seis argumentos
15    public BasePlusCommissionEmployee( String first, String last,
16       String ssn, double sales, double rate, double salary )
17    {
18       // chamada implícita para o construtor Object ocorre aqui
19       firstName = first;
20       lastName = last;
21       socialSecurityNumber = ssn;
22       setGrossSales( sales ); // valida e armazena as vendas brutas
23       setCommissionRate( rate ); // valida e armazena a taxa de comissão
24       setBaseSalary( salary ); // valida e armazena salário-base
25    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
26
27    // configura o nome
28    public void setFirstName( String first )
29    {
30       firstName = first; // deve validar
31    } // fim do método setFirstName
32
33    // retorna o nome
34    public String getFirstName()
35    {
36       return firstName;
37    } // fim do método getFirstName
38
39    // configura o sobrenome
```

```
40       public void setLastName( String last )
41       {
42          lastName = last; // deve validar
43       } // fim do método setLastName
44
45       // retorna o sobrenome
46       public String getLastName()
47       {
48          return lastName;
49       } // fim do método getLastName
50
51       // configura o CIC
52       public void setSocialSecurityNumber( String ssn )
53       {
54          socialSecurityNumber = ssn; // deve validar
55       } // fim do método setSocialSecurityNumber
56
57       // retorna o CIC
58       public String getSocialSecurityNumber()
59       {
60          return socialSecurityNumber;
61       } // fim do método getSocialSecurityNumber
62
63       // configura a quantidade de vendas brutas
64       public void setGrossSales( double sales )
65       {
66          grossSales = ( sales < 0.0 ) ? 0.0 : sales;
67       } // fim do método setGrossSales
68
69       // retorna a quantidade de vendas brutas
70       public double getGrossSales()
71       {
72          return grossSales;
73       } // fim do método getGrossSales
74
75       // configura a taxa de comissão
76       public void setCommissionRate( double rate )
77       {
78          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
79       } // fim do método setCommissionRate
80
81       // retorna a taxa de comissão
82       public double getCommissionRate()
83       {
84          return commissionRate;
85       } // fim do método getCommissionRate
86
87       // configura o salário-base
88       public void setBaseSalary( double salary )
89       {
90          baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
91       } // fim do método setBaseSalary
92
93       // retorna o salário-base
94       public double getBaseSalary()
95       {
96          return baseSalary;
97       } // fim do método getBaseSalary
98
99       // calcula os lucros
100      public double earnings()
101      {
102         return baseSalary + ( commissionRate * grossSales );
103      } // fim do método earnings
104
105      // retorna a representação de String de BasePlusCommissionEmployee
106      @Override // indica que esse método sobrescreve um método e superclasse
107      public String toString()
108      {
109         return String.format(
110            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f ",
```

```
111             "base-salaried commission employee", firstName, lastName,
112             "social security number", socialSecurityNumber,
113             "gross sales", grossSales, "commission rate", commissionRate,
114             "base salary", baseSalary );
115       } // fim do método toString
116    } // fim da classe BasePlusCommissionEmployee
```

Figura 9.6 | A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão.

Observe que a classe `BasePlusCommissionEmployee` não especifica "`extends Object`" na linha 5, então a classe estende `Object` implicitamente. Além disso, como o construtor da classe `CommissionEmployee` (linhas 13–22 da Figura 9.4), o construtor da classe `BasePlusCommissionEmployee` invoca o construtor padrão da classe `Object` implicitamente, como observado no comentário da linha 18.

O método `earnings` da classe `BasePlusCommissionEmployee` (linhas 100–103) retorna o resultado da adição do salário-base de `BasePlusCommissionEmployee` ao produto da porcentagem de comissão e as vendas brutas do funcionário.

A classe `BasePlusCommissionEmployee` sobrescreve o método `Object toString` para retornar uma `String` contendo informações de `BasePlusCommissionEmployee`. Mais uma vez, utilizamos o especificador de formato `%.2f` para formatar as vendas brutas, taxa de comissão e salário-base com dois dígitos de precisão à direita do ponto de fração decimal (linha 110).

Testando a classe `BasePlusCommissionEmployee`

A Figura 9.7 testa a classe `BasePlusCommissionEmployee`. As linhas 9–11 instanciam um objeto `BasePlusCommissionEmployee` e passam `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` e `300` para o construtor como o nome, sobrenome, CPF, vendas brutas, taxa de comissão e salário-base, respectivamente. As linhas 16–27 utilizam os métodos *get* `BasePlusCommissionEmployee` para recuperar os valores das variáveis de instância do objeto para saída. A linha 29 invoca o objeto método `setBaseSalary` para alterar o salário-base. O método `setBaseSalary` (Figura 9.6, linhas 88–91) assegura que não foi atribuído um valor negativo à variável de instância `baseSalary`, porque o salário-base de um empregado não pode ser negativo. As linhas 31–33 da Figura 9.7 invocam o método `toString` explicitamente para obter a representação `String` do objeto.

```
1  // Figura 9.7: BasePlusCommissionEmployeeTest.java
2  // Programa de teste de BasePlusCommissionEmployee.
3
4  public class BasePlusCommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instancia o objeto BasePlusCommissionEmployee
9        BasePlusCommissionEmployee employee =
10          new BasePlusCommissionEmployee(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // obtém os dados do empregado comissionado com salário-base
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
24       System.out.printf( "%s %.2f\n", "Commission rate is",
25          employee.getCommissionRate() );
26       System.out.printf( "%s %.2f\n", "Base salary is",
27          employee.getBaseSalary() );
28
29       employee.setBaseSalary( 1000 ); // configura o salário-base
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obtained by toString",
33           employee.toString() );
34    } // fim de main
35 } // fim da classe BasePlusCommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Figura 9.7 | Programa de teste de `BasePlusCommissionEmployee`.

Notas sobre a classe `BasePlusCommissionEmployee`

A maior parte do código da classe `BasePlusCommissionEmployee` (Figura 9.6) é semelhante, ou idêntica, à da classe `CommissionEmployee` (Figura 9.4). Por exemplo, as variáveis de instância `private` `firstName` e `lastName` e os métodos `setFirstName`, `getFirstName`, `setLastName` e `getLastName` são idênticos àqueles da classe `CommissionEmployee`. As duas classes também contêm as variáveis de instância `private` `socialSecurityNumber`, `commissionRate` e `grossSales` e métodos *get* e *set* correspondentes. Além disso, o construtor `BasePlusCommissionEmployee` é quase idêntico àquele da classe `CommissionEmployee`, exceto que o construtor de `BasePlusCommissionEmployee` também configura o `baseSalary`. Outras adições à classe `BasePlusCommissionEmployee` são as variáveis de instância `private` `baseSalary` e os métodos `setBaseSalary` e `getBaseSalary`. O método `toString` da classe `BasePlusCommissionEmployee` é quase idêntico àquele da classe `CommissionEmployee`, exceto que `toString` de `BasePlusCommissionEmployee` também gera saída da variável de instância `baseSalary` com dois dígitos de precisão à direita do ponto de fração decimal.

Copiamos literalmente o código de classe `CommissionEmployee` e o *colamos* na classe `BasePlusCommissionEmployee`, então modificamos a classe `BasePlusCommissionEmployee` para incluir um salário-base e métodos que manipulam o salário-base. Essa abordagem "copiar e colar" é frequentemente propensa a erro e consome tempo. Pior ainda, ela espalha cópias do mesmo código por todo o sistema, criando um pesadelo para a manutenção de código. Há alguma maneira de "absorver" as variáveis de instância e os métodos de uma classe de modo a torná-los parte de outras classes sem duplicar código? Em seguida respondemos a essa pergunta, utilizando uma abordagem mais elegante para criar classes que enfatiza os benefícios da herança.

Observação de engenharia de software 9.3

Com a herança, as variáveis de instância comuns e os métodos de todas as classes na hierarquia são declarados em uma superclasse. Quando são feitas modificações nessas características comuns na superclasse, as subclasses herdam, portanto, as modificações. Sem a herança, as alterações precisariam ser feitas em todos os arquivos de código-fonte que contêm uma cópia do código em questão.

9.4.3 Criando uma hierarquia de herança `CommissionEmployee`–`BasePlusCommissionEmployee`

Agora declaramos uma nova classe `BasePlusCommissionEmployee` (Figura 9.8), que estende a classe `CommissionEmployee` (Figura 9.4). Um objeto `BasePlusCommissionEmployee` *é um* `CommissionEmployee`, porque a herança transmite as capacidades da classe `CommissionEmployee`. A classe `BasePlusCommissionEmployee` também tem a variável de instância `baseSalary` (Figura 9.8, linha 6). A palavra-chave `extends` (linha 4) indica a herança. A subclasse `BasePlusCommissionEmployee` herda os métodos e as variáveis de instância de `CommissionEmployee` — lembre-se de que somente os membros `public` e `protected` da superclasse são diretamente acessíveis na subclasse. O construtor `CommissionEmployee` não é herdado. Desse modo, os serviços `public` `BasePlusCommissionEmployee` incluem seu construtor (linhas 9–16), métodos `public` herdados de `CommissionEmployee`, o método `setBaseSalary` (linhas 19–22), o método `getBaseSalary` (linhas 25–28), o método `earnings` (linhas 31–36) e o método `toString` (linhas 39–49). Os métodos `earnings` e `toString` sobrescrevem os métodos correspondentes na classe `CommissionEmployee` porque as versões de superclasse desses métodos não calculam corretamente os ganhos de um `BasePlusCommissionEmployee` nem retornam uma representação `String` apropriada.

```
1  // Figura 9.8: BaseplusCommissionEmployee.java
2  // membros private da superclasse não podem ser acessados em uma subclasse.
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // salário-base por semana
```

```
7
8       // construtor de seis argumentos
9       public BasePlusCommissionEmployee( String first, String last,
10         String ssn, double sales, double rate, double salary )
11      {
12         // chamada explícita para o construtor CommissionEmployee da superclasse
13         super( first, last, ssn, sales, rate );
14
15         setBaseSalary( salary ); // valida e armazena salário-base
16      } // fim do construtor BasePlusCommissionEmployee de seis argumentos
17
18      // configura o salário-base
19      public void setBaseSalary( double salary )
20      {
21         baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22      } // fim do método setBaseSalary
23
24      // retorna o salário-base
25      public double getBaseSalary()
26      {
27         return baseSalary;
28      } // fim do método getBaseSalary
29
30      // calcula os lucros
31      @Override // indica que esse método sobrescreve um método de superclasse
32      public double earnings()
33      {
34         // não permitido: commissionRate e grossSales private em superclasse
35         return baseSalary + ( commissionRate * grossSales );
36      } // fim do método earnings
37
38      // retorna a representação de String de BasePlusCommissionEmployee
39      @Override // indica que esse método sobrescreve um método de superclasse
40      public String toString()
41      {
42         // não permitido: tenta acessar membros private da superclasse
43         return String.format(
44            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
45            "base-salaried commission employee", firstName, lastName,
46            "social security number", socialSecurityNumber,
47            "gross saless", grossSales, "commission rate", commissionRate,
48            "base salary", baseSalary );
49      } // fim do método toString
50   } // fim da classe BasePlusCommissionEmployee
```

```
BasePlusCommissionEmployee.java:35: commissionRate has private access in CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee.java:35: grossSales has private access in CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:45: firstName has private access in CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee.java:45: lastName has private access in CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee.java:46: socialSecurityNumber has private access in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:47: grossSales has private access in CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:47: commissionRate has private access in CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                      ^
7 errors
```

Figura 9.8 | Os membros `private` da superclasse não podem ser acessados em uma subclasse.

Cada construtor de subclasse deve chamar implícita ou explicitamente seu construtor de superclasse para inicializar as variáveis de instância herdadas da superclasse. A linha 13 no construtor de seis argumentos de `BasePlusCommissionEmployee` (linhas 9–16) chama explicitamente o construtor de cinco argumentos da classe `CommissionEmployee` (declarada nas linhas 13–22 da Figura 9.4) para inicializar a parte da superclasse de um objeto `BasePlusCommissionEmployee` (isto é, variáveis `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`). Fazemos isso utilizando a **sintaxe de chamada de construtor de superclasse** — a palavra-chave `super`, seguida pelo conjunto de parênteses contendo os argumentos do construtor da superclasse. Os argumentos `first`, `last`, `ssn`, `sales` e `rate` são utilizados para inicializar os membros da superclasse `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`, respectivamente. Se o construtor de `BasePlusCommissionEmployee` não invocasse o construtor da superclasse explicitamente, o Java tentaria invocar o construtor sem argumentos ou construtor padrão da superclasse. A classe `CommissionEmployee` não tem esse construtor, portanto o compilador emitiria um erro. A chamada de construtor de superclasse explícita na linha 13 da Figura 9.8 deve ser a primeira instrução no corpo do construtor de subclasse. Quando uma superclasse contiver um construtor sem argumento, você pode utilizar `super()` para chamar esse construtor explicitamente, mas isso raramente é feito.

Erro comum de programação 9.3

Um erro de compilação ocorre se um construtor de subclasse chamar um construtor de superclasse com argumentos que não coincidem com o número e os tipos de parâmetro em um dos construtores da superclasse.

O compilador gera erros para a linha 35 da Figura 9.8 porque as variáveis de instância `commissionRate` e `grossSales` da superclasse `CommissionEmployee` são `private` — os métodos da subclasse `BasePlusCommissionEmployee` não podem acessar as variáveis de instância `private` da superclasse `CommissionEmployee`. Observe que utilizamos texto vermelho na Figura 9.8 para indicar o código incorreto. O compilador emite erros adicionais nas linhas 45–47 do método `toString` de `BasePlusCommissionEmployee` pela mesma razão. Os erros em `BasePlusCommissionEmployee` podem ter sido evitados utilizando os métodos *get* herdados da classe `CommissionEmployee`. Por exemplo, a linha 35 poderia ter utilizado `getCommissionRate` e `getGrossSales` para acessar as variáveis de instância `private` `commissionRate` e `grossSales` de `CommissionEmployee`, respectivamente. As linhas 45–47 também poderiam ter utilizado os métodos *get* adequados para recuperar os valores das variáveis de instância da superclasse.

9.4.4 Hierarquia de herança `CommissionEmployee`–`BasePlusCommissionEmployee` utilizando variáveis de instância `protected`

Para permitir que a classe `BasePlusCommissionEmployee` acesse diretamente as variáveis de instância de superclasse `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`, podemos declarar esses membros como `protected` na superclasse. Como discutimos na Seção 9.3, os membros `protected` de uma superclasse *são* acessíveis por todas as subclasses dessa superclasse. Na nova classe `CommissionEmployee`, modificamos somente as linhas 6–10 para declarar as variáveis de instância com o modificador de acesso `protected` como a seguir:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales; // vendas brutas semanais
protected double commissionRate; // porcentagem da comissão
```

O restante da declaração de classe (que não é mostrado aqui) é idêntico àquele da Figura 9.4.

Poderíamos ter declarado as variáveis de instância `public` de `CommissionEmployee` para permitir que a subclasse `BasePlusCommissionEmployee` pudesse acessá-las. Entretanto, declarar as variáveis de instância `public` é uma prática de engenharia de software pobre porque permite acesso irrestrito às variáveis de instância, aumentando significativamente a chance de erros. Com variáveis de instância `protected`, a subclasse obtém acesso às variáveis de instância, mas as classes que não são subclasses e as que não estão no mesmo pacote não podem acessar essas variáveis diretamente — lembre-se de que os membros de classe `protected` também são visíveis para outras classes no mesmo pacote.

Classe `BasePlusCommissionEmployee`

A classe `BasePlusCommissionEmployee` (Figura 9.9) estende a nova versão da classe `CommissionEmployee` com variáveis de instância `protected`. Objetos `BasePlusCommissionEmployee` herdam as variáveis de instância `protected` `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` de `CommissionEmployee` — todas essas variáveis são agora membros `protected` de `BasePlusCommissionEmployee`. Como resultado, o compilador não gera erros ao compilar a linha 33 do método `earnings` e as linhas 42–44 do método `toString`. Se outra classe estender essa versão da classe `BasePlusCommissionEmployee`, a nova subclasse também poderá acessar membros `protected`.

```
1  // Figura 9.9: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee herda a instância protected
3  // variáveis de CommissionEmployee.
4
5  public class BasePlusCommissionEmployee extends CommissionEmployee
6  {
7     private double baseSalary; // salário-base por semana
8
9     // construtor de seis argumentos
10    public BasePlusCommissionEmployee( String first, String last,
11       String ssn, double sales, double rate, double salary )
12    {
13       super( first, last, ssn, sales, rate );
14       setBaseSalary( salary ); // valida e armazena salário-base
15    } // fim do construtor BasePlusCommissionEmployee de seis argumentos
16
17    // configura o salário-base
18    public void setBaseSalary( double salary )
19    {
20       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21    } // fim do método setBaseSalary
22
23    // retorna o salário-base
24    public double getBaseSalary()
25    {
26       return baseSalary;
27    } // fim do método getBaseSalary
28
29    // calcula os lucros
30    @Override // indica que esse método sobrescreve um método de superclasse
31    public double earnings()
32    {
33       return baseSalary + ( commissionRate * grossSales );
34    } // fim do método earnings
35
36    // retorna a representação de String de BasePlusCommissionEmployee
37    @Override // indica que esse método sobrescreve um método de superclasse
38    public String toString()
39    {
40       return String.format(
41          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
42          "base-salaried commission employee", firstName, lastName,
43          "social security number", socialSecurityNumber,
44          "gross sales", grossSales, "commission rate", commissionRate,
45          "base salary", baseSalary );
46    } // fim do método toString
47 } // fim da classe BasePlusCommissionEmployee
```

Figura 9.9 | `BasePlusCommissionEmployee` herda as variáveis de instância `protected` de `CommissionEmployee`.

Ao criar um objeto `BasePlusCommissionEmployee`, ele contém todas as variáveis de instância declaradas na hierarquia de classes até esse ponto — isto é, aquelas das classes `Object`, `CommissionEmployee` e `BasePlusCommissionEmployee`. A classe `BasePlusCommissionEmployee` não herda o construtor da classe `CommissionEmployee`. Mas o construtor de seis argumentos da classe `BasePlusCommissionEmployee` (linhas 10–15) chama o construtor de cinco argumentos da classe `CommissionEmployee` explicitamente para inicializar as variáveis de instância que `BasePlusCommissionEmployee` herdou da classe `CommissionEmployee`. De maneira semelhante, o construtor da classe `CommissionEmployee` chama o construtor da classe `Objeto`. O construtor de `BasePlusCommissionEmployee` deve fazer isso explicitamente porque `CommissionEmployee` não fornece um construtor sem argumentos que pode ser invocado implicitamente.

Testando a classe `BasePlusCommissionEmployee`

A classe `BasePlusCommissionEmployeeTest` desse exemplo é idêntica àquela da Figura 9.7 e produz a mesma saída, portanto não a mostramos aqui. Embora a versão da classe `BasePlusCommissionEmployee` na Figura 9.6 não utilize a herança e a versão na Figura 9.9 utilize, *ambas as classes fornecem a mesma funcionalidade*. O código-fonte na Figura 9.9 (47 linhas) é consideravelmente mais curto

do que aquele na Figura 9.6 (116 linhas), porque a maioria das funcionalidades de `BasePlusCommissionEmployee` agora é herdada de `CommissionEmployee` — há agora apenas uma cópia da funcionalidade `CommissionEmployee`. Isso facilita a manutenção, modificação de depuração do código, porque o código relacionado a um empregado comissionado só existe na classe `CommissionEmployee`.

Notas sobre o uso de variáveis de instância `protected`

Nesse exemplo, declaramos as variáveis de instância de superclasse como `protected` para que as subclasses pudessem acessá-las. Herdar as variáveis de instância `protected` aumenta ligeiramente o desempenho, porque podemos acessar diretamente as variáveis na subclasse sem estar sujeitos ao overhead de uma chamada de método *set* ou *get*. Na maioria dos casos, porém, é melhor utilizar as variáveis de instância `private` para incentivar a engenharia de software adequada e deixar as questões de otimização de código para o compilador. Seu código será mais fácil de manter, modificar e depurar.

Utilizar variáveis de instância `protected` cria vários problemas potenciais. Primeiro, o objeto de subclasse pode configurar o valor de uma variável herdada diretamente sem utilizar um método *set*. Portanto, um objeto de subclasse pode atribuir um valor inválido à variável, possivelmente deixando o objeto em um estado inconsistente. Por exemplo, se fôssemos declarar a variável de instância de `CommissionEmployee` `grossSales` como `protected`, um objeto de subclasse (por exemplo, `BasePlusCommissionEmployee`) então poderia atribuir um valor negativo a `grossSales`. Outro problema em utilizar as variáveis de instância `protected` é que é mais provável que os métodos de subclasse sejam escritos para dependerem da implementação de dados da superclasse. Na prática, as subclasses devem depender somente dos serviços de superclasse (isto é, métodos não `private`) e não da implementação de dados de superclasse. Com as variáveis de instância `protected` na superclasse, podemos precisar modificar todas as subclasses da superclasse se a implementação de superclasse mudar. Por exemplo, se por alguma razão tivéssemos de mudar os nomes de variáveis de instância `firstName` e `lastName` para `first` e `last`, depois teríamos de fazer isso em todas as ocorrências em que uma subclasse referencia diretamente as variáveis de instância de superclasse `firstName` e `lastName`. Nesse caso, diz-se que o software é **frágil**, porque uma pequena alteração na superclasse pode "quebrar" a implementação da subclasse. Você deve ser capaz de alterar a implementação de superclasse ao mesmo tempo em que ainda fornece os mesmos serviços às subclasses. Naturalmente, se os serviços de superclasse mudam, devemos reimplementar nossas subclasses. Um terceiro problema é que os membros `protected` de uma classe são visíveis a todas as classes no mesmo pacote que a classe que contém os membros `protected` — isso nem sempre é desejável.

Observação de engenharia de software 9.4

Utilize o modificador de acesso `protected` quando uma superclasse precisar fornecer um método somente para suas subclasses e outras classes no mesmo pacote, mas não para outros clientes.

Observação de engenharia de software 9.5

Declarar as variáveis de instância da superclasse `private` (em oposição a `protected`) permite a implementação de superclasse dessas variáveis de instância para alterar sem afetar as implementações de subclasse.

Dica de prevenção de erro 9.2

Quando possível, não inclua variáveis de instância `protected` em uma superclasse. Em vez disso, inclua métodos não `private` que acessam as variáveis de instância `private`. Isso ajudará a assegurar que os objetos da classe mantenham estados consistentes.

9.4.5 Hierarquia de herança `CommissionEmployee`–`BasePlusCommissionEmployee` utilizando variáveis de instância `private`

Vamos reexaminar a nossa hierarquia, dessa vez utilizando boas práticas de engenharia de software. A classe `CommissionEmployee` (Figura 9.10) declara as variáveis de instância `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` como `private` (linhas 6–10) e fornece os métodos `public` `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` e `toString` para manipular esses valores. Observe que os métodos `earnings` (linhas 85–88) e `toString` (linhas 91–99) utilizam os métodos *get* da classe para obter os valores de suas variáveis de instância. Se decidirmos alterar os nomes da variável de instância, as declarações `earnings` e `toString` não exigirão modificação — somente os corpos do método *get* e *set* que manipulam diretamente as variáveis de instância precisarão mudar. Observe que essas modificações só ocorrem dentro da superclasse — nenhuma modificação para a subclasse é necessária. Localizar os efeitos de alterações como essa é uma boa prática de engenharia de software. A subclasse `BasePlusCommissionEmployee` (Figura 9.11) herda os métodos não `private` de `CommissionEmployee` e pode acessar os membros de superclasse `private` via esses métodos.

```
1  // Figura 9.10: CommissionEmployee.java
2  // A classe CommissionEmployee utiliza métodos para manipular suas
3  // variáveis de instância private.
4  public class CommissionEmployee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // vendas brutas semanais
10    private double commissionRate; // porcentagem da comissão
11
12    // construtor de cinco argumentos
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // chamada implícita para o construtor Object ocorre aqui
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // valida e armazena as vendas brutas
21       setCommissionRate( rate ); // valida e armazena a taxa de comissão
22    } // fim do construtor CommissionEmployee de cinco argumentos
23
24    // configura o nome
25    public void setFirstName( String first )
26    {
27       firstName = first; // deve validar
28    } // fim do método setFirstName
29
30    // retorna o nome
31    public String getFirstName()
32    {
33       return firstName;
34    } // fim do método getFirstName
35
36    // configura o sobrenome
37    public void setLastName( String last )
38    {
39       lastName = last; // deve validar
40    } // fim do método setLastName
41
42    // retorna o sobrenome
43    public String getLastName()
44    {
45       return lastName;
46    } // fim do método getLastName
47
48    // configura o CIC
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // deve validar
52    } // fim do método setSocialSecurityNumber
53
54    // retorna o CIC
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // fim do método getSocialSecurityNumber
59
60    // configura a quantidade de vendas brutas
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // fim do método setGrossSales
65
66    // retorna a quantidade de vendas brutas
67    public double getGrossSales()
68    {
```

```
69          return grossSales;
70       } // fim do método getGrossSales
71
72       // configura a taxa de comissão
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // fim do método setCommissionRate
77
78       // retorna a taxa de comissão
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // fim do método getCommissionRate
83
84       // calcula os lucros
85       public double earnings()
86       {
87          return getCommissionRate() * getGrossSales();
88       } // fim do método earnings
89
90       // retorna a representação String do objeto CommissionEmployee
91       @Override // indica que esse método sobrescreve um método de superclasse
92       public String toString()
93       {
94          return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95             "commission employee", getFirstName(),getLastName(),
96             "social security number", getSocialSecurityNumber(),
97             "gross sales", getGrossSales(),
98             "commission rate", getCommissionRate() );
99       } // fim do método toString
100   } // fim da classe CommissionEmployee
```

Figura 9.10 | A classe `CommissionEmployee` utiliza métodos para manipular suas variáveis de instância `private`.

A classe `BasePlusCommissionEmployee` (Figura 9.11) tem várias modificações que a distinguem da Figura 9.9. Os métodos `earnings` (Figura 9.11, linhas 31–35) e `toString` (linhas 38–43) invocam o método `getBaseSalary` para obter o valor do salário-base, em vez de acessar `baseSalary` diretamente. Se decidirmos renomear a variável de instância `baseSalary`, somente os corpos do método `setBaseSalary` e `getBaseSalary` precisarão mudar.

```
1    // Figura 9.11: BaseplusCommissionEmployee.java
2    // A classe BasePlusCommissionEmployee herda de CommissionEmployee
3    // e acessa os dados private da superclasse via
4    // métodos public herdados.
5
6    public class BasePlusCommissionEmployee extends CommissionEmployee
7    {
8       private double baseSalary; // salário-base por semana
9
10      // construtor de seis argumentos
11      public BasePlusCommissionEmployee( String first, String last,
12         String ssn, double sales, double rate, double salary )
13      {
14         super( first, last, ssn, sales, rate );
15         setBaseSalary( salary ); // valida e armazena salário-base
16      } // fim do construtor BasePlusCommissionEmployee de seis argumentos
17
18      // configura o salário-base
19      public void setBaseSalary( double salary )
20      {
21         baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22      } // fim do método setBaseSalary
23
24      // retorna o salário-base
```

```
25      public double getBaseSalary()
26      {
27         return baseSalary;
28      } // fim do método getBaseSalary
29
30      // calcula os lucros
31      @Override // indica que esse método sobrescreve um método de superclasse
32      public double earnings()
33      {
34         return getBaseSalary() + super.earnings();
35      } // fim do método earnings
36
37      // retorna a representação de String de BasePlusCommissionEmployee
38      @Override // indica que esse método sobrescreve um método de superclasse
39      public String toString()
40      {
41         return String.format( "%s %s\n%s: %.2f", "base-salaried",
42            super.toString(), "base salary", getBaseSalary() );
43      } // fim do método toString
44   } // fim da classe BasePlusCommissionEmployee
```

Figura 9.11 | A classe `BasePlusCommissionEmployee` herda de `CommissionEmployee` e acessa os dados `private` da superclasse via métodos `public` herdados.

Método `earnings` da classe `BasePlusCommissionEmployee`

O método `earnings` (Figura 9.11, linhas 31–35) sobrescreve o método `earnings` da classe `CommissionEmployee` (Figura 9.10, linhas 85–88) para calcular os ganhos das comissões de um funcionário com salário-base. A nova versão obtém a parte dos ganhos com base apenas na comissão chamando o método `earnings` de `CommissionEmployee` com `super.earnings()` (Figura 9.11, linha 34). O método `earnings` de `BasePlusCommissionEmployee` então adiciona o salário-base a esse valor para calcular os vencimentos totais. Note a sintaxe usada para invocar um método de superclasse a partir de uma subclasse — coloque a palavra-chave `super` e um ponto separador (`.`) antes do nome de método de superclasse. Essa invocação de método é uma boa prática de engenharia de software — se um método realizar todas ou algumas das ações necessárias por outro método, chame esse método em vez de duplicar seu código. Fazendo o método `earnings` de `BasePlusCommissionEmployee` invocar o método `earnings` de `CommissionEmployee` para calcular parte dos ganhos de um objeto `BasePlusCommissionEmployee`, evitamos duplicar o código e reduzimos problemas de manutenção de código. Se não utilizássemos "`super.`", o método `earnings` de `BasePlusCommissionEmployee` chamaria a si mesmo em vez da versão de superclasse. Isso resultaria em um fenômeno que estudamos no Capítulo 18 chamado recursão infinita, o que, por fim, faria a pilha de chamadas de método estourar — um erro em tempo de execução fatal.

Erro comum de programação 9.4

Quando um método de superclasse é sobrescrito em uma subclasse, a versão de subclasse frequentemente chama a versão de superclasse para fazer uma parte do trabalho. A falha em prefixar o nome do método da superclasse com a palavra-chave `super` e um ponto separador (`.`) ao chamar o método da superclasse faz com que o método de subclasse chame a si próprio, criando potencialmente um erro chamado recursão infinita. A recursão, utilizada corretamente, é uma capacidade poderosa discutida no Capítulo 18, "Recursão".

Método `toString` da classe `BasePlusCommissionEmployee`

De maneira semelhante, o método `toString` de `BasePlusCommissionEmployee` (Figura 9.11, linhas 38–43) sobrescreve o método `toString` da classe `CommissionEmployee` (Figura 9.10, linhas 91–99) para retornar uma representação de `String` que é adequada ao empregado comissionado com salário-base. A nova versão cria parte de uma representação de `String` do objeto `BasePlusCommissionEmployee` (isto é, a `String` `"commission employee"` e os valores das variáveis de instância `private` da classe `CommissionEmployee`) chamando o método `toString` de `CommissionEmployee` com a expressão `super.toString()` (Figura 9.11, linha 42). O método `toString` de `BasePlusCommissionEmployee` então gera saída do restante da representação de `String` de um objeto `BasePlusCommissionEmployee` (isto é, o valor do salário-base da classe `BasePlusCommissionEmployee`).

Testando a classe `BasePlusCommissionEmployee`

A classe `BasePlusCommissionEmployeeTest` realiza as mesmas manipulações em um objeto `BasePlusCommissionEmployee` como na Figura 9.7 e produz a mesma saída, portanto não a mostramos aqui. Embora cada classe `BasePlusCommissionEmployee` que

vimos se comporte de uma maneira idêntica, a versão na Figura 9.11 é mais bem projetada. Utilizando a herança e chamando os métodos que ocultam os dados e asseguram a consistência, criamos eficiente e efetivamente uma classe bem-projetada.

Resumo dos exemplos de herança nas seções 9.4.1–9.4.5

Vimos agora o conjunto de exemplos que foram projetados para ensinar a boa engenharia de software com a herança. Você empregou a palavra-chave `extends` para criar uma subclasse utilizando a herança, usou membros `protected` de superclasse para permitir que uma subclasse acesse variáveis de instância de superclasse herdadas e sobrescreveu os métodos da superclasse para fornecer versões que são mais apropriadas para objetos de subclasse. Além disso, você aplicou técnicas de engenharia de software apresentadas no Capítulo 8 e neste capítulo para criar classes que são fáceis de manter, modificar e depurar.

9.5 Construtores em subclasses

Como explicamos na seção anterior, instanciar um objeto de subclasse inicia uma cadeia de chamadas de construtor em que o construtor de subclasse, antes de realizar suas próprias tarefas, invoca o construtor de sua superclasse direta explicitamenta via referência `super` ou implicitamente chamando o construtor padrão ou construtor sem argumento da superclasse. De modo semelhante, se a superclasse é derivada de outra classe — como é, naturalmente, toda e qualquer classe, exceto `Object` — o construtor de superclasse invoca o construtor da próxima classe no topo da hierarquia e assim por diante. O último construtor chamado na cadeia é *sempre* o construtor da classe `Object`. O corpo do construtor de subclasse original termina a execução por último. O construtor de cada superclasse manipula as variáveis de instância de superclasse que o objeto de subclasse herda. Por exemplo, considere novamente a hierarquia `CommissionEmployee--BasePlusCommissionEmployee` das figuras 9.10 e 9.11. Quando um programa cria um objeto `BasePlusCommissionEmployee`, seu construtor é chamado. Esse chama o construtor de `CommissionEmployee`, que por sua vez chama o construtor de `Object`. O construtor da classe `Object` tem um corpo vazio, então, ele retorna imediatamente o controle para o construtor de `CommissionEmployee`, que, em seguida, inicializa as variáveis de instância `CommissionEmployee` `private` que são parte do objeto `BasePlusCommissionEmployee`. Quando o construtor de `CommissionEmployee` completar a execução, ele retorna o controle ao construtor de `BasePlusCommissionEmployee`, que inicializa o `baseSalary` do objeto `BasePlusCommissionEmployee`.

Observação de engenharia de software 9.6

Quando um programa cria um objeto de subclasse, o construtor de subclasse imediatamente chama o construtor de superclasse (explicitamente, via `super`, ou implicitamente). O corpo do construtor de superclasse executa para inicializar as variáveis de instância da superclasse que fazem parte do objeto de subclasse, então o corpo do construtor de subclasse executa para inicializar variáveis de instância somente de subclasse. O Java assegura que mesmo se um construtor não atribuir um valor a uma variável de instância, a variável ainda é inicializada como seu valor padrão (por exemplo, `0` para tipos numéricos primitivos, `false` para `boolean`s, `null` para referências).

9.6 Engenharia de software com herança

Ao estender uma classe, a nova classe herda os membros da superclasse — embora os membros `private` da superclasse permaneçam ocultos na nova classe. Você pode personalizar a nova classe para atender suas necessidades incluindo membros adicionais e sobrescrevendo membros de superclasse. Fazer isso não requer que o programador da subclasse altere o (ou mesmo tenha acesso ao) código-fonte da superclasse. O Java simplesmente requer acesso ao arquivo `.class` da superclasse para que possa compilar e executar qualquer programa que utilize ou estenda a superclasse. Essa poderosa capacidade é atraente para fornecedores de software independentes (Independent Software Vendors — ISVs), que podem desenvolver classes proprietárias (patenteadas) para a venda ou licença e disponibilizá-las para os usuários em formato de bytecode. Os usuários podem então derivar novas classes dessas classes de biblioteca rapidamente e sem acessar o código-fonte proprietário (patenteado) dos ISVs.

Observação de engenharia de software 9.7

Embora herdar de uma classe não requeira acesso ao código-fonte da classe, os desenvolvedores frequentemente insistem em examinar o código-fonte para entender como a classe é implementada. Os desenvolvedores na indústria querem assegurar que eles estejam estendendo uma classe sólida — por exemplo, uma classe que executa bem e é implementada de maneira firme e segura.

Às vezes é difícil apreciar o escopo dos problemas enfrentados por projetistas que trabalham em projetos de software de larga escala. Pessoas experientes nesses projetos dizem que a reutilização eficaz de software melhora o processo de desenvolvimento de software. A programação orientada a objetos facilita a reutilização de software, muitas vezes diminuindo significativamente o tempo de desenvolvimento.

A disponibilidade de bibliotecas de classes substanciais e úteis fornece os benefícios máximos de reutilização de software por meio da herança. As bibliotecas de classes Java padrão que são distribuídas com o Java tendem a ter uma finalidade mais ou menos geral, encorajando uma ampla reutilização de softwares. Há muitas outras bibliotecas de classes.

Ler as declarações de subclasse pode ser confuso, porque os membros herdados não são declarados explicitamente nas subclasses, mas estão presentes nelas. Há um problema semelhante em documentar membros de subclasse.

Observação de engenharia de software 9.8

Na etapa do design de um sistema orientado a objetos, você frequentemente descobrirá que certas classes estão intimamente relacionadas. Você deve "fatorar" as variáveis de instância e métodos comuns e colocá-los em uma superclasse. Então, deve utilizar a herança para desenvolver subclasses, especializando-as com capacidades além daquelas herdadas da superclasse.

Observação de engenharia de software 9.9

Declarar uma subclasse não afeta o código-fonte da sua superclasse. A herança preserva a integridade da superclasse.

Observação de engenharia de software 9.10

Assim como os projetistas de sistemas não orientados a objetos devem evitar a proliferação de métodos, os projetistas de sistemas orientados a objetos devem evitar a proliferação de classes. Essa proliferação cria problemas de gerenciamento e pode prejudicar a capacidade de reutilização de software, porque em uma enorme biblioteca de classes torna-se difícil localizar as classes mais apropriadas. A alternativa é criar menos classes que fornecem funcionalidades mais substanciais, mas isso pode se tornar complicado.

Dica de desempenho 9.1

Se as subclasses são maiores do que precisam ser (isto é, contêm funcionalidades demais), recursos de memória e de processamento podem ser desperdiçados. Estenda a superclasse que contém a funcionalidade mais próxima daquilo de que você precisa.

9.7 Classe `Object`

Como discutimos anteriormente neste capítulo, todas as classes no Java herdam, ou estendem, direta ou indiretamente da classe `Object` (pacote `java.lang`), portanto seus 11 métodos são herdados, ou estendidos, por todas as outras classes. A Figura 9.12 resume os métodos de `Object`.

Método	Descrição
`clone`	Esse método `protected`, que não aceita nenhum argumento e retorna uma referência `Object`, faz uma cópia do objeto em que é chamado. A implementação padrão realiza a chamada **cópia superficial** — os valores da variável de instância em um objeto são copiados em outro objeto do mesmo tipo. Para tipos por referência, apenas as referências são copiadas. Uma típica implementação do método `clone` sobrescrito realizaria uma **cópia em profundidade** que cria um novo objeto para cada variável de instância de tipo por referência. É difícil implementar `clone` corretamente. Por isso, seu uso não é recomendável. Muitos especialistas do setor sugerem utilizar a serialização de objetos em seu lugar. Discutimos a serialização de objetos no Capítulo 17, "Arquivos, fluxos e serialização de objetos".
`equals`	Esse método compara dois objetos quanto à igualdade e retorna `true` se eles forem iguais ou, caso contrário, `false`. O método aceita qualquer `Object` como um argumento. Quando os objetos de uma classe particular precisam ser comparados quanto à igualdade, a classe deve sobrescrever o método `equals` para comparar o *conteúdo* dos dois objetos. Para os requisitos da implementação desse método, consulte a documentação do método em `java.sun.com/javase/6/docs/api/java/lang/Object.html# equals(java.lang.Object)`. A implementação padrão de `equals` usa o operador == para determinar se duas referências *referenciam o mesmo objeto* na memória. A Seção 16.3.3 demonstra o método `equals` da classe `String` e diferencia entre comparar objetos `String` com == e com `equals`.
`finalize`	Esse método `protected` (introduzido na Seção 8.10) é chamado pelo coletor de lixo para realizar a limpeza de término em um objeto antes de o coletor de lixo reivindicar a memória do objeto. Lembre-se de que não é claro se, ou quando, o método `finalize` será chamado. Por essa razão, a maioria dos programadores deve evitar o método `finalize`.
`getClass`	Todo objeto no Java conhece seu próprio tipo em tempo de execução. O método `getClass` (utilizado nas seções 10.5, 14.5 e 24.3) retorna um objeto de classe `Class` (pacote `java.lang`) que contém as informações sobre o tipo de objeto, como seu nome de classe (retornado pelo método `Class` `getName`). Para obter informações adicionais sobre a classe `Class`, visite `java.sun.com/javase/6/docs/api/java/lang/Class.html`.

Método	Descrição
`hashCode`	Códigos de hash são valores `int` que são úteis para armazenamento e recuperação de alta velocidade das informações armazenadas em uma estrutura de dados que é conhecida como uma tabela de hash (discutida na Seção 20.11). Esse método também é chamado como parte da implementação padrão do método `toString` da classe `Object`.
`wait`, `notify`, `notifyAll`	Os métodos `notify`, `notifyAll` e três versões sobrecarregadas de `wait` estão relacionados ao multithreading, discutido no Capítulo 26.
`toString`	Esse método (introduzido na Seção 9.4.1) retorna uma representação `String` de um objeto. A implementação padrão desse método retorna o nome de pacote e o nome de classe da classe do objeto seguido por uma representação hexadecimal do valor retornado pelo método `hashCode` do objeto.

Figura 9.12 | Métodos de `Object`.

Discutimos vários métodos de `Object` por todo este livro (como indicado na Figura 9.12). Você pode aprender mais sobre os métodos de `Object` na documentação on-line de API de `Object` e no *The Java tutorial* nos seguintes sites:

```
java.sun.com/javase/6/docs/api/java/lang/Object.html
java.sun.com/docs/books/tutorial/java/IandI/objectclass.html
```

A partir do Capítulo 7, lembre-se de que os arrays são objetos. Como resultado, como todos os outros objetos, os arrays herdam os membros da classe `Object`. Observe que todo array tem um método `clone` sobrescrito que copia o array. Mas, se o array armazenar referências a objetos, os objetos não serão copiados — uma cópia superficial é realizada. Para informações adicionais sobre o relacionamento entre arrays e a classe `Object`, consulte *Java language specification,* Capítulo 10, em

```
java.sun.com/docs/books/jls/third_edition/html/arrays.html
```

9.8 (Opcional) Estudo de caso de GUI e imagens gráficas: exibindo texto e imagens com rótulos

Os programas frequentemente utilizam rótulos quando precisam exibir informações ou instruções para o usuário em uma interface gráfica com o usuário. **Rótulos** são um modo conveniente de identificar componentes GUI na tela e manter o usuário informado sobre o estado atual do programa. Em Java, um objeto da classe **`JLabel`** (do pacote `javax.swing`) pode exibir texto, imagem ou ambos. O exemplo na Figura 9.13 demonstra vários recursos `JLabel`, incluindo um rótulo de texto sem formatação, um rótulo de imagem e um rótulo tanto com texto como com uma imagem.

```
1  // Figura 9.13: LabelDemo.java
2  // Demonstra o uso de rótulos.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
8  public class LabelDemo
9  {
10    public static void main( String[] args )
11    {
12       // Cria um rótulo com texto simples
13       JLabel northLabel = new JLabel( "North" );
14
15       // cria um ícone de uma imagem para podermos colocar em um JLabel
16       ImageIcon labelIcon = new ImageIcon( "GUItip.gif" );
17
18       // cria um rótulo com um Icon em vez de texto
19       JLabel centerLabel = new JLabel( labelIcon );
20
21       // cria outro rótulo com um Icon
22       JLabel southLabel = new JLabel( labelIcon );
23
24       // configura o rótulo para exibir texto (bem como um ícone)
```

```
25          southLabel.setText( "South" );
26
27          // cria um frame para armazenar os rótulos
28          JFrame application = new JFrame();
29
30          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32          // adiciona os rótulos ao frame; o segundo argumento especifica
33          // onde adicionar o rótulo no frame
34          application.add( northLabel, BorderLayout.NORTH );
35          application.add( centerLabel, BorderLayout.CENTER );
36          application.add( southLabel, BorderLayout.SOUTH );
37
38          application.setSize( 300, 300 ); // configura o tamanho do frame
39          application.setVisible( true ); // mostra o frame
40       } // fim de main
41    } // fim da classe LabelDemo
```

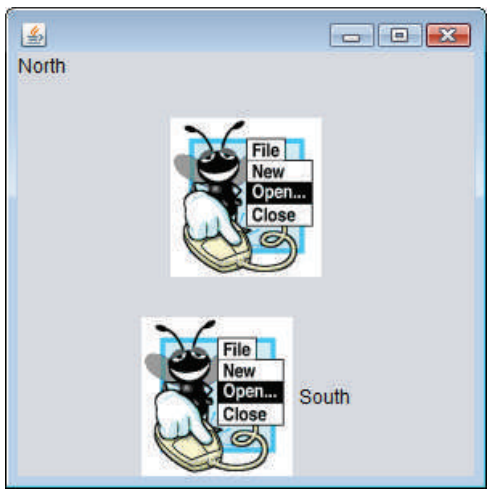


Figura 9.13 | `JLabel` com texto e imagens.

As linhas 3–6 importam as classes necessárias para exibir `JLabels`. `BorderLayout` do pacote `java.awt` contém constantes que especificam onde podemos colocar componentes GUI no `JFrame`. A classe **`ImageIcon`** representa uma imagem que pode ser exibida em um `JLabel`, e a classe `JFrame` representa a janela que conterá todos os rótulos.

A linha 13 cria um `JLabel` que exibe seu argumento de construtor — a string `"North"`. A linha 16 declara a variável local `labelIcon` e atribui a ela um novo `ImageIcon`. O construtor para `ImageIcon` recebe uma `String` que especifica o caminho para a imagem. Visto que especificamos somente um nome de arquivo, o Java assume que ele está no mesmo diretório que a classe `LabelDemo`. `ImageIcon` pode carregar imagens em formatos de imagem GIF, JPEG e PNG. A linha 19 declara e inicializa a variável local `centerLabel` com um `JLabel` que exibe o `labelIcon`. A linha 22 declara e inicializa a variável local `southLabel` com um `JLabel` semelhante ao que aparece na linha 19. Entretanto, a linha 25 chama o método **`setText`** para alterar o texto ao exibir o rótulo. O método `setText` pode ser chamado em qualquer `JLabel` para alterar seu texto. Esse `JLabel` exibe tanto o ícone como o texto.

A linha 28 cria o `JFrame` que exibe os `JLabels` e a linha 30 indica que o programa deve terminar quando o `JFrame` for fechado. Anexamos os rótulos ao `JFrame` nas linhas 34–36 chamando uma versão sobrecarregada do método `add` que aceita dois parâmetros. O primeiro parâmetro é o componente que queremos anexar e o segundo é a região em que ele deve ser colocado. Todo `JFrame` tem um **layout** associado que ajuda o `JFrame` a posicionar os componentes GUI que são anexados a ele. O layout padrão do `JFrame` é conhecido como um **`BorderLayout`** e tem cinco regiões — `NORTH` (parte superior), `SOUTH` (parte inferior), `EAST` (lado direito), `WEST` (lado esquerdo) e `CENTER`. Cada uma dessas regiões é declarada como uma constante na classe `BorderLayout`. Ao chamar o método `add` com um argumento, o `JFrame` coloca o componente automaticamente no `CENTER`. Se uma posição já contém um componente, então o novo componente ocupa seu lugar. As linhas 38 e 39 configuram o tamanho do `JFrame` e o tornam visível na tela.

Exercício do Estudo de caso GUI e imagens gráficas

9.1 Modifique o Exercício 8.1 do Estudo de caso de GUI e imagens gráficas para incluir um `JLabel` como uma barra de status que exibe contagens para representar o número de cada forma exibida. A classe `DrawPanel` deve declarar um método que retorna uma `String` que contém o texto de status. Em `main`, crie primeiro o `DrawPanel`, depois crie o `JLabel` com o texto de status como um argumento para o construtor de `JLabel`. Anexe o `JLabel` à região `SOUTH` do `JFrame`, como mostrado na Figura 9.14.

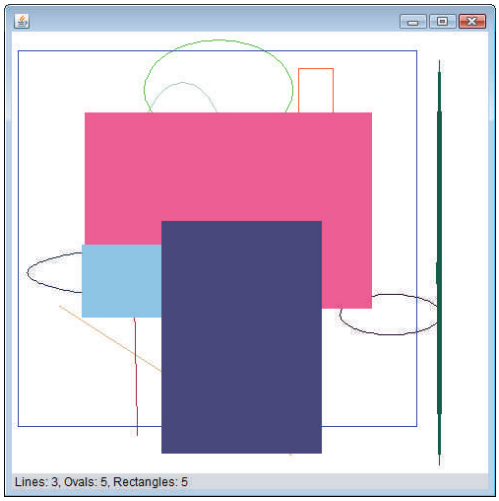


Figura 9.14 | `JLabel` exibindo a estatística de forma.

9.9 Conclusão

Este capítulo introduziu a herança — a capacidade de criar classes absorvendo membros de uma classe existente e aprimorando essas classes com novas capacidades. Você aprendeu as noções de superclasses e subclasses e utilizou a palavra-chave `extends` para criar uma subclasse que herda os membros de uma superclasse. Mostramos como usar a notação `@Override` para evitar sobrecarga não intencional indicando que um método sobrecarrega um método de superclasse. Introduzimos o modificador de acesso `protected`; os métodos de subclasse podem acessar diretamente os membros da superclasse `protected`. Você aprendeu como utilizar `super` para acessar membros da superclasse sobrescritos. Você também viu como os construtores são utilizados em hierarquias de herança. Por fim, você aprendeu os métodos da classe `Object`, a superclasse direta ou indireta de todas as classes Java.

No Capítulo 10, "Programação orientada a objetos: polimorfismo", avançamos nossa discussão de herança introduzindo polimorfismo — um conceito orientado a objetos que permite escrever programas que tratam convenientemente, de uma maneira mais geral, objetos de uma ampla variedade de classes relacionadas por herança. Depois de estudar o Capítulo 10, você estará familiarizado com classes, objetos, encapsulamento, herança e polimorfismo — as tecnologias-chave da programação orientada a objetos.

Resumo

Seção 9.1 Introdução

- A herança reduz o tempo de desenvolvimento do programa.
- A superclasse direta de uma subclasse (especificada pela palavra-chave `extends` na primeira linha de uma declaração de classe) é a superclasse a partir da qual a subclasse herda. A superclasse indireta de uma subclasse está dois ou mais níveis acima da hierarquia de classe dessa subclasse.
- Em herança única, uma classe é derivada de uma superclasse direta. Na herança múltipla, uma classe é derivada de mais de uma superclasse direta. O Java não suporta herança múltipla.
- Uma subclasse é mais específica que sua superclasse e representa um grupo menor de objetos.
- Cada objeto de uma subclasse também é um objeto da superclasse dessa classe. Entretanto, um objeto de superclasse não é um objeto de subclasses da sua classe.
- Um relacionamento *é um* representa herança. Em um relacionamento *é um*, um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse.
- Um relacionamento *tem um* representa a composição. Em um relacionamento *tem um*, um objeto de classe contém referências a objetos de outras classes.

Seção 9.2 Superclasses e subclasses

- Os relacionamentos de herança simples formam estruturas hierárquicas do tipo árvore — há uma superclasse em um relacionamento hierárquico com suas subclasses.

Seção 9.3 Membros `protected`

- Os membros `public` de uma superclasse são acessíveis onde quer que o programa tenha uma referência a um objeto dessa superclasse ou para uma de suas subclasses.
- Os membros de uma superclasse `private` só podem ser acessados diretamente a partir de dentro da declaração da superclasse.
- Os membros `protected` de uma superclasse têm um nível intermediário de proteção entre acesso `public` e `private`. Eles podem ser acessados por membros da superclasse, por membros de suas subclasses e por membros de outras classes no mesmo pacote.
- Os membros `private` de uma superclasse permanecem ocultos nas suas subclasses e só podem ser acessados por meio dos métodos `public` ou `protected` herdados da superclasse.
- Quando um método de subclasse sobrescrever um método de superclasse, o método de superclasse pode ser acessado a partir da subclasse se o nome de método de superclasse for precedido por `super` e um ponto separador (`.`).

Seção 9.4 Relacionamento entre superclasses e subclasses

- Uma subclasse não pode acessar os membros `private` de sua superclasse — permitir isso violaria o encapsulamento da superclasse. Uma subclasse pode, porém, acessar os membros não `private` de sua superclasse.
- Uma subclasse pode invocar um construtor da sua superclasse utilizando a palavra-chave `super`, seguida pelo conjunto de parênteses contendo os argumentos do construtor de superclasse. Isso deve aparecer como a primeira instrução no corpo do construtor da subclasse.
- Um método de superclasse pode ser sobrescrito em uma subclasse para declarar uma implementação apropriada para a subclasse.
- A notação `@Override` indica que um método deve sobrescrever um método de superclasse. Quando o compilador encontrar um método declarado com `@Override`, ele comparará a assinatura do método com as assinaturas de método da superclasse. Se não houver uma correspondência exata, o compilador emite uma mensagem de erro, como "method does not override or implement a method from a supertype" (método não sobrescreve ou implementa um método a partir de um supertipo).
- O método `toString` não recebe nenhum argumento e retorna uma `String`. O método `toString` da classe `Object` normalmente é sobrescrito por uma subclasse.
- Quando um objeto é enviado para saída utilizando o especificador de formato `%s`, o método `toString` do objeto é chamado implicitamente para obter sua representação de `String`.

Seção 9.5 Construtores em subclasses

- A primeira tarefa de qualquer construtor de subclasse é chamar o construtor de sua superclasse direta, explícita ou implicitamente, para assegurar que as variáveis de instância herdadas da superclasse são inicializadas.

Seção 9.6 Engenharia de software com herança

- Declarar variáveis de instância `private`, ao fornecer métodos não `private` para manipular e realizar a validação, ajuda a impor boa engenharia de software.

Seção 9.7 Classe `Object`

- Consulte a tabela dos métodos da classe `Object` na Figura 9.12.

Terminologia

`@Override`, anotação, 284
anular um método de superclasse, 281
biblioteca de classes, 279
`BorderLayout`, classe, 300
classe básica, 279
classe derivada, 279
componente reutilizável padrão, 279
componentes reutilizáveis de software, 279
cópia em profundidade, 298
cópia superficial, 298
é um, relacionamento, 279
especialização, 279
estender uma classe, 279
`extends`, palavra-chave, 282
herança, 279
herança múltipla, 279
herança única, 279
hierarquia de classes, 279
hierarquia de herança, 280
`ImageIcon`, classe, 300
`JLabel`, classe, 299
layout, 300
`Object`, classe, 282
`protected`, modificador de acesso, 281
relacionamento *tem um*, 279
sintaxe de chamada de construtor de superclasse, 291
software frágil, 293
subclasse, 279
superclasse, 279
superclasse direta, 279
superclasse indireta, 279

Exercícios de autorrevisão

9.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) _____ é uma forma de capacidade de reutilização de software em que novas classes adquirem os membros de classes existentes e aprimoram essas classes com novas capacidades.

b) Os membros _____ de uma superclasse podem ser acessados na declaração de superclasse *e nas* declarações de subclasse.

c) Em um relacionamento _____, um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse.

d) Em um relacionamento _____, um objeto de classe tem referências a objetos de outras classes como membros.

e) Na herança simples, há uma classe em um relacionamento _____ com suas subclasses.

f) Uma superclasse de membros _____ são acessíveis em qualquer lugar em que o programa tem uma referência para um objeto daquela superclasse ou para um objeto de uma de suas subclasses.

g) Quando um objeto de uma subclasse é instanciado, uma superclasse _____ é chamada implícita ou explicitamente.

h) Os construtores de subclasse podem chamar construtores de superclasse via palavra-chave _____.

9.2 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se uma instrução for *falsa*, explique por quê.

a) Os construtores de superclasse não são herdados por subclasses.

b) Um relacionamento *tem um* é implementado via herança.

c) Uma classe `Car` tem um relacionamento *é um* com as classes `SteeringWheel` e `Brakes`.

d) Quando uma subclasse redefinir um método de superclasse utilizando a mesma assinatura, diz-se que a subclasse sobrecarrega esse método de superclasse.

Respostas dos exercícios de autorrevisão

9.1 a) Herança. b) `public` e `protected`. c) *é um* ou herança. d) *tem um* ou composição. e) hierárquico. f) `public`. g) construtor. h) `super`.

9.2 a) Verdadeiro. b) Falso. Um relacionamento *tem um* é implementado via composição. Um relacionamento *é um* é implementado via herança. c) Falso. Esse é um exemplo de um relacionamento *tem um*. A classe `Car` tem um relacionamento *é um* com a classe `Vehicle`. d) Falso. Isso é conhecido como sobrescrição (*overriding*), não sobrecarga (*overloading*) — um método sobrecarregado tem o mesmo nome, mas uma assinatura diferente.

Exercícios

9.3 Muitos programas escritos com herança podem ser escritos com composição e vice-versa. Reescreva a classe `BasePlusCommissionEmployee` (Figura 9.11) da hierarquia `CommissionEmployee-BasePlusCommissionEmployee` para utilizar a composição em vez da herança.

9.4 Discuta de que maneira a herança promove a reutilização de software, economiza tempo durante o desenvolvimento de programa e ajuda a evitar erros.

9.5 Desenhe uma hierarquia de herança para os alunos em uma universidade semelhante à hierarquia mostrada na Figura 9.2. Utilize `Student` como a superclasse da hierarquia, então estenda `Student` com as classes `UndergraduateStudent` e `GraduateStudent`. Continue a estender a hierarquia o mais profundo (isto é, com muitos níveis) possível. Por exemplo, `Freshman`, `Sophomore`, `Junior` e `Senior` [primeiranista, segundanista, terceiranista e quartanista, respectivamente] poderiam estender `UndergraduateStudent`, e `DoctoralStudent` e `MastersStudent` poderiam ser subclasses de `GraduateStudent`. Depois de desenhar a hierarquia, discuta os relacionamentos entre as classes. [*Nota:* você não precisa escrever nenhum código para este exercício.]

9.6 O mundo das formas é muito mais rico do que as formas incluídas na hierarquia de herança da Figura 9.3. Anote todas as formas que você pode imaginar — bidimensionais e tridimensionais — e transforme-as em uma hierarquia `Shape` mais completa com o maior número possível de níveis. Sua hierarquia deve ter a classe `Shape` na parte superior. As classes `TwoDimensionalShape` e `ThreeDimensionalShape` devem estender `Shape`. Acrescente subclasses adicionais, como `Quadrilateral` e `Sphere`, em suas localizações corretas na hierarquia conforme necessário.

9.7 Alguns programadores preferem não utilizar acesso `protected`, porque acreditam que ele quebra o encapsulamento da superclasse. Discuta os méritos relativos de utilizar acesso `protected` *vs.* utilizar acesso `private` em superclasses.

9.8 Escreva uma hierarquia de herança para as classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` e `Square`. Use `Quadrilateral` como a superclasse da hierarquia. Crie e use uma classe `Point` para representar os pontos em cada forma. Faça a hierarquia o mais profunda (isto é, com muitos níveis) possível. Especifique as variáveis de instância e os métodos para cada classe. As variáveis de instância `private` de `Quadrilateral` devem ser os pares de coordenadas *x*-*y* para os quatro pontos que delimitam o `Quadrilateral`. Escreva um programa que instancia objetos de suas classes e gera saída da área de cada objeto (exceto `Quadrilateral`).

10

Um Anel para a todos governar, Um Anel para encontrá-los,
Um Anel para a todos trazer e na escuridão aprisioná-los.
— **John Ronald Reuel Tolkien**

Propostas genéricas não decidem casos concretos.
— **Oliver Wendell Holmes**

Um filósofo de estatura imponente não pensa em um vazio. Mesmo suas ideias mais abstratas são, em alguma medida, condicionadas pelo que é ou não conhecido na época em que ele vive.
— **Alfred North Whitehead**

Por que estás abatida, ó minha alma?
— **Salmos 42:5**

Programação orientada a objetos: polimorfismo

Objetivos

Neste capítulo, você aprenderá:

- O conceito de polimorfismo.
- A utilizar métodos sobrescritos para executar o polimorfismo.
- A distinguir entre classes concretas e abstratas.
- A declarar métodos abstratos para criar classes abstratas.
- Como o polimorfismo torna sistemas extensíveis e sustentáveis.
- A determinar um tipo de objeto em tempo de execução.
- A declarar e implementar interfaces.

Sumário

10.1 Introdução
10.2 Exemplos de polimorfismo
10.3 Demonstrando um comportamento polimórfico
10.4 Classes e métodos abstratos
10.5 Estudo de caso: sistema de folha de pagamentos utilizando polimorfismo
- 10.5.1 Superclasse abstrata `Employee`
- 10.5.2 Subclasse concreta `SalariedEmployee`
- 10.5.3 Subclasse concreta `HourlyEmployee`
- 10.5.4 Subclasse concreta `CommissionEmployee`
- 10.5.5 Subclasse concreta indireta `BasePlusCommissionEmployee`
- 10.5.6 Processamento polimórfico, operador `instanceof` e downcasting
- 10.5.7 Resumo das atribuições permitidas entre variáveis de superclasse e de subclasse

10.6 Métodos e classes `final`
10.7 Estudo de caso: criando e utilizando interfaces
- 10.7.1 Desenvolvendo uma hierarquia `Payable`
- 10.7.2 Interface `Payable`
- 10.7.3 Classe `Invoice`
- 10.7.4 Modificando a classe `Employee` para implementar a Interface `Payable`
- 10.7.5 Modificando a classe `SalariedEmployee` para uso na hierarquia `Payable`
- 10.7.6 Utilizando a interface `Payable` para processar `Invoices` e `Employees` polimorficamente
- 10.7.7 Interfaces comuns da Java API

10.8 (Opcional) Estudo de caso de GUIs e imagens gráficas: desenhando com polimorfismo
10.9 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

10.1 Introdução

Continuaremos nosso estudo de programação orientada a objetos explicando e demonstrando o **polimorfismo** com hierarquias de herança. O polimorfismo permite "programar no geral" em vez de "programar no específico". O polimorfismo, em particular, permite escrever programas que processam objetos que compartilham a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse; isso pode simplificar a programação.

Considere o exemplo de polimorfismo a seguir. Suponha que criamos um programa que simula o movimento de vários tipos de animais para um estudo biológico. As classes `Peixe`, `Anfíbio` e `Pássaro` representam os três tipos de animais sob investigação. Imagine que cada classe estende a superclasse `Animal`, que contém um método `mover` e mantém a localização atual de um animal como coordenadas *x*-*y*. Toda subclasse implementa o método `mover`. Nosso programa mantém um array `Animal` que contém referências a objetos das várias subclasses `Animal`. Para simular os movimentos dos animais, o programa envia a mesma mensagem a cada objeto uma vez por segundo — a saber, `mover`. Entretanto, cada tipo específico de `Animal` responde a uma mensagem `mover` de uma maneira única — um `Peixe` poderia nadar um metro, um `Anfíbio` poderia pular um metro e meio e um `Pássaro` poderia voar três metros. O programa emite a *mesma* mensagem (isto é, `mover`) para cada objeto de animal, mas cada objeto sabe como modificar suas coordenadas *x*-*y* apropriadamente de acordo com seu tipo *específico* de movimento. Contar com o fato de que cada objeto sabe "fazer a coisa certa" (isto é, faz o que é apropriado a esse tipo de objeto) em resposta à mesma chamada de método é o conceito-chave do polimorfismo. A mesma mensagem (nesse caso, `mover`) enviada a uma variedade de objetos tem "muitas formas" de resultados — daí o termo polimorfismo.

Implementando para extensibilidade

Com o polimorfismo, podemos projetar e implementar sistemas que são facilmente extensíveis — novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que adicionamos à hierarquia. Por exemplo, se estendermos a classe `Animal` para criar a classe `Tartaruga` (que poderia responder a uma mensagem `mover` deslizando uma polegada), precisaremos escrever somente a classe `Tartaruga` e a parte da simulação que instancia um objeto `Tartaruga`. As partes da simulação que processam cada `Animal` genericamente podem permanecer idênticas.

Visão geral do capítulo

Este capítulo tem várias partes. Primeiro, discutiremos os exemplos comuns do polimorfismo. Então, fornecemos um pequeno exemplo que demonstra o comportamento polimórfico. Utilizamos referências de superclasse para manipular polimorficamente tanto objetos de superclasse como objetos de subclasse.

Em seguida, apresentaremos um estudo de caso que revisita a hierarquia de funcionários da Seção 9.4.5. Desenvolveremos um aplicativo simples de folha de pagamentos que calcula polimorficamente o salário semanal de diferentes funcionários usando o método `earnings` de cada funcionário. Embora os vencimentos de cada tipo de funcionário sejam calculados de uma maneira específica, o polimorfismo permite processar os funcionários "no geral". No estudo de caso, expandimos a hierarquia para incluir duas novas classes — `SalariedEmployee` (para pessoas que recebem um salário semanal fixo) e `HourlyEmployee` (para pessoas que recebem um salário por hora e horas extras com um valor 50% maior). Declaramos um conjunto comum de funcionalidades para todas as classes na hierarquia atualizada em uma classe "abstrata", `Employee`, da qual as classes "concretas" `SalariedEmployee`, `HourlyEmployee` e `CommissionEmployee` herdam diretamente e a classe "concreta" `BasePlusCommissionEmployee` herda indiretamente. Como você verá mais adiante, quando invocamos o método `earnings` de cada funcionário a partir de uma referência à superclasse `Employee`, o cálculo correto dos vencimentos da subclasse é realizado devido às capacidades polimórficas do Java.

Programando no específico

Ocasionalmente, ao realizar o processamento polimórfico, precisamos programar "no específico". Nosso estudo de caso de `Employee` demonstra que um programa pode determinar o tipo de um objeto em tempo de *execução* e atuar sobre esse objeto de maneira correspondente. No estudo de caso, decidimos que `BasePlusCommissionEmployees` devem receber um aumento de 10% no salário-base. Desse modo, utilizamos essas capacidades para determinar se determinado objeto funcionário *é um* `BasePlusCommissionEmployee`. Se for, aumentamos o salário-base desse funcionário em 10%.

Interfaces

O capítulo continua com uma introdução a interfaces Java. Uma interface descreve um conjunto de métodos que podem ser chamados em um objeto, mas não fornece implementações concretas para todos os métodos. Você pode declarar classes que **implementam** (isto é, fornecem implementações concretas para os métodos de) uma ou mais interfaces. Cada método de interface deve ser declarado em todas as classes que implementam explicitamente a interface. Depois que uma classe implementa uma interface, todos os objetos dessa classe têm um relacionamento *é um* com o tipo de interface e temos a garantia de que todos os objetos da classe fornecem a funcionalidade descrita pela interface. Isso também é verdade para todas as subclasses dessa classe.

As interfaces são particularmente úteis para atribuir funcionalidades comuns a classes possivelmente não relacionadas. Isso permite que objetos de classes não relacionadas sejam processados polimorficamente — objetos de classes que implementam a mesma interface podem responder a todas as chamadas de método da interface. Para demonstrar a criação e uso de interfaces, modificaremos nosso aplicativo de folha de pagamentos para criar um aplicativo geral de contas a pagar que pode calcular pagamentos devidos aos funcionários da empresa e as quantias das faturas a serem cobradas por mercadorias adquiridas. Como veremos, as interfaces permitem capacidades polimórficas semelhantes às possíveis com a herança.

10.2 Exemplos de polimorfismo

Agora, veremos vários exemplos adicionais do polimorfismo.

Quadriláteros

Se a classe `Retângulo` for derivada da classe `Quadrilátero`, então um objeto `Retângulo` é uma versão mais específica de um `Quadrilátero`. Qualquer operação (por exemplo, calcular o perímetro ou a área) que pode ser realizada em um `Quadrilátero` também pode ser realizada em um `Retângulo`. Essas operações também podem ser realizadas em outros `Quadriláteros`, como `Quadrados`, `Paralelogramos` e `Trapezoides`. O polimorfismo ocorre quando um programa invoca um método por meio de uma superclasse variável `Quadrilátero` — em tempo de execução, a versão de subclasse correta do método é chamada, com base no tipo da referência armazenada na variável de superclasse. Veremos um exemplo simples do código que ilustra esse processo na Seção 10.3.

Objetos espaciais em um videogame

Suponha que fôssemos projetar um videogame que manipulasse objetos das classes `Marciano`, `Venusiano`, `Plutoniano`, `NaveEspacial` e `CanhãoDeLaser`. Imagine que cada classe herda da superclasse `ObjetoEspacial`, que contém o método `desenhar`. Cada subclasse implementa esse método. Um programa de gerenciamento de tela mantém uma coleção (por exemplo, um array `ObjetoEspacial`) de referências a objetos das várias classes. Para atualizar a tela, o gerenciador de tela envia periodicamente a mesma mensagem a cada objeto — a saber, `desenhar`. Cada objeto, porém, responde de uma maneira única. Por exemplo, um objeto `Marciano` desenharia a si mesmo em vermelho com olhos verdes e o número apropriado de antenas. Um objeto `NaveEspacial` desenharia a si mesmo como disco voador brilhante prateado. Um objeto `CanhãoDeLaser` poderia se desenhar como um feixe vermelho brilhante através da tela. Mais uma vez, a *mesma* mensagem (nesse caso, `desenhar`) enviada a uma variedade de objetos tem "muitas formas" de resultados.

Um gerenciador de tela poderia utilizar o polimorfismo para facilitar a adição de novas classes a um sistema com modificações mínimas no código do sistema. Suponha que queremos adicionar objetos `Mercurianos` ao nosso videogame. Para fazer isso, construiríamos uma classe `Mercuriano` que estende `ObjetoEspacial` e fornece sua própria implementação do método `desenhar`. Quando objetos `Mercurianos` aparecem na coleção `ObjetoEspacial`, o código do gerenciador de tela invoca o método `desenhar`, exatamente como faz para um ou outro objeto na coleção, independentemente do seu tipo. Assim, os novos objetos `Mercurianos` são simplesmente "conectados"

sem nenhuma modificação no código do gerenciador de tela pelo programador. Portanto, sem modificar o sistema (além de construir novas classes e modificar o código que cria novos objetos), você pode utilizar o polimorfismo para incluir convenientemente tipos adicionais que não foram considerados quando o sistema foi criado.

Observação de engenharia de software 10.1

O polimorfismo permite-lhe tratar as generalidades e deixar que o ambiente de tempo de execução trate as especificidades. Você pode instruir objetos a se comportarem de maneiras apropriadas para esses objetos, sem nem mesmo conhecer seus tipos (contanto que os objetos pertençam à mesma hierarquia de herança).

Observação de engenharia de software 10.2

O polimorfismo promove extensibilidade: o software que invoca o comportamento polimórfico é independente dos tipos de objeto para os quais as mensagens são enviadas. Novos tipos de objetos que podem responder a chamadas de método existentes podem ser incorporados a um sistema sem modificar o sistema básico. Somente o código de cliente que instancia os novos objetos deve ser modificado para acomodar os novos tipos.

10.3 Demonstrando um comportamento polimórfico

A Seção 9.4 criou uma hierarquia de classe, na qual a classe `BasePlusCommissionEmployee` estendeu `CommissionEmployee`. Os exemplos nessa seção manipularam os objetos `CommissionEmployee` e `BasePlusCommissionEmployee` utilizando referências para invocar seus métodos — temos por alvo variáveis de superclasse em objetos de superclasse e variáveis de subclasse em objetos de subclasse. Essas atribuições são naturais, simples e diretas — variáveis de superclasse são concebidas para referenciar objetos de superclasse e variáveis de subclasse são concebidas para referenciar objetos de subclasse. Mas como você verá mais adiante, outras atribuições são possíveis.

No próximo exemplo, temos por alvo uma referência de superclasse em um objeto de subclasse. Mostramos então como invocar um método em um objeto de subclasse via uma referência de superclasse que invoca a funcionalidade da subclasse — o tipo de *objeto referenciado real*, não o tipo de *variável*, determina qual método é chamado. Esse exemplo demonstra que um objeto de uma subclasse pode ser tratado como um objeto de sua superclasse, permitindo várias manipulações interessantes. Um programa pode criar um array de variáveis de superclasse que referencia objetos de muitos tipos de subclasse. Isso é permitido porque cada objeto de subclasse *é um* objeto da sua superclasse. Por exemplo, podemos atribuir a referência de um objeto `BasePlusCommissionEmployee` a uma superclasse variável `CommissionEmployee`, porque um `BasePlusCommissionEmployee` *é um* `CommissionEmployee` — podemos tratar um `BasePlusCommissionEmployee` como um `CommissionEmployee`.

Como veremos mais tarde neste capítulo, você não pode tratar um objeto de superclasse como um objeto de subclasse, porque um objeto de superclasse não é um objeto de quaisquer das suas subclasses. Por exemplo, não podemos atribuir a referência de um objeto `CommissionEmployee` a uma subclasse variável `BasePlusCommissionEmployee`, porque um `CommissionEmployee` não é um `BasePlusCommissionEmployee` — um `CommissionEmployee` não tem uma variável de instância `baseSalary` e não tem os métodos `setBaseSalary` e `getBaseSalary`. O relacionamento *é um* se aplica somente para cima na *hierarquia* a partir de uma subclasse até suas superclasses diretas (e indiretas), e não vice-versa (isto é, não da hierarquia de uma superclasse às suas subclasses).

O compilador Java *realmente* permite a atribuição de uma referência de superclasse a uma variável de subclasse se fizermos explicitamente uma coerção (*casting*) da referência de superclasse para o tipo de subclasse — uma técnica discutida na Seção 10.5. Por que iríamos querer realizar essa atribuição? Uma referência de superclasse só pode ser utilizada para invocar os métodos declarados na superclasse — tentativas de invocar métodos somente de subclasse por meio de uma referência de superclasse resultam em erros de compilação. Se um programa precisar realizar uma operação específica na superclasse em um objeto de superclasse referenciado por uma variável de superclasse, o programa deverá primeiro fazer uma coerção [*cast*] da referência de superclasse para uma referência de subclasse por meio de uma técnica conhecida como **downcasting**. Isso permite ao programa invocar métodos de subclasse que não estão na superclasse. Mostramos um exemplo de downcasting na Seção 10.5.

O exemplo na Figura 10.1 demonstra três maneiras de utilizar variáveis de superclasse e subclasse para armazenar referências a objetos de superclasse e subclasse. Os dois primeiros são simples e diretos — como na Seção 9.4, atribuímos uma referência de superclasse a uma variável de superclasse, e atribuímos uma referência de subclasse a uma variável de subclasse. Demonstraremos então o relacionamento entre subclasses e superclasses (isto é, o relacionamento *é um*) atribuindo uma referência de subclasse a uma variável de superclasse. [*Nota:* esse programa utiliza as classes `CommissionEmployee` e `BasePlusCommissionEmployee` das figuras 9.10 e 9.11, respectivamente.]

Na Figura 10.1, as linhas 10–11 criam um objeto `CommissionEmployee` e atribuem sua referência a uma variável `CommissionEmployee`. As linhas 14–16 criam um objeto `BasePlusCommissionEmployee` e atribuem sua referência a uma variável `BasePlusCommissionEmployee`. Essas atribuições são naturais — por exemplo, o principal propósito de uma variável `CommissionEmployee` é armazenar uma referência a um objeto `CommissionEmployee`. As linhas 19–21 utilizam `commissionEmployee` para invocar `toString` explicitamente. Como `commissionEmployee` referencia um objeto `CommissionEmployee`, a versão da superclasse `CommissionEmployee` de `toString` é chamada. Do modo semelhante, as linhas 24–27 utilizam `basePlusCommissionEmployee` para invocar `toString` explicitamente no objeto `BasePlusCommissionEmployee`. Isso invoca a versão da subclasse `BasePlusCommissionEmployee` para `toString`.

```
1  // Figura 10.1: PolymorphismTest.java
2  // Atribuindo referências de superclasse e subclasse a
3  // variáveis de superclasse e de subclasse.
4
5  public class PolymorphismTest
6  {
7     public static void main( String[] args )
8     {
9        // atribui uma referência de superclasse a variável de superclasse
10       CommissionEmployee commissionEmployee = new CommissionEmployee(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // atribui uma referência de subclasse a variável de subclasse
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoca toString no objeto de superclasse utilizando a variável de superclasse
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
23       // invoca toString no objeto de subclasse utilizando a variável de subclasse
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Call BasePlusCommissionEmployee's toString with subclass",
26          "reference to subclass object",
27          basePlusCommissionEmployee.toString() );
28
29       // invoca toString no objeto de subclasse utilizando a variável de superclasse
30       CommissionEmployee commissionEmployee2 =
31          basePlusCommissionEmployee;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee's toString with superclass",
34       "reference to subclass object", commissionEmployee2.toString() );
35    } // fim de main
36 } // fim da classe PolymorphismTest
```

```
Call CommissionEmployee's toString with superclass reference to superclass object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Figura 10.1 | Atribuindo referências de superclasse e subclasse a variáveis de superclasse e subclasse.

As linhas 30–31 então atribuem a referência do objeto de subclasse `basePlusCommissionEmployee` a uma variável `CommissionEmployee` de superclasse, que as linhas 32–34 utilizam para invocar o método `toString`. *Quando uma variável de superclasse contém uma referência a um objeto de subclasse, e essa referência é utilizada para chamar um método, a versão de subclasse do método é chamada.* Portanto, `commissionEmployee2.toString()` na linha 34 de fato chama o método `toString` *da classe* `BasePlusCommissionEmployee`. O compilador Java permite esse "cruzamento" porque um objeto de uma subclasse é um objeto da sua superclasse (mas não vice-versa). Quando o compilador encontra uma chamada de método feita por meio de uma variável, ele determina se o método pode ser chamado verificando o tipo de classe da variável. Se essa classe contiver a declaração de método apropriada (ou estender uma), a chamada é compilada. Em tempo de execução, o tipo do objeto que a variável referencia determina o método real a utilizar. Esse processo, chamado de vinculação dinâmica, é discutido detalhadamente na Seção 10.5.

10.4 Classes e métodos abstratos

Quando pensamos num tipo de classe, supomos que os programas criam objetos desse tipo. Às vezes é útil declarar classes — chamadas **classes abstratas** — para as quais você nunca pretende criar objetos. Como elas só são usadas como superclasses em hierarquias de herança, são chamadas **superclasses abstratas**. Essas classes não podem ser usadas para instanciar objetos, porque, como veremos mais adiante, classes abstratas são incompletas. As subclasses devem declarar as "partes ausentes" para tornarem-se classes "concretas", a partir das quais você pode instanciar objetos. Caso contrário, essas subclasses, também, serão abstratas. Demonstraremos as classes abstratas na Seção 10.5.

Propósito das classes abstratas

O propósito de uma classe abstrata é fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e assim podem compartilhar um design comum. Na hierarquia `Forma` da Figura 9.3, por exemplo, as subclasses herdam a noção do que seria uma `Forma` — talvez atributos comuns como `localização`, `cor` e `espessuraDaBorda` e comportamentos como `desenhar`, `mover`, `redimensionar` e `mudarDeCor`. As classes que podem ser utilizadas para instanciar objetos são chamadas **classes concretas**. Essas classes fornecem implementações de cada método que elas declaram (algumas implementações podem ser herdadas). Por exemplo, poderíamos derivar as classes concretas `Círculo`, `Quadrado` e `Triângulo` da superclasse abstrata `FormaBidimensional`. De modo semelhante, podemos derivar classes `Esfera`, `Cubo` e `Tetraedro` concretas a partir da superclasse `FormaTridimensional` abstrata. Superclasses abstratas são excessivamente gerais para criar objetos reais — elas só especificam o que é comum entre subclasses. Precisamos ser mais específicos antes de criar objetos. Por exemplo, se você enviar a mensagem `desenhar` à classe abstrata `FormaBidimensional`, ela sabe que formas bidimensionais devem ser desenháveis, mas não que forma específica desenhar, portanto ela não pode implementar um verdadeiro método `desenhar`. As classes concretas fornecem os aspectos específicos que tornam razoável instanciar objetos.

Nem todas as hierarquias contêm classes abstratas. Entretanto, programadores costumam escrevem código de cliente que utiliza apenas tipos abstratos de superclasse para reduzir dependências do código de cliente em um intervalo de tipos de subclasse. Por exemplo, você pode escrever um método com o parâmetro de um tipo de superclasse abstrata. Quando chamado, esse método pode receber um objeto de qualquer classe concreta que direta ou indiretamente estende a superclasse especificada como o tipo do parâmetro.

As classes abstratas às vezes constituem vários níveis da hierarquia. Por exemplo, a hierarquia `Forma` da Figura 9.3 inicia com a classe abstrata `Forma`. No próximo nível da hierarquia estão as classes abstratas `FormaBidimensional` e `FormaTridimensional`. O nível seguinte da hierarquia declara classes concretas para `FormaBidimensionais` (`Círculo`, `Quadrado` e `Triângulo`) e para `FormaTridimensionais` (`Esfera`, `Cubo` e `Tetraedro`).

Declarando uma classe abstrata e métodos abstratos

Você cria uma classe abstrata declarando-a com a palavra-chave **`abstract`**. Uma classe abstrata normalmente contém um ou mais **métodos abstratos**. Um método abstrato é aquele com a palavra-chave `abstract` na sua declaração, como em

```
public abstract void draw(); // método abstrato
```

Métodos abstratos não fornecem implementações. Uma classe que contém métodos abstratos deve ser declarada como uma classe abstrata mesmo se essa classe contiver alguns métodos concretos (não abstratos). Cada subclasse concreta de uma superclasse abstrata também deve fornecer implementações concretas de cada um dos métodos abstratos da superclasse. Os construtores e métodos `static` não podem ser declarados `abstract`. Os construtores não são herdados, portanto um construtor `abstract` nunca seria implementado. Embora os métodos `static` não `private` sejam herdados, eles não podem ser sobrescritos. Como métodos `abstract` se destinam a serem sobrescritos de tal modo que possam processar objetos baseados em seus tipos, não faria sentido declarar um método `static` como `abstract`.

Observação de engenharia de software 10.3

Uma classe abstrata declara atributos e comportamentos comuns (ambos abstratos e concretos) das várias classes em uma hierarquia de classes. Em geral, uma classe abstrata contém um ou mais métodos abstratos que as subclasses devem sobrescrever se elas precisarem ser concretas. Variáveis de instância e métodos concretos de uma classe abstrata estão sujeitos às regras normais da herança.

Erro comum de programação 10.1

Tentar instanciar um objeto de uma classe abstrata é um erro de compilação.

Erro comum de programação 10.2

A falha em implementar os métodos abstratos de uma superclasse em uma subclasse é um erro de compilação a menos que a subclasse também seja declarada `abstract`.

Utilizando classes abstratas para declarar variáveis

Embora não seja possível instanciar objetos de superclasses abstratas, você verá mais adiante que é *possível* utilizar superclasses abstratas para declarar variáveis que podem conter referências a objetos de qualquer classe concreta derivados dessas superclasses abstratas. Os programas em geral utilizam essas variáveis para manipular objetos de subclasse polimorficamente. Você também pode utilizar nomes abstratos de superclasse para invocar métodos `static` declarados nessas superclasses abstratas.

Considere uma outra aplicação do polimorfismo. Um programa de desenho precisa exibir muitas formas, incluindo tipos de novas formas que você adicionará ao sistema depois de escrever o programa de desenho. O programa de desenho talvez precise exibir formas, como `Círculos`, `Triângulos`, `Retângulos`, ou outros, que derivam da classe abstrata `Forma`. O programa de desenho utiliza variáveis `Forma` para gerenciar os objetos que são exibidos. Para desenhar qualquer objeto nessa hierarquia de herança, o programa de desenho utiliza uma variável da superclasse `Forma` contendo uma referência ao objeto da subclasse para invocar o método `desenhar` do objeto. Esse método é declarado `abstract` na superclasse `Forma`, portanto toda subclasse concreta *deve* implementar o método `desenhar` de maneira específica para essa forma — todo objeto na hierarquia de herança `Forma` *sabe como se desenhar*. O programa de desenho não precisa se preocupar com o tipo de cada objeto ou se o programa encontrou objetos desse tipo.

Sistemas de software em camadas

O polimorfismo é particularmente eficaz para implementar os chamados sistemas de software em camadas. Em sistemas operacionais, por exemplo, cada tipo de dispositivo físico poderia operar diferentemente dos outros. Mesmo assim, os comandos para *ler* (*read*) ou *gravar* (*write*) os dados de e a partir de dispositivos poderiam ter certa uniformidade. Para cada dispositivo, o sistema operacional usa um software chamado driver de dispositivo para controlar toda a comunicação entre o sistema e o dispositivo. A mensagem write enviada a um driver de dispositivo precisa ser interpretada especificamente no contexto desse driver e como ela manipula dispositivos de um tipo específico. Entretanto, a própria chamada de gravar não é realmente diferente de gravar em qualquer outro dispositivo no sistema — transfira um número de bytes da memória para esse dispositivo. Um sistema operacional orientado a objetos talvez utilize uma superclasse abstrata para fornecer uma "interface" apropriada para todos os drivers de dispositivo. Então, por meio da herança a partir dessa superclasse abstrata, todas as subclasses são formadas com um comportamento semelhante. Os métodos do driver de dispositivo são declarados como métodos abstratos na superclasse abstrata. As implementações desses métodos abstratos são fornecidas nas subclasses que correspondem com os tipos de drivers de dispositivo específicos. Novos dispositivos são continuamente desenvolvidos, muitas vezes bem depois que o sistema operacional foi distribuído. Ao comprar um novo dispositivo, ele vem com um driver de dispositivo do fornecedor do dispositivo. O dispositivo torna-se imediatamente operacional depois que você conecta e instala o driver no seu computador. Esse é um outro exemplo elegante de como o polimorfismo torna sistemas extensíveis.

10.5 Estudo de caso: sistema de folha de pagamentos utilizando polimorfismo

Esta seção reexamina a hierarquia `CommissionEmployee-BasePlusCommissionEmployee` que exploraremos integralmente na Seção 9.4. Agora usaremos um método abstrato e o polimorfismo para realizar cálculos da folha de pagamentos com base no tipo de hierarquia de herança de um funcionário. Criamos uma hierarquia de herança de funcionários aprimorada para satisfazer os seguintes requisitos:

> *Uma empresa paga seus funcionários semanalmente. Os funcionários são de quatro tipos: Funcionários assalariados recebem salários fixos semanais independentemente do número de horas trabalhadas, funcionários que trabalham por hora são pagos da mesma forma e recebem horas extras (isto é, 1,5 vezes sua taxa de salário por hora) por todas as horas trabalhadas além das 40 horas normais, funcionários comissionados recebem uma porcentagem sobre suas vendas e funcionários assalariados/comissionados recebem um salário-base mais uma porcentagem sobre suas vendas. Para o período salarial atual, a empresa decidiu recompensar os funcionários assalariados/comissionados adicionando 10% aos seus salários-base. A empresa quer escrever um aplicativo Java que realiza os cálculos da folha de pagamentos polimorficamente.*

Utilizamos a classe `abstract Employee` para representar o conceito geral de um funcionário. As classes que estendem `Employee` são `SalariedEmployee`, `CommissionEmployee` e `HourlyEmployee`. A classe `BasePlusCommissionEmployee` — que estende `CommissionEmployee` — representa o último tipo de funcionário. O diagrama de classe UML na Figura 10.2 mostra a hierarquia de herança do nosso aplicativo polimórfico de folha de pagamentos de funcionários. Note que o nome da classe abstrata `Employee` está italizado — uma convenção da UML.

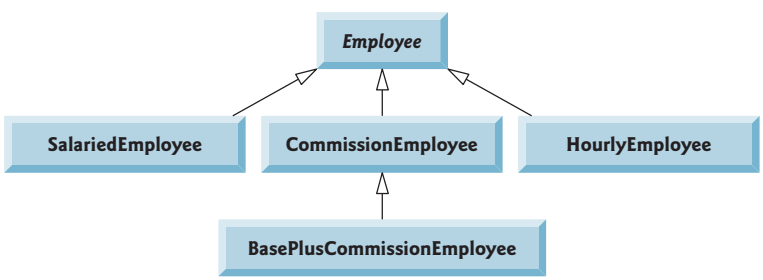


Figura 10.2 | Diagrama de classe da UML da hierarquia `Employee`.

A superclasse abstrata `Employee` declara a "interface" para a hierarquia — isto é, o conjunto de métodos que um programa pode invocar em todos os objetos `Employee`. Aqui, utilizamos o termo "interface" em um sentido geral para nos referirmos às várias maneiras como os programas se comunicam com objetos de qualquer subclasse `Employee`. Cuidado para não confundir a noção geral de uma "in-

terface" com a noção formal de uma interface Java, o tema da Seção 10.7. Cada funcionário, independentemente de como seus vencimentos são calculados, tem um nome, sobrenome e um número de seguro social, portanto variáveis de instância `private` `firstName`, `lastName` e `socialSecurityNumber` aparecem na superclasse abstrata `Employee`.

As próximas seções implementam a hierarquia de classe `Employee` da Figura 10.2. A primeira seção implementa a superclasse abstrata `Employee`. As próximas quatro seções implementam uma das classes concretas. A última seção implementa um programa de teste que constrói objetos de todas essas classes e processa esses objetos polimorficamente.

10.5.1 Superclasse abstrata `Employee`

A classe `Employee` (Figura 10.4) fornece os métodos `earnings` e `toString`, além dos métodos *get* e *set* que manipulam as variáveis de instância de `Employee`. Um método `earnings` certamente se aplica genericamente a todos os funcionários. Mas cada cálculo dos vencimentos depende da classe do funcionário. Assim, declaramos `earnings` como `abstract` na superclasse `Employee` porque uma implementação padrão não faz sentido para esse método — não há informações suficientes para determinar o valor monetário que `earnings` deve retornar. Cada subclasse sobrescreve `earnings` com uma implementação apropriada. Para calcular os vencimentos de um funcionário, o programa atribui uma referência ao objeto do funcionário a uma variável da superclasse `Employee` e, então, invoca o método `earnings` nessa variável. Mantemos um array de variáveis `Employee`, cada um armazenando uma referência a um objeto `Employee`. (Naturalmente, não pode haver objetos `Employee`, porque `Employee` é uma classe abstrata. Por causa da herança, porém, todos os objetos de todas as subclasses de `Employee` podem ser considerados objetos `Employee`.) O programa iterará pelo array e chamará o método `earnings` para cada objeto `Employee`. O Java processa essas chamadas de método polimorficamente. Incluir `earnings` como um método `abstract` em `Employee` permite que as chamadas a `earnings` por meio de variáveis `Employee` sejam compiladas e força toda subclasse concreta direta de `Employee` a sobrescrever `earnings`.

O método `toString` na classe `Employee` retorna uma `String` que contém o nome, o sobrenome e o número do seguro social do funcionário. Como veremos, cada subclasse de `Employee` sobrescreve o método `toString` para criar uma representação `String` de um objeto dessa classe que contém o tipo do funcionário (por exemplo, `"salaried employee:"`) seguida pelas demais informações do funcionário.

O diagrama na Figura 10.3 mostra cada uma das cinco classes na hierarquia no canto inferior esquerdo e os métodos `earnings` e `toString` na parte superior. Para cada classe, o diagrama mostra os resultados desejados de cada método. Não listamos os métodos *set* e *get* da superclasse `Employee` porque eles não são sobrescritos em nenhuma das subclasses — cada um desses métodos é herdado e utilizado "como é" por cada subclasse.

	`earnings`	`toString`
`Employee`	`abstract`	*firstName lastName* `social security number:` *SSN*
`Salaried-Employee`	`weeklySalary`	`salaried employee:` *firstName lastName* `social security number:` *SSN* `weekly salary:` *weeklysalary*

`Hourly-Employee`	`if (hours <= 40)`
	`  wage * hours`
	`else if (hours > 40)`
	`{`
	`  40 * wage +`
	`  ( hours - 40 ) *`
	`  wage * 1.5`
	`}`
		`hourly employee:` *firstName lastName*
		`social security number:` *SSN*
		`hourly wage:` *wage*`; hours worked:` *hours*

`Commission-Employee`	`commissionRate *` `grossSales`	`commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*`;` `commission rate:` *commissionRate*

`BasePlus-Commission-Employee`	`(commissionRate *`
	`grossSales) +`
	`baseSalary`
		`base salaried commission employee:`
		firstName lastName
		`social security number:` *SSN*
		`gross sales:` *grossSales*`;`
		`commission rate:` *commissionRate*`;`
		`base salary:` *baseSalary*

Figura 10.3 | Interface polimórfica para as classes na hierarquia `Employee`.

Vamos considerar a declaração da classe `Employee` (Figura 10.4). Essa classe inclui um construtor que recebe o nome, sobrenome e número do seguro social como argumentos (linhas 11–16); os métodos *get* que retornam o nome, o sobrenome e o número do seguro social (linhas 25–28, 37–40 e 49–52, respectivamente); os métodos *set* que configuram o nome, o sobrenome e o número do seguro social (linhas 19–22, 31–34 e 43–46, respectivamente); o método `toString` (linhas 55–60), que retorna a representação de `String` de um `Employee`

e o método `earnings` `abstract` (linha 63), que será implementado por cada uma das subclasses concretas. Observe que o construtor `Employee` não valida o número do seguro social nesse exemplo. Normalmente, essa validação deve ser fornecida.

```
1  // Figura 10.4: Employee.java
2  // Superclasse abstrata Employee.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // construtor com três argumentos
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // fim do construtor Employee com três argumentos
17
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first; // deve validar
22    } // fim do método setFirstName
23
24    // retorna o nome
25    public String getFirstName()
26    {
27       return firstName;
28    } // fim do método getFirstName
29
30    // configura o sobrenome
31    public void setLastName( String last )
32    {
33       lastName = last; // deve validar
34    } // fim do método setLastName
35
36    // retorna o sobrenome
37    public String getLastName()
38    {
39       return lastName;
40    } // fim do método getLastName
41
42    // configura o CIC
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // deve validar
46    } // fim do método setSocialSecurityNumber
47
48    // retorna o CIC
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // fim do método getSocialSecurityNumber
53
54    // retorna a representação de String do objeto Employee
55    @Override
56    public String toString()
57    {
58       return String.format( "%s %s\nsocial security number: %s",
59          getFirstName(), getLastName(), getSocialSecurityNumber() );
60    } // fim do método toString
61
62    // método abstract sobrescrito por subclasses concretas
63    public abstract double earnings(); // nenhuma implementação aqui
64 } // fim da classe abstrata Employee
```

Figura 10.4 | Superclasse `Employee` abstrata.

Por que decidimos declarar `earnings` como um método `abstract`? Simplesmente não faz sentido fornecer uma implementação desse método na classe `Employee`. Não podemos calcular os vencimentos para um `Employee` geral — primeiro precisamos conhecer o tipo de `Employee` específico para determinar o cálculo apropriado dos vencimentos. Declarando esse método `abstract`, indicamos que cada subclasse concreta *deve* fornecer uma implementação de `earnings` apropriada e que um programa será capaz de utilizar as variáveis da superclasse `Employee` para invocar o método `earnings` polimorficamente para qualquer tipo de `Employee`.

10.5.2 Subclasse concreta SalariedEmployee

A classe `SalariedEmployee` (Figura 10.5) estende a classe `Employee` (linha 4) e sobrescreve o método abstrato `earnings` (linhas 29–33), o que torna `SalariedEmployee` uma classe concreta. A classe inclui um construtor (linhas 9–14) que recebe um nome, um sobrenome, um número do seguro social e um salário semanal como argumentos; um método *set* para atribuir um novo valor não negativo à variável de instância `weeklySalary` (linhas 17–20); um método *get* para retornar o valor de `weeklySalary` (linhas 23–26); um método `earnings` (linhas 29–33) para calcular os vencimentos de `SalariedEmployee`; e um método `toString` (linhas 36–41), que retorna uma `String` incluindo o tipo do funcionário, a saber, `"salaried employee: "` seguida pelas informações específicas ao funcionário produzidas pelo método `toString` da superclasse `Employee` e pelo método `getWeeklySalary` da `SalariedEmployee`. O construtor da classe `SalariedEmployee` passa o nome, sobrenome e o número de seguro social para o construtor de `Employee` (linha 12) a fim de inicializar as variáveis de instância `private` não herdadas da superclasse. O método `earnings` sobrescreve o método abstrato `earnings` da `Employee` para fornecer uma implementação concreta que retorna o salário semanal da `SalariedEmployee`. Se não implementarmos `earnings`, a classe `SalariedEmployee` deve ser declarada `abstract` — caso contrário, a classe `SalariedEmployee` não compilará. Naturalmente, queremos que `SalariedEmployee` seja aqui uma classe concreta nesse exemplo.

```
1  // Figura 10.5: SalariedEmployee.java
2  // A classe concreta SalariedEmployee estende a classe Employee abstrata.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // construtor com quatro argumentos
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // passa para o construtor Employee
13       setWeeklySalary( salary ); // valida e armazena o salário
14    } // fim do construtor SalariedEmployee com quatro argumentos
15
16    // configura o salário
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // fim do método setWeeklySalary
21
22    // retorna o salário
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // fim do método getWeeklySalary
27
28    // calcula os rendimentos; sobrescreve o método earnings em Employee
29    @Override
30    public double earnings()
31    {
32       return getWeeklySalary();
33    } // fim do método earnings
34
35    // retorna a representação String do objeto SalariedEmployee
36    @Override
37    public String toString()
38    {
39       return String.format( "salaried employee: %s\n%s: $%,.2f",
40          super.toString(), "weekly salary", getWeeklySalary() );
41    } // fim do método toString
42 } // fim da classe SalariedEmployee
```

Figura 10.5 | A classe concreta `SalariedEmployee` estende a classe `abstract` `Employee`.

O método `toString` (linhas 36–41) da classe `SalariedEmployee` sobrescreve o método `Employee` `toString`. Se a classe `SalariedEmployee` não sobrescrevesse `toString`, `SalariedEmployee` teria estendido a versão de `Employee` para `toString`. Nesse caso, o método `toString` da `SalariedEmployee` simplesmente retornaria o nome completo e número do seguro social do funcionário, o que não representa adequadamente uma `SalariedEmployee`. Para produzir uma representação completa de `String` de uma `SalariedEmployee`, o método `toString` da subclasse retorna `"salaried employee:"` seguido pelas informações específicas da superclasse `Employee` (isto é, nome, sobrenome e número do seguro social) obtidas invocando o método `Employee` da superclasse (linha 40) — esse é um exemplo elegante de reutilização de código. A representação de `String` de uma `SalariedEmployee` também contém o salário semanal do funcionário obtido invocando o método `getWeeklySalary` da classe.

10.5.3 Subclasse concreta `HourlyEmployee`

A classe `HourlyEmployee` (Figura 10.6) também estende `Employee` (linha 4). Essa classe inclui um construtor (linhas 10–16) que recebe como argumentos um nome, sobrenome, número do seguro social, um salário por hora e o número de horas trabalhadas. As linhas 19–22 e 31–35 declaram os métodos *set* que atribuem novos valores às variáveis de instância `wage` e `hours`, respectivamente. O método `setWage` (linhas 19–22) assegura que `wage` é não negativo, e o método `setHours` (linhas 31–35) assegura que `hours` está entre 0 e 168 (o número total de horas durante uma semana) inclusive. A classe `HourlyEmployee` também inclui os métodos *get* (linhas 25–28 e 38–41) para retornar os valores de `wage` e `hours`, respectivamente; um método `earnings` (linhas 44–51) para calcular os vencimentos de `HourlyEmployee`; e um método `toString` (linhas 54–60), que retorna uma `String` contendo o tipo do funcionário (`"hourly employee: "`) e informações específicas ao funcionário. Observe que o construtor `HourlyEmployee`, como o construtor `SalariedEmployee`, passa o nome, o sobrenome e o número de seguro social para o construtor da superclasse `Employee` (linha 13) a fim de inicializar as variáveis de instância `private`. Além disso, o método `toString` chama o método `toString` da superclasse (linha 58) para obter informações específicas da `Employee` (isto é, nome, sobrenome e número do seguro social) — esse é um outro exemplo elegante de reutilização de código.

```
1  // Figura 10.6: HourlyEmployee.java
2  // Classe HourlyEmployee estende Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // salário por hora
7     private double hours; // horas trabalhadas durante a semana
8
9     // construtor de cinco argumentos
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // valida a remuneração por hora
15       setHours( hoursWorked ); // valida as horas trabalhadas
16    } // fim do construtor HourlyEmployee com cinco argumentos
17
18    // configura a remuneração
19    public void setWage( double hourlyWage )
20    {
21       wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // fim do método setWage
23
24    // retorna a remuneração
25    public double getWage()
26    {
27       return wage;
28    } // fim do método getWage
29
30    // configura as horas trabalhadas
31    public void setHours( double hoursWorked )
32    {
33       hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34          hoursWorked : 0.0;
35    } // fim do método setHours
36
37    // retorna as horas trabalhadas
38    public double getHours()
39    {
40       return hours;
41    } // fim do método getHours
42
43    // calcula os rendimentos; sobrescreve o método earnings em Employee
```

```
44       @Override
45       public double earnings()
46       {
47          if ( getHours() <= 40 ) // nenhuma hora extra
48             return getWage() * getHours();
49          else
50             return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
51       } // fim do método earnings
52
53       // retorna a representação de String do objeto HourlyEmployee
54       @Override
55       public String toString()
56       {
57       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
58             super.toString(), "hourly wage", getWage(),
59             "hours worked", getHours() );
60       } // fim do método toString
61    } // fim da classe HourlyEmployee
```

Figura 10.6 | Classe `HourlyEmployee` derivada de `Employee`.

10.5.4 Subclasse concreta `CommissionEmployee`

A classe `CommissionEmployee` (Figura 10.7) estende a classe `Employee` (linha 4). A classe inclui um construtor (linhas 10–16) que recebe um nome, um sobrenome, um número do seguro social, uma quantia de vendas e uma taxa de comissão; os métodos *set* (linhas 19–22 e 31–34) atribuem os novos valores às variáveis de instância `commissionRate` e `grossSales`, respectivamente; os métodos *get* (linhas 25–28 e 37–40) que recuperam os valores dessas variáveis de instância; o método `earnings` (linhas 43–47) para calcular os vencimentos de `CommissionEmployee`; e o método `toString` (linhas 50–57), que retorna o tipo de funcionário, a saber, `"commission employee: "`, e as informações específicas do funcionário. O construtor também passa o nome, sobrenome e número do seguro social ao construtor de `Employee` (linha 13) para inicializar as variáveis de instância `private` da `Employee`. O método `toString` chama o método `toString` da superclasse (linha 54) para obter as informações específicas de `Employee` (isto é, nome, sobrenome e número do seguro social).

```
1    // Figura 10.7: CommissionEmployee.java
2    // Classe CommissionEmployee estende Employee.
3
4    public class CommissionEmployee extends Employee
5    {
6       private double grossSales; // vendas brutas semanais
7       private double commissionRate; // porcentagem da comissão
8
9       // construtor de cinco argumentos
10      public CommissionEmployee( String first, String last, String ssn,
11         double sales, double rate )
12      {
13         super( first, last, ssn );
14         setGrossSales( sales );
15         setCommissionRate( rate );
16      } // fim do construtor CommissionEmployee de cinco argumentos
17
18      // configura a taxa de comissão
19      public void setCommissionRate( double rate )
20      {
21         commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22      } // fim do método setCommissionRate
23
24      // retorna a taxa de comissão
25      public double getCommissionRate()
26      {
27         return commissionRate;
28      } // fim do método getCommissionRate
29
30      // configura a quantidade de vendas brutas
31      public void setGrossSales( double sales )
32      {
33         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34      } // fim do método setGrossSales
35
36      // retorna a quantidade de vendas brutas
37      public double getGrossSales()
```

```
38      {
39         return grossSales;
40      } // fim do método getGrossSales
41
42      // calcula os rendimentos; sobrescreve o método earnings em Employee
43      @Override
44      public double earnings()
45      {
46         return getCommissionRate() * getGrossSales();
47      } // fim do método earnings
48
49      // retorna a representação String do objeto CommissionEmployee
50      @Override
51      public String toString()
52      {
53         return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
54            "commission employee", super.toString(),
55            "gross sales", getGrossSales(),
56            "commission rate", getCommissionRate() );
57      } // fim do método toString
58   } // fim da classe CommissionEmployee
```

Figura 10.7 | Classe `CommissionEmployee` derivada de `Employee`.

10.5.5 Subclasse concreta indireta BasePlusCommissionEmployee

A classe `BasePlusCommissionEmployee` (Figura 10.8) estende a classe `CommissionEmployee` (linha 4) e, portanto, é uma subclasse indireta da classe `Employee`. A classe `BasePlusCommissionEmployee` tem um construtor (linhas 9–14) que recebe como argumentos um nome, sobrenome, número do seguro social, um valor de vendas, uma taxa de comissão e um salário-base. Ele então passa o nome, o sobrenome, o número de seguro social, o valor total das vendas e a taxa de comissão para o construtor `CommissionEmployee` (linha 12) para inicializar os membros herdados. `BasePlusCommissionEmployee` também contém um método *set* (linhas 17–20) para atribuir um novo valor à variável de instância `baseSalary` e um método *get* (linhas 23–26) para retornar o valor de `baseSalary`. O método `earnings` (linhas 29–33) calcula os ganhos de um `BasePlusCommissionEmployee`. Note que a linha 32 no método `earnings` chama o método `earnings` da superclasse `CommissionEmployee` para calcular a parte baseada em comissão dos vencimentos do funcionário. Esse é um exemplo elegante de reutilização de código. O método `toString` da `BasePlusCommissionEmployee` (linhas 36–42) cria uma representação de `String` de uma `BasePlusCommissionEmployee`, que contém "`base-salaried`", seguida da `String` obtida invocando o método `toString` da superclasse `CommissionEmployee` (um outro exemplo de reutilização de código) e, então, o salário-base. O resultado é uma `String` que começa com "`base-salaried commission employee`" seguida pelas demais informações de `BasePlusCommissionEmployee`. Lembre-se de que o `toString` de `CommissionEmployee` obtém o nome, sobrenome e número do seguro social do funcionário invocando o método `toString` da sua superclasse (isto é, `Employee`) — um outro exemplo de reutilização de código. Note que o `toString` de `BasePlusCommissionEmployee` inicia uma cadeia de chamadas de método que se distribuem por todos os três níveis da hierarquia `Employee`.

```
1   // Fig. 10.8: BaseplusCommissionEmployee.java
2   // Classe BasePlusCommissionEmployee estende a CommissionEmployee.
3
4   public class BasePlusCommissionEmployee extends CommissionEmployee
5   {
6      private double baseSalary; // salário de base por semana
7
8      // construtor de seis argumentos
9      public BasePlusCommissionEmployee( String first, String last,
10        String ssn, double sales, double rate, double salary )
11     {
12        super( first, last, ssn, sales, rate );
13        setBaseSalary( salary ); // valida e armazena salário-base
14     } // fim do construtor BasePlusCommissionEmployee de seis argumentos
15
16     // configura o salário-base
17     public void setBaseSalary( double salary )
18     {
19        baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // não negativo
20     } // fim do método setBaseSalary
21
22     // retorna o salário-base
23     public double getBaseSalary()
24     {
```

```
25          return baseSalary;
26       } // fim do método getBaseSalary
27
28       // calcula os vencimentos; sobrescreve o método earnings em CommissionEmployee
29       @Override
30       public double earnings()
31       {
32          return getBaseSalary() + super.earnings();
33       } // fim do método earnings
34
35       // retorna a representação String do objeto BasePlusCommissionEmployee
36       @Override
37       public String toString()
38       {
39          return String.format( "%s %s; %s: $%,.2f",
40             "base-salaried", super.toString(),
41             "base salary", getBaseSalary() );
42       } // fim do método toString
43    } // fim da classe BasePlusCommissionEmployee
```

Fig. 10.8 | A classe `BasePlusCommissionEmployee` estende `CommissionEmployee`.

10.5.6 Processamento polimórfico, operador `instanceof` e downcasting

Para testar nossa hierarquia `Employee`, o aplicativo na Figura 10.9 cria um objeto de cada uma das quatro classes concretas `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`. O programa manipula esses objetos não polimorficamente, via variáveis do próprio tipo de cada objeto e, então, polimorficamente, utilizando um array de variáveis `Employee`. Ao processar os objetos polimorficamente, o programa aumenta o salário-base de cada `BasePlusCommissionEmployee` em 10% — isso requer *determinar o tipo de objeto em tempo de execução*. Por fim, o programa determina polimorficamente e gera a saída do tipo de cada objeto no array `Employee`. As linhas 9–18 criam objetos de cada uma das quatro subclasses `Employee` concretas. As linhas 22–30 geram a saída da representação de `String` e vencimentos de cada um desses objetos não polimorficamente. Note que o método `toString` de cada objeto é chamado implicitamente por `printf` quando é gerada a saída do objeto como uma `String` com o especificador de formato `%s`.

```
1  // Figura 10.9: PayrollSystemTest.java
2  // Programa de teste da hierarquia Employee.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String[] args )
7     {
8        // cria objetos de subclasse
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15          "Sue", "Jones", "333-33-3333", 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
24       System.out.printf( "%s\n%s: $%,.2f\n\n",
25          hourlyEmployee, "earned", hourlyEmployee.earnings() );
26       System.out.printf( "%s\n%s: $%,.2f\n\n",
27          commissionEmployee, "earned", commissionEmployee.earnings() );
28       System.out.printf( "%s\n%s: $%,.2f\n\n",
29          basePlusCommissionEmployee,
30          "earned", basePlusCommissionEmployee.earnings() );
31
32       // cria um array Employee de quatro elementos
33       Employee[] employees = new Employee[ 4 ];
34
```

```
35          // inicializa o array com Employees
36          employees[ 0 ] = salariedEmployee;
37          employees[ 1 ] = hourlyEmployee;
38          employees[ 2 ] = commissionEmployee;
39          employees[ 3 ] = basePlusCommissionEmployee;
40
41          System.out.println( "Employees processed polymorphically:\n" );
42
43          // processa genericamente cada elemento no employees
44          for ( Employee currentEmployee : employees )
45          {
46             System.out.println(currentEmployee ); // invoca toString
47
48             // determina se elemento é um BasePlusCommissionEmployee
49             if (currentEmployee instanceof BasePlusCommissionEmployee )
50             {
51                // downcast da referência de Employee para
52                // referência a BasePlusCommissionEmployee
53                BasePlusCommissionEmployee employee =
54                   ( BasePlusCommissionEmployee ) currentEmployee;
55
56                employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
58                System.out.printf(
59                   "new base salary with 10%% increase is: $%,.2f\n",
60                   employee.getBaseSalary() );
61             } // fim do if
62
63             System.out.printf(
64                "earned $%,.2f\n\n", currentEmployee.earnings() );
65          } // for final
66
67          // obtém o nome do tipo de cada objeto no array employees
68          for ( int j = 0; j < employees.length; j++ )
69             System.out.printf( “Employee %d is a %s\n”, j,
70                employees[ j ].getClass().getName() );
71       } // fim de main
72    } // fim da classe PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00

Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00
```

```
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Figura 10.9 | Programa de teste da hierarquia `Employee`.

Criando arrays de `Employee`s

A linha 33 declara `employees` e lhe atribui um array de quatro variáveis `Employee`. A linha 36 atribui a referência a um objeto `SalariedEmployee` para `employees[ 0 ]`. A linha 37 atribui a referência a um objeto `HourlyEmployee` para `employees[ 1 ]`. A linha 38 atribui a referência a um objeto `CommissionEmployee` para `employees[ 2 ]`. A linha 39 atribui a referência a um objeto `BasePlusCommissionEmployee` para `employee[ 3 ]`. Essas atribuições são permitidas, porque um `SalariedEmployee` *é um* `Employee`, um `HourlyEmployee` *é um* `Employee`, um `CommissionEmployee` *é um* `Employee` e um `BasePlusCommissionEmployee` *é um* `Employee`. Portanto, podemos atribuir as referências de `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e objetos `BasePlusCommissionEmployee` a variáveis da superclasse `Employee`, *mesmo que `Employee` seja uma classe abstrata*.

Processando `Employee`s polimorficamente

As linhas 44–65 iteram pelo array `employees` e invocam os métodos `toString` e `earnings` com a variável `Employee` `currentEmployee`, a qual é atribuída a referência a uma diferente `Employee` no array em cada iteração. A saída ilustra que os métodos apropriados para cada classe foram de fato invocados. Todas as chamadas ao método `toString` e `earnings` são resolvidas em tempo de execução, com base no tipo do objeto que `currentEmployee` referencia. Esse processo é conhecido como **vinculação dinâmica** ou **vinculação tardia**. Por exemplo, a linha 46 invoca implicitamente o método `toString` do objeto ao qual `currentEmployee` se refere. Como resultado da vinculação dinâmica, o Java decide qual método `toString` da classe é chamado *em tempo de execução em vez de em tempo de compilação*. Observe que apenas os métodos da classe `Employee` podem ser chamados via uma variável `Employee` (e `Employee`, naturalmente, incluindo os métodos da classe `Object`). Uma referência de superclasse pode ser utilizada para invocar somente métodos da superclasse — as implementações de método de subclasse são invocadas polimorficamente.

Realizando operações específicas ao tipo em `BasePlusCommissionEmployee`s

Realizamos um processamento especial nos objetos `BasePlusCommissionEmployee` — à medida que encontramos esses objetos, aumentamos seu salário-base em 10%. Ao processar objetos polimorficamente, em geral não precisamos nos preocupar com as "especificidades", mas, para ajustar o salário-base, temos de determinar o tipo específico de objeto `Employee` em tempo de execução. A linha 49 utiliza o operador **`instanceof`** para determinar se um tipo particular do objeto `Employee` é `BasePlusCommissionEmployee`. A condição na linha 49 é verdadeira se o objeto referenciado por `currentEmployee` *é um* `BasePlusCommissionEmployee`. Isso também seria verdadeiro para qualquer objeto de uma subclasse `BasePlusCommissionEmployee` por causa um relacionamento *é um* que uma subclasse tem com sua superclasse. As linhas 53–54 fazem downcast de `currentEmployee` do tipo `Employee` para o tipo `BasePlusCommissionEmployee` — essa coerção é permitida somente se o objeto tiver um relacionamento *é um* com `BasePlusCommissionEmployee`. A condição na linha 49 assegura que esse seja o caso. Essa coerção é requerida se invocarmos os métodos `getBaseSalary` e `setBaseSalary` da subclasse `BasePlusCommissionEmployee` no objeto `Employee` atual — como você verá mais adiante, tentar invocar um método apenas de subclasse diretamente em uma referência de superclasse é um erro de compilação.

Erro comum de programação 10.3

Atribuir uma variável de superclasse a uma variável de subclasse (sem uma coerção explícita) é um erro de compilação.

Observação de engenharia de software 10.4

Se, em tempo de execução, a referência de um objeto de subclasse tiver sido atribuída a uma variável de uma das suas superclasses diretas ou indiretas, é aceitável fazer downcast da referência armazenada nessa variável de superclasse de volta a uma referência do tipo da subclasse. Antes de realizar essa coerção, utilize o operador `instanceof` *para assegurar que o objeto é de fato um objeto de um tipo de subclasse apropriado.*

Erro comum de programação 10.4

Ao fazer o downcast de um objeto, ocorre uma `ClassCastException` *se em tempo de execução o objeto não tiver um relacionamento* é um *com o tipo especificado no operador de coerção. Só é possível fazer a coerção em uma referência no seu próprio tipo ou no tipo de uma das suas superclasses.*

Se a expressão `instanceof` na linha 49 for `true`, as linhas 53–60 realizam o processamento especial requerido pelo objeto `BasePlusCommissionEmployee`. Utilizando a variável `BasePlusCommissionEmployee employee`, a linha 56 invoca apenas os métodos `getBaseSalary` e `setBaseSalary` da subclasse para recuperar e atualizar o salário-base do funcionário com um aumento de 10%.

Chamando `earnings` polimorficamente

As linhas 63–64 invocam o método `earnings` em `currentEmployee`, que chama polimorficamente o método `earnings` apropriado do objeto da subclasse. Obter os vencimentos de `SalariedEmployee`, `HourlyEmployee` e `CommissionEmployee` polimorficamente nas linhas 63–64 produz o mesmo resultado de obter os vencimentos desses funcionários individualmente nas linhas 22–27. A quantia dos vencimentos obtidos para a `BasePlusCommissionEmployee` nas linhas 63–64 é maior do que o obtido nas linhas 28–30, devido ao aumento de 10% no seu salário-base.

Processando `Employees` polimorficamente

As linhas 68–70 exibem o tipo de cada funcionário como uma `String`. Cada objeto em Java conhece sua própria classe e pode acessar essas informações por meio do método **`getClass`**, que todas as classes herdam da classe `Object`. O método `getClass` retorna um objeto do tipo **`Class`** (do pacote `java.lang`), que contém as informações sobre o tipo do objeto, incluindo seu nome de classe. A linha 70 invoca `getClass` sobre o objeto atual a fim de obter sua classe em tempo de execução. O resultado da chamada a `getClass` é utilizado para invocar **`getName`** a fim de obter o nome de classe do objeto. Para aprender mais sobre a classe `Class`, consulte sua documentação online em `java.sun.com/javase/6/docs/api/java/lang/Class.html`.

Evitando erros de compilação com downcasting

No exemplo anterior, evitamos vários erros de compilação fazendo um downcast de uma variável `Employee` para uma variável `BasePlusCommissionEmployee` nas linhas 53–54. Se você remover o operador de coerção (`BasePlusCommissionEmployee`) da linha 54 e tentar atribuir a variável `Employee currentEmployee` diretamente à variável `BasePlusCommissionEmployee employee`, receberia um erro de compilação "`incompatible types`". Esse erro indica que a tentativa de atribuir a referência de objeto de superclasse `currentEmployee` à variável de subclasse `employee` não é permitida. O compilador evita essa atribuição porque uma `CommissionEmployee` não é uma `BasePlusCommissionEmployee` — o relacionamento *é um* só se aplica entre a subclasse e suas superclasses, não vice-versa.

De modo semelhante, se as linhas 56 e 60 utilizassem a variável de superclasse `currentEmployee` para invocar apenas os métodos de subclasse `getBaseSalary` e `setBaseSalary`, receberíamos os erros de compilação "`cannot find symbol`" [não foi possível encontrar o símbolo] nessas linhas. Não é permitido tentar invocar métodos apenas de subclasse por meio de uma variável de superclasse — embora as linhas 56 e 60 só executem se `instanceof` na linha 49 retornar `true` para indicar que `currentEmployee` armazena uma referência a um objeto `BasePlusCommissionEmployee`. Utilizando uma variável da superclasse `Employee`, podemos invocar somente os métodos localizados na classe `Employee` — `earnings`, `toString` e os métodos *set* e *get* da `Employee`.

Observação de engenharia de software 10.5

Embora o método real que é chamado dependa do tipo em tempo de execução do objeto que uma variável referencia, uma variável pode ser utilizada para invocar somente aqueles métodos que são membros do tipo dessa variável, o qual o compilador verifica.

10.5.7 Resumo das atribuições permitidas entre variáveis de superclasse e de subclasse

Agora que você viu um aplicativo completo que processa diversos objetos de subclasse polimorficamente, resumiremos o que você pode e o que não pode fazer com objetos e variáveis de superclasse e de subclasse. Embora um objeto de subclasse também *seja um* objeto de superclasse, os dois objetos são, contudo, diferentes. Como discutido anteriormente, objetos de subclasse podem ser tratados como se fossem objetos de superclasse. Mas como a subclasse pode ter apenas membros de subclasse adicionais, não é permitido atribuir uma referência de superclasse a uma variável de subclasse sem uma coerção explícita — tal atribuição deixaria os membros de subclasse indefinidos para o objeto de superclasse.

Discutimos quatro modos de atribuição de referências de superclasse e subclasse a variáveis de tipos de subclasse e superclasse:

1. Atribuir uma referência de superclasse a uma variável de superclasse é simples e direto.
2. Atribuir uma referência de subclasse a uma variável de subclasse é simples e direto.
3. Atribuir uma referência de subclasse a uma variável de superclasse é seguro, porque o objeto de subclasse *é um* objeto de sua superclasse. A variável da superclasse, porém, pode ser utilizada para referenciar apenas membros da superclasse. Se esse código referencia membros somente da subclasse por meio da variável de superclasse, o compilador informa erros.
4. Tentar atribuir uma referência de superclasse a uma variável de subclasse é um erro de compilação. Para evitar esse erro, a referência de superclasse deve sofrer uma coerção explícita para um tipo de subclasse. Em *tempo de execução*, se o objeto referenciado não for um objeto de subclasse, ocorrerá uma exceção. (Para informações adicionais sobre tratamento de exceções, ver o Capítulo 11.) Você deve utilizar o operador `instanceof` para assegurar que tal coerção seja realizada somente se o objeto for um objeto de subclasse.

10.6 Métodos e classes final

Vimos nas seções 6.3 e 6.10 que as variáveis podem ser declaradas `final` para indicar que elas não podem ser modificadas depois de serem inicializadas — essas variáveis representam valores constantes. Também é possível declarar métodos, parâmetros de método e classes com o modificador `final`.

Métodos final não podem ser sobrescritos

Um **método `final`** em uma superclasse não pode ser sobrescrito em uma subclasse. Os métodos que são declarados `private` são implicitamente `final`, porque não é possível sobrescrevê-los em uma subclasse. Os métodos que são declarados `static` também são implicitamente `final`. Uma declaração do método `final` nunca pode mudar, assim todas as subclasses utilizam a mesma implementação do método; e chamadas a métodos `final` são resolvidas em tempo de compilação — isso é conhecido como **vinculação estática**.

Classes final não podem ser superclasses

Uma **classe `final`** que é declarada `final` não pode ser uma superclasse (isto é, uma classe não pode estender uma classe `final`). Todos os métodos em uma classe `final` são implicitamente `final`. A classe `String` é um exemplo de uma classe `final`. Se fosse possível criar uma subclasse de `String`, os objetos dessa subclasse poderiam ser utilizados onde quer que `String`s fossem esperadas. Como a classe `String` não pode ser estendida, os programas que utilizam `String`s podem contar com a funcionalidade dos objetos `String` conforme especificada na Java API. Tornar a classe `final` também impede que programadores criem subclasses que poderiam driblar as restrições de segurança. Para mais informações adicionais sobre o uso de palavra-chave `final`, visite

```
java.sun.com/docs/books/tutorial/java/IandI/final.html
```

e

```
www.ibm.com/developerworks/java/library/j-jtp1029.html
```

Erro comum de programação 10.5

Tentar declarar uma subclasse de uma classe `final` é um erro de compilação.

Observação de engenharia de software 10.6

Na Java API, a ampla maioria das classes não é declarada `final`. Isso permite a herança e o polimorfismo. Entretanto, em alguns casos, é importante declarar classes `final` — em geral por questões de segurança.

10.7 Estudo de caso: criando e utilizando interfaces

Nosso próximo exemplo (figuras 10.11 a 10.13) reexamina o sistema de folha de pagamentos da Seção 10.5. Suponha que a empresa nesse exemplo deseje realizar várias operações de contabilidade em um único aplicativo de contas a pagar — além de calcular os vencimentos que devem ser pagos para cada funcionário, a empresa também deve calcular o pagamento devido de cada uma de várias faturas (isto é, contas de mercadorias adquiridas). Embora aplicadas a coisas não relacionadas (isto é, funcionários e faturas), as duas operações têm a ver com a obtenção de algum tipo de quantia de pagamento. Para um funcionário, o pagamento se refere aos vencimentos do funcionário. Para uma fatura, o pagamento se refere ao custo total das mercadorias listadas na fatura. Poderíamos calcular coisas tão diferentes como os pagamentos devidos a funcionários e faturas em um único aplicativo polimorficamente? O Java oferece uma capacidade que exige que classes não relacionadas implementem um conjunto de métodos comuns (por exemplo, um método que calcula a quantia de um pagamento)? As **interfaces** do Java oferecem exatamente essa capacidade.

Estandardizando interações

Interfaces definem e padronizam como coisas, pessoas e sistemas podem interagir entre si. Por exemplo, os controles em um rádio servem como uma interface entre os usuários do rádio e os componentes internos do rádio. Os controles permitem que os usuários realizem somente uma série limitada de operações (por exemplo, mudar de estação, ajustar o volume, escolher entre AM e FM) e diferentes rádios podem implementar os controles de diferentes maneiras (por exemplo, uso de botões, sintonizadores, comandos de voz). A interface especifica *quais* operações um rádio deve permitir que os usuários realizem, mas não especifica *como* essas operações são realizadas.

Objetos de software se comunicam via interfaces

Objetos de software também se comunicam via interfaces. Uma interface Java descreve um conjunto de métodos que pode ser chamado em um objeto para instruí-lo, por exemplo, a realizar alguma tarefa ou retornar algumas informações. O exemplo a seguir apresenta uma interface chamada `Payable` para descrever a funcionalidade de qualquer objeto que deve ser capaz de ser pago e assim deve oferecer um método para determinar a quantia de pagamento devida apropriada. Uma **declaração de interface** inicia com a palavra-chave `interface` e contém somente constantes e métodos `abstract`. Diferentemente das classes, todos os membros de interface devem ser `public` e as *interfaces não podem especificar nenhum detalhe de implementação*, como declarações de método concretos e variáveis de instância. Todos os métodos declarados em uma interface são implicitamente métodos `public abstract`, e todos os campos são implicitamente `public`, `static` e `final`. [*Nota:* a partir do Java SE 5, tornou-se uma melhor prática de programação declarar conjuntos de constantes como listas com a palavra-chave `enum`. Consulte a Seção 6.10 para uma introdução a `enum` e a Seção 8.9 para detalhes adicionais sobre `enum`.]

Boa prática de programação 10.1

De acordo com o Capítulo 9 da Java Language Specification, é estilisticamente correto declarar os métodos de uma interface sem as palavras-chave `public` *e* `abstract`*, porque elas são redundantes nas declarações de método de interface. De modo semelhante, as constantes devem ser declaradas sem as palavras-chave* `public`*,* `static` *e* `final`*, porque elas também são redundantes.*

Utilizando uma interface

Para utilizar uma interface, uma classe concreta deve especificar que ela **`implementa`** a interface e deve declarar cada método na interface com a assinatura especificada na declaração de interface. Para especificar que uma classe implementa uma interface adicione a palavra-chave `implements` e o nome da interface no fim da primeira linha da declaração da classe. Uma classe que não implementa todos os métodos da interface é uma classe abstrata e deve ser declarada `abstract`. Implementar uma interface é como assinar um contrato com o compilador que afirma "Irei declarar todos os métodos especificados pela interface ou irei declarar minha classe `abstract`".

Erro comum de programação 10.6

Falhar em implementar qualquer método de uma interface em uma classe concreta que `implementa` *a interface resulta em um erro de compilação indicando que a classe deve ser declarada* `abstract`*.*

Relacionando tipos díspares

Uma interface é geralmente utilizada quando classes díspares (isto é, não relacionadas) precisam compartilhar métodos e constantes comuns. Isso permite que objetos de classes não relacionadas sejam processados polimorficamente — objetos de classes que implementam a mesma interface podem responder às mesmas chamadas de método. Você pode criar uma interface que descreve a funcionalidade desejada e então implementar essa interface em quaisquer classes que requerem essa funcionalidade. Por exemplo, no aplicativo de contas a pagar desenvolvido nesta seção, implementamos a interface `Payable` em qualquer classe capaz de calcular uma quantia valor de pagamento (por exemplo, `Employee`, `Invoice`).

Interfaces versus classes abstratas

Uma interface costuma ser utilizada no lugar de uma classe `abstract` quando não há nenhuma implementação padrão a herdar — isto é, nenhum campo e nenhuma implementação padrão de método. Assim como as classes `public abstract`, as interfaces são geralmente tipos `public`. Assim como uma classe `public`, uma interface `public` deve ser declarada em um arquivo com o mesmo nome da interface e a extensão de arquivo `.java`.

Interfaces de marcação

Veremos no Capítulo 17, "Arquivos, fluxos e serialização de objetos", a noção de "interfaces de marcação" — interfaces vazias que não têm *nenhum* método ou valor constante. Elas são usadas para adicionar relacionamentos "*é um*" a classes. Por exemplo, no Capítulo 17 discutiremos um mecanismo do Java chamado de serialização de objetos, que pode converter objetos em representações de bytes e então converter essas representações de byte de volta em objetos. A fim de ativar esse mecanismo para funcionar com objetos, basta marcá-los como `Serializable` adicionando `implements Serializable` ao fim da primeira linha de declaração da classe. Então, todos os objetos da classe têm o relacionamento *é um* com `Serializable`.

10.7.1 Desenvolvendo uma hierarquia Payable

Para construir um aplicativo que pode determinar pagamentos para empregados, faturas e coisas parecidas, primeiro criamos a interface `Payable`, que contém o método `getPaymentAmount` que retorna um valor `double` que deve ser pago para um objeto de qualquer classe que implementa a interface. O método `getPaymentAmount` é uma versão de uso geral do método `earnings` da hierarquia `Employee` — o método `earnings` calcula especificamente um valor de pagamento para uma `Employee`, enquanto `getPaymentAmount` pode ser aplicado a um amplo intervalo de objetos não relacionados. Após declarar a interface `Payable`, apresentamos a classe `Invoice`, que `implementa` a interface `Payable`. Então, modificamos a classe `Employee` para também implementar a interface `Payable`. Por fim, atualizamos a subclasse `Employee` `SalariedEmployee` para "ajustá-la" à hierarquia `Payable` renomeando o método `SalariedEmployee` `earnings` como `getPaymentAmount`.

Boa prática de programação 10.2

Ao declarar um método em uma interface, escolha um nome de método que descreva o propósito do método de uma maneira geral, pois o método pode ser implementado por muitas classes não relacionadas.

As classes `Invoice` e `Employee` representam aspectos para os quais a empresa deve ser capaz de calcular um valor de pagamento. Ambas as classes implementam a interface `Payable`, assim um programa pode invocar o método `getPaymentAmount`. Como veremos a seguir, isso permite o processamento polimórfico de `Invoices` e `Employees` necessárias para nosso aplicativo de contas a pagar da empresa.

O diagrama de classes da UML na Figura 10.10 mostra a hierarquia utilizada no nosso aplicativo de contas a pagar. A hierarquia inicia com a interface `Payable`. A UML separa uma interface de outras classes colocando a palavra "interface" entre o símbolo de aspas francesas (« e ») acima do nome da interface. A UML expressa o relacionamento entre uma classe e uma interface por meio de um relacionamento conhecido como **realização**. Diz-se que uma classe "realiza", ou implementa, os métodos de uma interface. Um diagrama de classe modela uma realização como uma seta tracejada com uma ponta oca apontando da implementação da classe para a interface. O diagrama na Figura 10.10 indica que as classes `Invoice` e `Employee` realizam (isto é, implementam) a interface `Payable`. Observe que, como no diagrama de classe da Figura 10.2, a classe `Employee` aparece em itálico, indicando que é uma classe abstrata. A classe concreta `SalariedEmployee` estende `Employee` e *herda seu relacionamento de realização da superclasse* com a interface `Payable`.

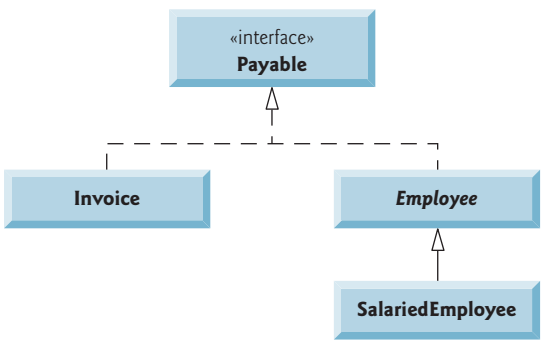


Figura 10.10 | Diagrama de classe em UML da hierarquia da interface `Payable`.

10.7.2 Interface Payable

A declaração da interface `Payable` inicia na Figura 10.11 na linha 4. A interface `Payable` contém o método `getPaymentAmount` `public` `abstract` (linha 6). Note que o método não é declarado explicitamente `public` ou `abstract`. Métodos de interface sempre são `public` e `abstract`, de modo que não seja necessário declará-los como tais. A interface `Payable` contém apenas um método — interfaces podem conter um número qualquer de métodos. Além disso, o método `getPaymentAmount` não tem parâmetros, mas os métodos de interface *podem* ter parâmetros. Observe que as interfaces também podem conter campos que são implicitamente `final` e `static`.

```
1  // Figura 10.11: Payable.java
2  // Declaração da interface Payable.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calcula pagamento; nenhuma implementação
7  } // fim da interface Payable
```

Figura 10.11 | Declaração da interface `payable`.

10.7.3 Classe Invoice

Agora, criaremos a classe `Invoice` (Figura 10.12) para representar uma fatura simples com informações de cobrança apenas para um tipo de peça. A classe declara como `private` as variáveis de instância `partNumber`, `partDescription`, `quantity` e `pricePerItem` (nas linhas 6–9) que indicam o número da peça, uma descrição da peça, a quantidade de peças pedida e o preço por item. A classe `Invoice` também contém um construtor (linhas 12–19), métodos *get* e *set* (linhas 22–67) que manipulam as variáveis de instância da classe e um método `toString` (linhas 70–76) que retorna uma representação `St3ring` de um objeto `Invoice`. Note que os métodos `setQuantity` (linhas 46–49) e `setPricePer-Item` (linhas 58–61) asseguram que `quantity` e `pricePerItem` obtêm somente valores não negativos.

```
1  // Figura 10.12: Invoice.java
2  // Classe Invoice que implementa Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // construtor com quatro argumentos
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // valida e armazena a quantidade
18       setPricePerItem( price ); // valida e armazena o preço por item
19    } // fim do construtor Invoice de quatro argumentos
20
21    // configura número de peças
22    public void setPartNumber( String part )
23    {
24       partNumber = part; // deve validar
25    } // fim do método setPartNumber
26
27    // obtém o número da peça
28    public String getPartNumber()
29    {
30       return partNumber;
31    } // fim do método getPartNumber
32
33    // configura a descrição
34    public void setPartDescription( String description )
35    {
36       partDescription = description; // deve validar
37    } // fim do método setPartDescription
38
39    // obtém a descrição
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // fim do método getPartDescription
44
45    // configura a quantidade
46    public void setQuantity( int count )
47    {
48       quantity = ( count < 0 ) ? 0 : count; // a quantidade não pode ser negativa
49    } // fim do método setQuantity
50
51    // obtém a quantidade
52    public int getQuantity()
53    {
54       return quantity;
55    } // fim do método getQuantity
56
57    // configura preço por item
58    public void setPricePerItem( double price )
59    {
```

```
60          pricePerItem = ( price < 0.0 ) ? 0.0 : price; // valida preço
61       } // fim do método setPricePerItem
62
63       // obtém preço por item
64       public double getPricePerItem()
65       {
66          return pricePerItem;
67       } // fim do método getPricePerItem
68
69       // retorno da representação de String do objeto Invoice
70       @Override
71       public String toString()
72       {
73          return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
74          "invoice", "part number", getPartNumber(), getPartDescription(),
75          "quantity", getQuantity(), "price per item", getPricePerItem() );
76       } // fim do método toString
77
78       // método requerido para executar o contrato com a interface Payable
79       @Override
80       public double getPaymentAmount()
81       {
82          return getQuantity() * getPricePerItem(); // calcula custo total
83       } // fim do método getPaymentAmount
84    } // fim da classe Invoice
```

Figura 10.12 | A classe `Invoice` que implementa `Payable`.

A linha 4 indica que a classe `Invoice` implementa a interface `Payable`. Como ocorre com todas as classes, a classe `Invoice` também estende implicitamente `Object`. O Java não permite que subclasses estendam mais de uma superclasse, mas permite que uma classe estenda uma superclasse e implemente quantas interfaces ela precisar. Para implementar mais de uma interface, utilize uma lista separada por vírgulas de nomes de interfaces depois da palavra-chave `implements` na declaração de classe, como em:

```
public class NomeDaClasse extends NomeDaSuperClasse implements PrimeiraInterface, SegundaInterface, ...
```

Observação de engenharia de software 10.7

Todos os objetos de uma classe que implementam múltiplas interfaces têm o relacionamento do tipo "é um" *com cada tipo de interface implementado.*

A classe `Invoice` implementa um método na interface `Payable` — o método `getPaymentAmount` é declarado nas linhas 79–83. O método calcula o pagamento total necessário para pagar a fatura. O método multiplica os valores de `quantity` e `pricePerItem` (obtidos por meio dos métodos *get* apropriados) e retorna o resultado (linha 82). Esse método satisfaz o requisito de implementação para esse método na interface `Payable` — cumprimos o contrato de interface com o compilador.

10.7.4 Modificando a classe `Employee` para implementar a interface `Payable`

Agora, modificaremos a classe `Employee` para que ela implemente a interface `Payable`. A Figura 10.13 contém a classe modificada, que é idêntica àquela da Figura 10.4 com duas exceções. Primeiro, a linha 4 da Figura 10.13 indica que a classe `Employee` agora `implementa` a interface `Payable`. Então devemos renomear `earnings` como `getPaymentAmount` em todas as partes da hierarquia `Employee`. Entretanto, como com o método `earnings` na versão da classe `Employee` na Figura 10.4, não faz sentido implementar o método `getPaymentAmount` na classe `Employee`, porque não podemos calcular o pagamento dos lucros devidos a um `Employee` geral — primeiro devemos saber o tipo específico de `Employee`. Na Figura 10.4, declaramos o método `earnings` como `abstract` por essa razão, portanto, a classe `Employee` teve de ser declarada `abstract`. Isso forçou cada subclasse concreta `Employee` a sobrescrever `earnings` com uma implementação.

Na Figura 10.13, tratamos essa situação de uma maneira diferente. Lembre-se de que, quando uma classe implementa uma interface, a classe faz um contrato com o compilador afirmando que a classe implementará cada um dos métodos na interface ou que a classe será declarada `abstract`. Se a última opção for escolhida, não precisamos declarar os métodos de interface como `abstract` na classe `abstract` — eles já são implicitamente declarados como tais na interface. Qualquer subclasse concreta da classe `abstract` deve implementar os métodos de interface para cumprir o contrato da superclasse com o compilador. Se a subclasse não fizer isso, ela também deverá ser declarada `abstract`. Como indicado pelos comentários nas linhas 62–63, a classe `Employee` da Figura 10.13 não implementa o método `getPaymentAmount`, portanto a classe é declarada `abstract`. Cada subclasse `Employee` direta herda o contrato da superclasse para implementar o método `getPaymentAmount` e assim deve implementar esse método para tornar-se uma classe concreta na qual os objetos podem ser instanciados. Uma classe que estende uma das subclasses concretas de `Employee` herdará uma implementação de `getPaymentAmount` e assim também será uma classe concreta.

```
1  // Figura 10.13: Employee.java
2  // Superclasse abstrata Employee implementa Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // construtor com três argumentos
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // fim do construtor Employee com três argumentos
17
18    // configura o nome
19    public void setFirstName( String first )
20    {
21       firstName = first; // deve validar
22    } // fim do método setFirstName
23
24    // retorna o nome
25    public String getFirstName()
26    {
27       return firstName;
28    } // fim do método getFirstName
29
30    // configura o sobrenome
31    public void setLastName( String last )
32    {
33       lastName = last; // deve validar
34    } // fim do método setLastName
35
36    // retorna o sobrenome
37    public String getLastName()
38    {
39       return lastName;
40    } // fim do método getLastName
41
42    // configura o CIC
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // deve validar
46    } // fim do método setSocialSecurityNumber
47
48    // retorna o CIC
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // fim do método getSocialSecurityNumber
53
54    // retorna a representação de String do objeto Employee
55    @Override
56    public String toString()
57    {
58       return String.format( "%s %s\nsocial security number: %s",
59          getFirstName(), getLastName(), getSocialSecurityNumber() );
60    } // fim do método toString
61
62    // Nota: não implementamos o método Payable getPaymentAmount aqui, mas essa
63    // classe deve ser declarada abstrata para evitar um erro de compilação.
64 } // fim da classe abstrata Employee
```

Figura 10.13 | A classe `Employee` que implementa `Payable`.

10.7.5 Modificando a classe SalariedEmployee para uso na hierarquia Payable

A Figura 10.14 contém uma classe `SalariedEmployee` modificada que estende `Employee` e cumpre o contrato da superclasse `Employee` para implementar o método `Payable` `getPaymentAmount`. Essa versão de `SalariedEmployee` é idêntica à da Figura 10.5, mas substitui o método `earnings` pelo método `getPaymentAmount` (linhas 30–34). Lembre-se de que a versão do método `Payable` tem um nome mais geral que poderia ser aplicado a classes díspares. As subclasses `Employee` remanescentes (por exemplo, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`) também devem ser modificadas a fim de conter o método `getPaymentAmount` no lugar do `earnings` para refletir o fato de que `Employee` agora implementa `Payable`. Deixamos essas modificações como um exercício (Exercício 10.11) e utilizamos apenas `SalariedEmployee` no nosso programa de teste aqui. O Exercício 10.12 pede para você implementar a interface `Payable` em toda a hierarquia de classe `Employee` das figuras 10.4 a 10.9 sem modificar as subclasses `Employee`.

```
1  // Figura 10.14: SalariedEmployee.java
2  // Classe SalariedEmployee estende Employee que implementa Payable.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // construtor com quatro argumentos
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // passa para o construtor Employee
13       setWeeklySalary( salary ); // valida e armazena o salário
14    } // fim do construtor SalariedEmployee com quatro argumentos
15
16    // configura o salário
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // fim do método setWeeklySalary
21
22    // retorna o salário
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // fim do método getWeeklySalary
27
28    // calcula vencimentos; implementa o método Payable da
29    // interface que era abstrata na superclasse Employee
30    @Override
31    public double getPaymentAmount()
32    {
33       return getWeeklySalary();
34    } // fim do método getPaymentAmount
35
36    // retorna a representação String do objeto SalariedEmployee
37    @Override
38    public String toString()
39    {
40       return String.format( "salaried employee: %s\n%s: $%,.2f",
41          super.toString(), "weekly salary", getWeeklySalary() );
42    } // fim do método toString
43 } // fim da classe SalariedEmployee
```

Figura 10.14 | A classe `SalariedEmployee` que implementa o método `getPaymentAmount` da interface `Payable`.

Quando uma classe implementa uma interface, o mesmo relacionamento *é um* fornecido por herança se aplica. A classe `Employee` implementa `Payable`, portanto podemos dizer que um `Employee` *é um* `Payable`. De fato, objetos de quaisquer classes que estendem `Employee` também são objetos `Payable`. Objetos `SalariedEmployee`, por exemplo, são objetos `Payable`. Os objetos de quaisquer subclasses da classe que `implementa` a interface também podem ser pensados como objetos do tipo de interface. Assim, da mesma forma como podemos atribuir a referência de um objeto `SalariedEmployee` a uma variável `Employee` da superclasse, podemos atribuir a referência de um objeto `SalariedEmployee` a uma variável `Payable` da interface. `Invoice` `implements` `Payable`, portanto um objeto `Invoice` também *é um* objeto `Payable`, e podemos atribuir a referência de um objeto `Invoice` a uma variável `Payable`.

Observação de engenharia de software 10.8

Quando um parâmetro de método é declarado com uma superclasse ou tipo de interface, o método processa o objeto recebido polimorficamente como um argumento.

Observação de engenharia de software 10.9

Utilizando uma referência de superclasse, podemos invocar polimorficamente qualquer método declarado na superclasse e suas superclasses (por exemplo, a classe `Object`). Utilizando uma referência de interface, podemos invocar polimorficamente qualquer método declarado na interface, suas superinterfaces (uma interface pode estender outra) e na classe `Object` — uma variável de tipo de interface deve referenciar um objeto para chamar métodos, e todos os objetos têm os métodos da classe `Object`.

10.7.6 Utilizando a interface `Payable` para processar `Invoice` e `Employee` polimorficamente

`PayableInterfaceTest` (Figura 10.15) demonstra que a interface `Payable` pode ser utilizada para processar um conjunto de classes `Invoice` e `Employee` polimorficamente em um único aplicativo. A linha 9 declara `payableObjects` e lhe atribui um array de quatro variáveis `Payable`. As linhas 12–13 atribuem as referências de objetos `Invoice` aos dois primeiros elementos de `payableObjects`. As linhas 14–17 atribuem então as referências de objetos `SalariedEmployee` aos dois elementos de `payableObjects` remanescentes. Essas atribuições são permitidas porque um `Invoice` *é um* `Payable`, um `SalariedEmployee` *é um* `Employee` e um `Employee` *é um* `Payable`. As linhas 23–29 usam a instrução `for` aprimorada para processar polimorficamente cada objeto `Payable` em `payableObjects`, imprimindo o objeto como uma `String`, com a quantidade de pagamento devido. Observe que a linha 27 invoca o método `toString` via uma interface de referência `Payable`, mesmo que `toString` não seja declarado na interface `Payable` — todas as referências (incluindo as dos tipos de interface) referenciam objetos que estendem `Object` e, portanto, contêm um método `toString`. (Note que `toString` também pode ser invocado implicitamente aqui.) A linha 28 invoca o método `Payable` `getPaymentAmount` para obter a quantia de pagamento para cada objeto em `payableObjects`, independentemente do tipo real do objeto. A saída revela que as chamadas de método nas linhas 27–28 invocam a implementação da classe apropriada dos métodos `toString` e `getPaymentAmount`. Por exemplo, quando `currentPayable` referencia uma `Invoice` durante a primeira iteração do loop `for`, os métodos `toString` e `getPaymentAmount` da classe `Invoice` são executados.

```
1  // Figura 10.15: PayableInterfaceTest.java
2  // Testa a interface Payable.
3
4  public class PayableInterfaceTest
5  {
6     public static void main( String[] args )
7     {
8        // cria array Payable de quatro elementos
9        Payable[] payableObjects = new Payable[ 4 ];
10
11       // preenche o array com objetos que implementam Payable
12       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14       payableObjects[ 2 ] =
15          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16       payableObjects[ 3 ] =
17          new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Invoices and Employees processed polymorphically:\n" );
21
22       // processa genericamente cada elemento no array payableObjects
23       for ( Payable currentPayable : payableObjects )
24       {
25          // gera saída de currentPayable e o pagamento apropriado
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27             currentPayable.toString(),
28             "payment due", currentPayable.getPaymentAmount() );
29       } // for final
30    } // fim de main
31 } // fim da classe PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Figura 10.15 | Programa de teste da interface `Payable` que processa `Invoices` e `Employees` polimorficamente.

10.7.7 Interfaces comuns da Java API

Nesta seção, apresentamos uma visão geral de várias interfaces comuns encontradas na Java API. O poder e a flexibilidade das interfaces são usados frequentemente por toda a Java API. Essas interfaces são implementadas e usadas da mesma maneira como as interfaces que você cria (por exemplo, a interface `Payable` na Seção 10.7.2). As interfaces da Java API permitem que você use suas próprias classes dentro das estruturas fornecidas pelo Java, como comparar os objetos dos seus próprios tipos e criar tarefas que podem executar concorrentemente com outras tarefas no mesmo programa. A Figura 10.16 apresenta uma breve visão geral de algumas das interfaces mais populares da Java API usadas neste livro.

Interface	Descrição
`Comparable`	O Java contém vários operadores de comparação (por exemplo, <, <=, >, >=, ==, !=) que permitem comparar valores primitivos. Contudo, esses operadores não podem ser utilizados para comparar objetos. A interface `Comparable` é utilizada para permitir que objetos de uma classe que `implementam` a interface sejam comparados entre si. A interface `Comparable` é comumente utilizada para ordenar objetos em uma coleção, como um array. Utilizamos `Comparable` no Capítulo 20, "Coleções genéricas", e no Capítulo 21, "Classes e métodos genéricos".
`Serializable`	Uma interface usada para identificar classes cujos objetos podem ser gravados em (isto é, serializados) ou lidos de (isto é, desserializados) algum tipo de armazenamento (por exemplo, arquivo em disco, campo de banco de dados) ou transmitidos por uma rede. Usamos `Serializable` nos capítulos 17, "Arquivos, fluxos e serialização de objetos", e 27, "Redes".
`Runnable`	Implementada por qualquer classe por meio da qual objetos dessa classe devem ser capazes de executar em paralelo utilizando uma técnica chamada multithreading (discutida no Capítulo 26, Multithreading). A interface contém um método, `run`, que descreve o comportamento de um objeto quando executado.
Interfaces listener de eventos com GUIs	Você utiliza interfaces gráficas com o usuário (GUIs) todos os dias. No navegador Web, você poderia digitar o endereço de um site Web a visitar, ou clicar em um botão para voltar a um site anterior. O navegador responde à sua interação e realiza a tarefa desejada. Sua interação é conhecida como um evento e o código que o navegador utiliza para responder a um evento é conhecido como um handler de eventos. No Capítulo 14, Componentes GUI: Parte 1, e no Capítulo 25, Componentes GUI: Parte 2, você aprenderá a construir GUI e rotinas de tratamento de evento que respondem a interações de usuário. Handlers de eventos são declarados em classes que implementam uma interface ouvinte de eventos apropriada. Cada interface ouvinte de eventos especifica um ou mais métodos que devem ser implementados para responder a interações de usuário.
`SwingConstants`	Contém um conjunto de constantes utilizado em programação de GUI para posicionar elementos da GUI na tela. Exploramos a programação de GUI nos capítulos 14 e 25.

Figura 10.16 | Interfaces comuns da Java API.

10.8 (Opcional) Estudo de caso de GUIs e imagens gráficas: desenhando com polimorfismo

Você pode ter notado no programa de desenho criado no Exercício 8.1 de Estudo de Caso sobre GUIs e imagens gráficas (e modificado no Exercício 9.1 do Estudo de caso sobre GUIs e imagens gráficas) que as classes de forma têm muitas semelhanças. Com a herança, podemos "dividir" os recursos comuns de todas as três classes e colocá-los em uma única superclasse de forma. Portanto, utilizando variáveis do tipo de superclasse, podemos manipular polimorficamente os objetos de todos os três tipos de forma. Remover a redundância no código resultará em um programa menor, mais flexível e mais fácil de manter.

Exercícios do Estudo de caso sobre GUIs e imagens gráficas

10.1 Modifique as classes `MinhaLinha`, `MinhaOval` e `MeuRetângulo` dos exercícios 8.1 e 9.1 do Estudo de caso sobre GUIs e imagens gráficas para criar a hierarquia de classes da Figura 10.17. Classes da hierarquia `MinhaForma` devem ser classes de forma "inteligentes" que sabem como se desenhar (se fornecidas com um objeto `Graphics` que lhes diz onde desenhar). Depois que o programa cria um objeto a partir dessa hierarquia, ele pode manipulá-lo polimorficamente para o restante do seu tempo de vida como um `MinhaForma`.

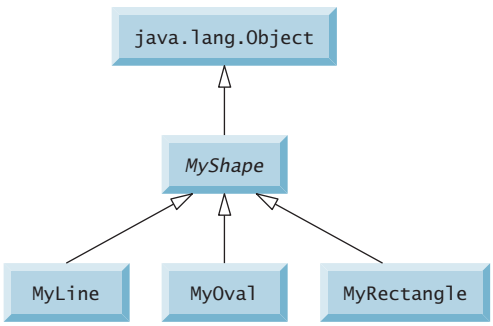


Figura 10.17 | Hierarquia `MinhaForma`.

Na sua solução, a classe `MinhaForma` na Figura 10.17 *deve* ser `abstract`. Como `MinhaForma` representa qualquer forma em geral, você não pode implementar um método `desenhar` sem saber exatamente qual é a forma. Os dados que representam as coordenadas e a cor das formas na hierarquia devem ser declarados como membros `private` da classe `MinhaForma`. Além dos dados comuns, a classe `MinhaForma` deve declarar os métodos a seguir:

a) Um construtor sem argumentos que configura todas as coordenadas da forma como `0` e a cor como `Cor.PRETO`.

b) Um construtor que inicializa as coordenadas e cores com os valores dos argumentos fornecidos.

c) Métodos *set* para as coordenadas e cores individuais que permitem ao programador configurar quaisquer dados para uma forma na hierarquia de maneira independente.

d) Métodos *get* para as coordenadas e cores individuais que permitem ao programador recuperar quaisquer dados para uma forma na hierarquia de maneira independente.

e) O método `abstract`

```
public abstract void draw( Graphics g );
```

que será chamado no método `pintarComponente` do programa para desenhar uma forma na tela.

Para assegurar um encapsulamento adequado, todos os dados na classe `MinhaForma` devem ser `private`. Isso exige declarar métodos *set* e *get* adequados para manipular os dados. A classe `MinhaLinha` deve fornecer um construtor sem argumentos e um construtor com argumentos para as coordenadas e cores. As classes `MinhaOval` e `MeuRetângulo` devem fornecer um construtor sem argumentos e um construtor com argumentos para as coordenadas e as cores e determinar se a forma será preenchida. O construtor sem argumentos deve, além de configurar os valores padrão, configurar a forma como uma forma não preenchida.

Você pode desenhar linhas, retângulos e ovais se conhecer dois pontos no espaço. As linhas exigem as coordenadas *x1*, *y1*, *x2* e *y2*. O método `drawLine` da classe `Graphics` ligará os dois pontos fornecidos com uma linha. Se você tiver os mesmos quatro valores de coordenadas (*x1*, *y1*, *x2* e *y2*) para ovais e retângulos, você pode calcular os quatro argumentos necessários para desenhá-los. Cada uma requer um valor da coordenada *x* superior esquerda (o menor dos dois valores da coordenada *x*), um valor da coordenada y superior esquerda (o menor dos dois valores da coordenada *y*), uma largura (o valor absoluto da diferença entre os dois valores da coordenada *x*) e uma altura (o valor absoluto da diferença entre os dois valores da coordenada *y*). Retângulos e ovais também devem ter um flag `filled` que determina se a forma deve ser desenhada como uma forma preenchida.

Não deve haver variáveis `MinhaLinha`, `MinhaOval` ou `MeuRetângulo` no programa — somente variáveis `MinhaForma` que contenham referências aos objetos `MinhaLinha`, `MinhaOval` ou `MeuRetângulo`. O programa deve gerar formas aleatórias e armazená-las em um array do tipo `MinhaForma`. O método `paintComponent` deve percorrer o array `MinhaForma` e desenhar cada forma (isto é, chamando polimorficamente o método `desenhar` de cada forma).

Permita que o usuário especifique (via um diálogo de entrada) o número de formas a gerar. O programa então irá gerar e exibir as formas juntamente com uma barra de status que informa ao usuário quantas de cada forma foram criadas.

10.2 (***Modificação de aplicativo de desenho***) No Exercício 10.1, você criou uma hierarquia `MinhaForma` na qual as classes `MinhaLinha`, `MinhaOval` ou `MeuRetângulo` estendem `MinhaForma` diretamente. Se sua hierarquia foi projetada adequadamente, você deve ser capaz de ver as semelhanças entre as classes `MinhaOval` e `MeuRetângulo`. Reprojete e reimplemente o código para as classes `MinhaOval` e `MeuRetângulo` para "fatorar" os recursos comuns na classe abstrata `MinhaFormaLimitada` a fim de produzir a hierarquia na Figura 10.18.

A classe `MinhaFormaLimitada` deve declarar dois construtores que simulam os construtores da classe `MinhaForma`, somente com um parâmetro adicionado para configurar se a forma é preenchida. A classe `MinhaFormaLimitada` também deve declarar os métodos *get* e *set* para manipular o flag preenchido e os métodos que calculam a coordenada *x* superior esquerda , a coordenada *y* superior esquerda, largura e altura. Lembre-se de que os valores necessários para desenhar uma oval ou um retângulo podem ser calculados a partir de duas coordenadas (*x*, *y*). Se você projetou adequadamente, as novas classes `MinhaOval` e `MeuRetângulo` devem ter cada uma dois construtores e um método `desenhar`.

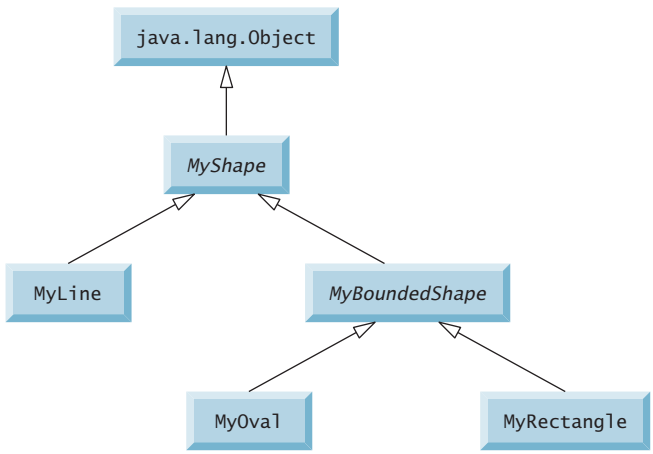


Figura 10.18 | Hierarquia `MinhaForma` com `MinhaFormaLimitada`.

10.9 Conclusão

Este capítulo introduziu o polimorfismo — a capacidade de processar objetos que compartilham a mesma superclasse em uma hierarquia de classes como se todos fossem objetos das superclasses. O capítulo discutiu como o polimorfismo torna sistemas extensíveis e sustentáveis e então demonstrou como utilizar métodos sobrescritos para executar o comportamento polimórfico. Introduzimos as classes abstratas, que permitem-lhe fornecer uma superclasse apropriada a partir da qual outras classes podem herdar. Você aprendeu que uma classe abstrata pode declarar métodos abstratos que cada subclasse deve implementar para tornar-se uma classe concreta e que um programa pode utilizar variáveis de uma classe abstrata para invocar implementações das subclasses de métodos abstratos polimorficamente. Você também aprendeu a determinar um tipo de objeto em tempo de execução. Discutimos os conceitos dos métodos e classes `final`. Por fim, o capítulo discutiu a declaração e implementação de uma interface como uma outra maneira de alcançar o comportamento polimórfico.

Você agora deve conhecer classes, objetos, encapsulamento, herança, interfaces e polimorfismo — os aspectos mais essenciais da programação orientada a objetos.

No próximo capítulo, você aprenderá sobre as exceções, úteis para tratamento de erros durante a execução de um programa. O tratamento de exceções fornece programas mais resistentes.

Resumo

Seção 10.1 Introdução

- O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse em uma hierarquia de classes como se todos eles fossem objetos da superclasse; isso pode simplificar a programação.
- Com o polimorfismo, podemos projetar e implementar sistemas que são facilmente extensíveis — novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do programa, contanto que as novas classes façam parte da mesma hierarquia de herança. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que você adiciona à hierarquia.

Seção 10.3 Demonstrando um comportamento polimórfico

- Quando o compilador encontra uma chamada de método feita por meio de uma variável, ele determina se o método pode ser chamado verificando o tipo de classe da variável. Se essa classe contiver a declaração de método apropriada (ou estender uma), a chamada é compilada. Em tempo de execução, o tipo do objeto que a variável referencia determina o método real a utilizar.

Seção 10.4 Classes e métodos abstratos

- As classes abstratas não podem ser utilizadas para instanciar objetos, porque são incompletas.
- O propósito principal de uma classe abstrata é fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e, assim, compartilhar um design comum.
- As classes que podem ser utilizadas para instanciar objetos são chamadas classes concretas. Essas classes fornecem implementações de cada método que elas declaram (algumas implementações podem ser herdadas).
- Nem todas as hierarquias de herança contêm classes abstratas. Entretanto, programadores costumam escrever código de cliente que utiliza apenas tipos abstratos de superclasse para reduzir dependências do código de cliente em um intervalo de tipos específicos de subclasse.
- As classes abstratas às vezes constituem vários níveis da hierarquia.
- Você cria uma classe abstrata declarando-a com a palavra-chave `abstract`. Uma classe abstrata normalmente contém um ou mais métodos abstratos.
- Métodos abstratos não fornecem implementações.
- Uma classe que contém qualquer método abstrato deve ser declarada como uma classe abstrata mesmo se contiver alguns métodos concretos. Cada subclasse concreta de uma superclasse abstrata deve fornecer implementações concretas dos métodos abstratos da superclasse.
- Os construtores e métodos `static` não podem ser declarados `abstract`.
- Você *pode* utilizar superclasses abstratas para declarar variáveis que podem armazenar referências a objetos de qualquer classe concreta derivada dessas superclasses abstratas. Os programas em geral utilizam essas variáveis para manipular objetos de subclasse polimorficamente.
- O polimorfismo é particularmente eficaz para implementar sistemas em camadas de software.

Seção 10.5 Estudo de caso: sistema de folha de pagamentos utilizando polimorfismo

- Incluir um método `abstract` em uma superclasse força toda subclasse direta da superclasse a sobrescrever o método `abstract` para se tornar uma classe concreta. Isso permite que o projetista de hierarquia exija que cada subclasse concreta forneça uma implementação de método apropriada.
- A maioria das chamadas de método é resolvida em tempo de execução, com base no tipo de objeto que é manipulado. Esse processo é conhecido como vinculação dinâmica ou vinculação tardia.
- Uma variável de superclasse pode ser utilizada para invocar somente métodos declarados na superclasse (e a superclasse pode invocar versões sobrescritas dessas na subclasse).
- O operador `instanceof` pode ser utilizado para determinar se o tipo de um objeto particular tem um relacionamento *é um* com um tipo específico.
- Cada objeto em Java conhece sua própria classe e pode acessá-la por meio do método `getClass`, que todas as classes herdam da classe `Object`. O método `getClass` retorna um objeto do tipo `Class` (pacote `java.lang`), que contém as informações sobre o tipo do objeto, incluindo seu nome de classe.
- O relacionamento *é um* se aplica apenas entre a subclasse e suas superclasses, não vice-versa.

Seção 10.6 Métodos e classes `final`

- Um método declarado `final` em uma superclasse não pode ser sobrescrito em uma subclasse.
- Os métodos declarados `private` são implicitamente `final`, porque você não pode sobrescrevê-los em uma subclasse.
- Os métodos que são declarados `static` são implicitamente `final`.
- Uma declaração do método `final` nunca pode mudar, assim todas as subclasses utilizam a mesma implementação do método; e chamadas a métodos `final` são resolvidas em tempo de compilação — isso é conhecido como vinculação estática.
- Depois que o compilador sabe que os métodos `final` não podem ser sobrescritos, ele pode otimizar os programas removendo chamadas a métodos `final` e substituindo-as pelo código expandido das suas declarações em cada local da chamada de método — técnica conhecida como colocar o código inline.
- Uma classe que é declarada `final` não pode ser uma superclasse.
- Todos os métodos em uma classe `final` são implicitamente `final`.

Seção 10.7 Estudo de caso: criando e utilizando interfaces

- Uma interface especifica *quais* operações são permitidas, mas não *como* as operações são realizadas.
- Uma interface Java descreve o conjunto de métodos que pode ser chamado em um objeto.
- Uma declaração de interface começa com a palavra-chave `interface`.
- Todos os membros de interface devem ser `public` e as interfaces não podem especificar nenhum detalhe de implementação, como declarações de métodos concretos e variáveis de instância.
- Todos os métodos declarados em uma interface são implicitamente métodos `public abstract` e todos os campos são implicitamente `public`, `static` e `final`.
- Para utilizar uma interface, uma classe concreta deve especificar que ela `implementa` a interface e deve declarar cada método de interface com a assinatura especificada na declaração de interface. Uma classe que não implementa todos os métodos da interface deve ser declarada `abstract`.

- Implementar uma interface é como assinar um contrato com o compilador que afirma "Irei declarar todos os métodos especificados pela interface ou irei declarar minha classe `abstract`".
- Em geral, uma interface é utilizada quando classes díspares (isto é, não relacionadas) precisam compartilhar métodos e constantes comuns. Isso permite que objetos de classes não relacionadas sejam processados polimorficamente — objetos de classes que implementam a mesma interface podem responder às mesmas chamadas de método.
- Você pode criar uma interface que descreve a funcionalidade desejada e, então, implementar a interface em quaisquer classes que requerem essa funcionalidade.
- Uma interface é geralmente utilizada em lugar de uma classe `abstract` quando não há implementação padrão a herdar — isto é, nenhuma variável de instância e nenhuma implementação de método padrão.
- Como ocorre com classes `public abstract`, interfaces são, em geral, tipos `public`, portanto normalmente são declaradas em arquivos próprios com o mesmo nome da interface e com a extensão `.java` no nome do arquivo.
- O Java não permite que subclasses herdem de mais de uma superclasse, mas permite que uma classe herde de uma superclasse e implemente mais de uma interface.
- Todos os objetos de uma classe que implementam múltiplas interfaces têm o relacionamento *é um* com cada tipo de interface implementado.
- Uma interface pode declarar constantes. As constantes são implicitamente `public`, `static` e `final`.

Terminologia

`abstract`, palavra-chave, 309
`Class`, classe, 320
classe abstrata, 309
classe concreta, 309
declaração de interface, 322
downcast, 307
`final`, classe, 321
`final`, método, 321
`getClass`, método da classe `Object`, 320
`getName`, método da classe `Class`, 320
implementa uma interface, 306
`implements`, palavra-chave, 322
`instanceof`, operador, 319
`interface`, palavra-chave, 322
Interfaces, 321
método abstrato, 309
polimorfismo, 305
realização na UML, 323
superclasse abstrata, 309
vinculação dinâmica, 319
vinculação estática, 321
vinculação tardia, 319

Exercícios de autorrevisão

10.1 Preencha as lacunas em cada uma das seguintes afirmações:
a) Se uma classe contiver pelo menos um método abstrato, é uma classe _____.
b) As classes a partir das quais os objetos podem ser instanciados são chamadas _____.
c) _____ envolve a utilização de uma variável de superclasse para invocar métodos nos objetos de superclasse e de subclasse, permitindo que você "programe no geral".
d) Os métodos que não são métodos de interface e que não fornecem implementações devem ser declarados com a palavra-chave _____.
e) Fazer uma coerção em uma referência armazenada em uma variável da superclasse para um tipo de subclasse é chamado _____.

10.2 Determine se cada uma das instruções a seguir é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
a) Todos os métodos em uma classe `abstract` devem ser declarados como métodos `abstract`.
b) Não é permitido invocar um método apenas de subclasse por uma variável de subclasse.
c) Se uma superclasse declarar um método como `abstract`, uma subclasse deverá implementar esse método.
d) Um objeto de uma classe que implementa uma interface pode ser pensado como um objeto desse tipo de interface.

Respostas dos exercícios de autorrevisão

10.1 a) abstratas. b) concretas. c) Polimorfismo. d) `abstract`. e) downcasting.

10.2 a) Falsa. Uma classe abstrata pode incluir métodos com implementações e métodos `abstract`. b) Falsa. Tentar invocar um método somente de subclasse com uma variável de superclasse é permitido. c) Falsa. Apenas uma subclasse concreta deve implementar o método. d) Verdadeira.

Exercícios

10.3 Como o polimorfismo permite programar "no geral" em vez de "no específico"? Discuta as vantagens cruciais da programação "no geral".

10.4 O que são métodos abstratos? Descreva as circunstâncias em que um método abstrato seria apropriado.

10.5 Como o polimorfismo promove extensibilidade?

10.6 Discuta quatro maneiras como você pode atribuir referências de superclasse e de subclasse a variáveis de superclasse e a tipos de subclasse.

10.7 Compare e contraste classes abstratas e interfaces. Por que você utilizaria uma classe abstrata? Por que você utilizaria uma interface?

10.8 (***Modificação do sistema de folha de pagamentos***) Modifique o sistema de folha de pagamentos das figuras 10.4 a 10.9 para incluir a variável de instância `private` `birthDate` na classe `Employee`. Utilize a classe `Date` da Figura 8.7 para representar o aniversário de um funcionário. Adicione métodos *get* à classe `Date`. Suponha que a folha de pagamentos seja processada uma vez por mês. Crie um array de variáveis `Employee` para armazenar referências aos vários objetos de funcionário. Em um loop, calcule a folha de pagamentos para cada `Employee` (polimorficamente) e adicione um bônus de US$ 100,00 à quantia da folha de pagamentos do funcionário se o mês atual for aquele em que o ocorre o aniversário do `Employee`.

10.9 (***Hierarquia de formas***) Implemente a hierarquia `Forma` mostrada na Figura 9.3. Cada `FormaBidimensional` deve conter o método `obterArea` para calcular a área da forma bidimensional. Cada `FormaTridimensional` deve ter métodos `obterArea` e `obterVolume` para calcular a área do volume e superfície, respectivamente, da forma tridimensional. Crie um programa que utiliza um array de referências `Forma` para objetos de cada classe concreta na hierarquia. O programa deve imprimir uma descrição de texto do objeto ao qual cada elemento no array se refere. Além disso, no loop que processa todas as formas no array, determine se cada forma é uma `FormaBidimensional` ou uma `FormaTridimensional`. Se for uma `FormaBidimensional`, exiba sua área. Se for uma `FormaTridimensional`, exiba sua área e volume.

10.10 (***Modificação do sistema de folha de pagamentos***) Modifique o sistema de folha de pagamentos das figuras 10.4 a 10.9 para incluir uma subclasse `PieceWorker` adicional de `Employee` que representa um funcionário cujo pagamento está baseado no número de peças de mercadorias produzido. A classe `PieceWorker` deve conter variáveis de instância `wage` `private` (para armazenar o salário do funcionário por peça) e `pieces` (para armazenar o número de peças produzidas). Forneça uma implementação concreta do método `earnings` na classe `PieceWorker` que calcula os vencimentos do funcionário multiplicando o número de peças produzidas pelo salário por peças. Crie um array de variáveis `Employee` para armazenar referências a objetos de cada classe concreta na nova hierarquia `Employee`. Para cada `Employee`, exiba sua representação de `String` e vencimentos.

10.11 (***Modificação do sistema de contas a pagar***) Neste exercício, modificamos o aplicativo de contas a pagar das figuras 10.11 a 10.15 a fim de incluir a funcionalidade completa do aplicativo de folha de pagamentos das figuras 10.4 a 10.9. O aplicativo ainda deve processar dois objetos `Invoice`, mas agora deve processar um objeto de cada uma das quatro subclasses `Employee`. Se o objeto atualmente processado for uma `BasePlusCommissionEmployee`, o aplicativo deverá aumentar o salário-base de `BasePlusCommissionEmployee` em 10%. Por fim, o aplicativo deve gerar a saída da quantia de pagamento para cada objeto. Complete os seguintes passos para criar o novo aplicativo:

a) Modifique as classes `HourlyEmployee` (Figura 10.6) e `CommissionEmployee` (Figura 10.7) para colocá-las na hierarquia `Payable` como subclasses da versão de `Employee` (Figura 10.13) que implementa `Payable`. [*Dica:* altere o nome do método `earnings` para `getPaymentAmount` em cada subclasse de modo que a classe satisfaça seu contrato herdado com a interface `Payable`.]

b) Modifique a classe `BasePlusCommissionEmployee` (Figura 10.8) para que ela estenda a versão da classe `CommissionEmployee` criada na parte (a).

c) Modifique `PayableInterfaceTest` (Figura 10.15) para processar polimorficamente duas `Invoices`, um `SalariedEmployee`, um `HourlyEmployee`, um `CommissionEmployee` e um `BasePlusCommissionEmployee`. Primeiro imprima uma representação `String` de cada objeto `Payable`. Em seguida, se um objeto for uma `BasePlusCommissionEmployee`, aumente seu salário-base em 10%. Por fim, gere a saída da quantia de pagamento para cada objeto `Payable`.

10.12 (***Modificação do sistema de contas a pagar***) É possível incluir as funcionalidades do aplicativo de folha de pagamentos (figuras 10.4 a 10.9) no aplicativo de contas a pagar sem modificar as subclasses `Employee` `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` ou `BasePlusCommissionEmployee`. Para tanto, você pode modificar a classe `Employee` (Figura 10.4) para implementar a interface `Payable` e declarar o método `getPaymentAmount` para invocar o método `earnings`. O método `getPaymentAmount` então seria herdado pelas subclasses na hierarquia `Employee`. Quando `getPaymentAmount` é chamado para determinado objeto de subclasse, ele invoca polimorficamente o método `earnings` apropriado dessa subclasse. Reimplemente o Exercício 10.11 utilizando a hierarquia original `Employee` do aplicativo de folha de pagamentos das figuras 10.4 a 10.9. Modifique a classe `Employee` como descrito nesse exercício, e *não* modifique nenhuma das subclasses da classe `Employee`.

Fazendo a diferença

10.13 (***Interface CarbonFootprint: polimorfismo***) Utilizando interfaces, como aprendeu neste capítulo, você pode especificar comportamentos semelhantes para classes possivelmente díspares. Governos e empresas no mundo inteiro se preocupam cada vez mais com as pegadas de carbono (*carbon footprints*, emissões anuais de gás carbônico na atmosfera) a partir de instalações que queimam vários tipos de combustíveis para aquecimento, veículos que queimam combustíveis para se mover, e assim por diante. Muitos cientistas culpam essas emissões de gases causadores do efeito estufa chamado aquecimento global. Crie três pequenas classes não relacionadas por herança — classes `Building`, `Car` e `Bicycle`. Dê a cada classe alguns atributos e comportamentos apropriados únicos que ela não tem em comum com outras classes. Escreva uma interface `CarbonFootprint` com um método `getCarbonFootprint`. Faça cada uma das suas classes implementar essa interface, para que seu método `getCarbonFootprint` calcule uma pegada de carbono apropriada para essa classe (consulte alguns sites Web que explicam como calcular as pegadas de carbono). Escreva um aplicativo que crie objetos de cada uma das três classes, coloque referências àqueles objetos em `ArrayList<CarbonFootprint>` e, então, itere pelo `ArrayList`, invocando polimorficamente método `getCarbonFootprint` de cada objeto. Para cada objeto, imprima algumas informações de identificação e as pegadas de carbono do objeto.

É do senso comum capturar um método e experimentá-lo. Se ele falhar, admita isso com franqueza e experimente outro. Mas, acima de tudo, tente algo.
— Franklin Delano Roosevelt

Jogai fora a metade que não presta,
para com a outra parte serdes pura.
— William Shakespeare

Se eles estiverem correndo e não olharem para onde estão indo
Tenho de sair de algum lugar e capturá-los.
— Jerome David Salinger

Rei dos reis, heroísmo sem limites, sorridente escapaste da cilada gigantesca do mundo?
— William Shakespeare

11

Tratamento de exceções

Objetivos

Neste capítulo, você aprenderá:

- O que são exceções.
- Como o tratamento de exceções e de erros funciona.
- A utilizar `try`, `throw` e `catch` para detectar, indicar e tratar exceções, respectivamente.
- A utilizar o bloco `finally` para liberar recursos.
- Como o desempilhamento permite que exceções não capturadas em um escopo sejam capturadas em outro.
- Como os rastreamentos de pilha ajudam na depuração.
- Como as exceções são organizadas em uma hierarquia de classes de exceção.
- A declarar novas classes de exceção.
- A criar exceções encadeadas que mantêm informações do rastreamento de pilha completo.

Sumário

11.1 Introdução
11.2 Visão geral do tratamento de erros
11.3 Exemplo de divisão por zero sem tratamento de exceções
11.4 Exemplo de tratamento de `ArithmeticException`s e `InputMismatchException`s
11.5 Quando utilizar o tratamento de exceções
11.6 Hierarquia de exceção Java
11.7 Bloco `finally`
11.8 Desempilhamento de pilha
11.9 `printStackTrace`, `getStackTrace` e `getMessage`
11.10 Exceções encadeadas
11.11 Declarando novos tipos de exceção
11.12 Pré-condições e pós-condições
11.13 Assertivas
11.14 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

11.1 Introdução

Neste capítulo, apresentamos o **tratamento de exceções**. Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa. O nome "exceção" significa que o problema não ocorre frequentemente. Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema. Isso ajuda a assegurar a robustez dos aplicativos que colaboram para o que é chamado de computação de missão crítica ou de negócios críticos. Um problema mais grave poderia impedir um programa de continuar executando normalmente, em vez de requerer que o problema seja informado antes de encerrar de uma maneira controlada. O Java permite-lhe lidar facilmente com o tratamento de exceções desde o princípio de um projeto. Os recursos apresentados neste capítulo o ajudam a escrever **programas robustos** e **tolerantes a falhas** (isto é, programas que podem lidar com os problemas à medida que eles surgem e continuar a executar). O tratamento de exceções do Java baseia-se em parte no artigo de Andrew Koenig e Bjarne Stroustrup, "Exception Handling for C++ (revised)".[1]

Dica de prevenção de erro 11.1
O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa.

Nos capítulos anteriores introduzimos exceções resumidamente. No Capítulo 7 você aprendeu que uma `ArrayIndexOutOfBoundsException` ocorre quando uma tentativa de acessar um elemento é feita depois de cada final de um array. Esse problema pode ocorrer, por exemplo, se houver um erro *off by one* em uma instrução `for` que manipula um array. No Capítulo 10, introduzimos a `ClassCastException`, que ocorre quando se tenta fazer coerção de um objeto que não tem um relacionamento *é um* com o tipo especificado no operador de coerção. Uma `NullPointerException` pode ocorrer quando uma referência `null` é utilizada onde se espera um objeto — por exemplo, ao se fazer uma tentativa de utilizar uma variável de instância do tipo por referência antes que ela seja inicializada. Por todo este texto você também utilizou a classe `Scanner` — que, como veremos neste capítulo, também pode causar exceções.

Começamos com uma visão geral de conceitos de tratamento de exceções e, em seguida, demonstramos técnicas básicas de tratamento de exceções. Mostramos essas técnicas em ação tratando uma exceção que ocorre quando um método tenta dividir um inteiro por zero. Logo depois introduzimos várias classes no topo da hierarquia de classe para tratamento de exceções do Java. Como você verá, somente as classes que estendem `Throwable` (pacote `java.lang`) direta ou indiretamente podem ser usadas com o tratamento de exceções. Mostramos então como utilizar exceções encadeadas. Ao invocar um método que indica uma exceção, você pode lançar outra exceção e encadear a original com a nova — isso permite adicionar informações específicas do aplicativo à exceção original. Em seguida, introduziremos pré-condições e pós-condições, que devem ser verdadeiras quando seus métodos são chamados e quando eles retornam, respectivamente. Por fim, apresentamos as assertivas, que você pode utilizar em tempo de desenvolvimento para ajudar a depurar o seu código.

[*Nota:* se você executar os exemplos neste capítulo a partir de um IDE, como o Eclipse ou NetBeans, as saídas dos seus programas poderiam ser diferentes daquelas mostradas aqui.]

11.2 Visão geral do tratamento de erros

Os programas costumam testar condições para determinar como a execução do programa deve prosseguir. Considere o pseudocódigo a seguir:

[1] KOENIG, A; STROUSTRUP, B. "Exception Handling for C++ (revised)", *Proceedings of the Usenix C++ Conference*, p. 149–176, São Francisco, abr. 1990.

Realize uma tarefa
Se a tarefa anterior não tiver sido executada corretamente
Realize processamento de erro
Realize a próxima tarefa
Se a tarefa anterior não tiver sido executada corretamente
Realize processamento de erro
...

Esse pseudocódigo inicia executando uma tarefa; então testa se ele executou corretamente. Se não tiver sido, realizamos processamento de erro. Caso contrário, continua com a próxima tarefa. Embora essa forma de tratamento de erro funcione, mesclar o programa e a lógica de tratamento de erro pode dificultar a leitura, modificação, manutenção e depuração dos programas — especialmente em aplicativos grandes.

Dica de desempenho 11.1

Se os problemas potenciais ocorrem raramente, mesclar o programa e a lógica do tratamento de erro pode degradar o desempenho do programa, porque o programa deve potencialmente realizar testes frequentes para determinar se a tarefa foi executada corretamente e se a próxima tarefa pode ser realizada.

O tratamento de exceções permite-lhe remover da "linha principal" de execução do programa o código de tratamento de erro, aprimorando a clareza do programa e destacando sua capacidade de modificação. Você pode decidir tratar a exceção que escolher — todas as exceções, todas as exceções de um certo tipo ou todas as exceções de um grupo de tipos relacionados (isto é, tipos de exceções que são relacionados por meio de uma superclasse em uma hierarquia de herança). Essa flexibilidade reduz a probabilidade de que erros serão negligenciados, tornando assim os programas mais robustos.

11.3 Exemplo de divisão por zero sem tratamento de exceções

Primeiro demonstramos o que acontece quando surgem erros em um aplicativo que não utiliza tratamento de exceções. A Figura 11.1 solicita ao usuário dois inteiros e os passa para o método `quotient`, que calcula o quociente inteiro e retorna um resultado `int`. Nesse exemplo, veremos que as exceções são **lançadas** (isto é, a exceção ocorre) quando um método detecta um problema e é incapaz de tratá-lo. Observe que `quotient` é declarado `static`, portanto `main` pode chamá-lo diretamente sem criar um objeto `DivideByZeroNoExceptionHandling`.

A primeira das três execuções de exemplo na Figura 11.1 mostra uma divisão bem-sucedida. Na segunda execução de exemplo, o usuário insere o valor `0` como o denominador. Observe que várias linhas de informações são exibidas em resposta a essa entrada inválida. Essas informações são conhecidas como **rastreamento de pilha**, que incluem o nome da exceção (`java.lang.ArithmeticException`) em uma mensagem descritiva que indica o problema que ocorreu e a pilha de chamadas de método (isto é, a cadeia de chamadas) no momento em que ela ocorreu. O rastreamento de pilha inclui o caminho de execução que resultou na exceção método por método. Essas informações o ajudam a depurar o programa. A primeira linha especifica a ocorrência de uma `ArithmeticException`. O texto depois do nome da exceção ("`/ by zero`") indica que essa exceção ocorreu como resultado de uma tentativa de dividir por zero. O Java não permite divisão por zero na aritmética de inteiros. Quando ocorrer a divisão por zero na aritmética de inteiroa, o Java lança uma **`ArithmeticException`**. `ArithmeticExceptions` podem surgir de vários problemas diferentes em aritmética, então os dados extras ("`/ by zero`") fornecem informações mais específicas. [*Nota:* o Java *realmente* permite a divisão por zero com valores de ponto flutuante. Um cálculo como esse resulta no valor positivo ou negativo infinito, que é representado em Java como um valor de ponto flutuante (mas aparece como a string `Infinity` ou `-Infinity`). Se 0.0 for dividido por 0.0, o resultado será NaN (not a number), que também é representado no Java como um valor de ponto flutuante (mas exibido como `NaN`).]

```
1  // Figura 11.1: DivideByZeroNoExceptionHandling.java
2  // Divisão de inteiro sem tratamento de exceções.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling
6  {
7     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
8     public static int quotient( int numerator, int denominator )
9     {
10       return numerator / denominator; // possível divisão por zero
11    } // fim do método quotient
12
13    public static void main( String[] args )
14    {
```

```
15          Scanner scanner = new Scanner( System.in ); // scanner para entrada
16
17          System.out.print( "Please enter an integer numerator: " );
18          int numerator = scanner.nextInt();
19          System.out.print( "Please enter an integer denominator: " );
20          int denominator = scanner.nextInt();
21
22          int result = quotient( numerator, denominator );
23          System.out.printf(
24             "\nResult: %d / %d = %d\n", numerator, denominator, result );
25       } // fim de main
26    } // fim da classe DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:20)
```

Figura 11.1 | Divisão de inteiro sem tratamento de exceções.

Iniciando da última linha do rastreamento de pilha, vemos que a exceção foi detectada na linha 22 do método `main`. Cada linha do rastreamento de pilha contém o nome da classe e o nome do método (`DivideByZeroNoExceptionHandling.main`) seguidos pelo nome de arquivo e número de linha (`DivideByZeroNoExceptionHandling.java:22`). Subindo o rastreamento de pilha, vemos que a exceção ocorre na linha 10, no método `quotient`. A linha superior da cadeia de chamadas indica o **ponto de lançamento** — o ponto inicial em que a exceção ocorre. O ponto de lançamento dessa exceção está na linha 10 do método `quotient`.

Na terceira execução, o usuário insere a string `"hello"` como o denominador. Observe novamente que um rastreamento de pilha é exibido. Isso informa a ocorrência de uma `InputMismatchException` (pacote `java.util`). Nossos exemplos anteriores que leem valores numéricos a partir do usuário assumiram que o usuário iria inserir um valor de inteiro adequado. Entretanto, às vezes, os usuários cometem erros e inserem valores não inteiros. Uma **`InputMismatchException`** ocorre quando o método `Scanner` `nextInt` recebe uma `string` que não representa um inteiro válido. Iniciando do fim do rastreamento de pilha, vemos que a exceção foi detectada na linha 20 do método `main`. Subindo o rastreamento de pilha, vemos que a exceção ocorreu no método `nextInt`. Observe que no lugar do nome de arquivo e número da linha, o texto `Unknown Source` é fornecido. Isso significa que os chamados símbolos de depuração que fornecem informações sobre o nome do arquivo e número da linha relativos à classe desse método não estavam disponíveis para a JVM — esse é tipicamente o caso das classes da Java API. Muitos IDEs têm acesso ao código-fonte da Java API e exibirão nomes de arquivo nos rastreamentos de pilha.

Observe que nas execuções de exemplo da Figura 11.1 quando exceções ocorrem e rastreamentos de pilha são exibidos, o programa também se fecha. Isso nem sempre ocorre no Java — às vezes um programa pode continuar, embora uma exceção tenha ocorrido e um rastreamento de pilha tenha sido impresso. Nesses casos, o aplicativo pode produzir resultados inesperados. A próxima seção demonstra como tratar essas exceções.

Na Figura 11.1 os dois tipos de exceções foram detectados no método `main`. No próximo exemplo, veremos como tratar essas exceções para permitir que o programa conclua sua execução normalmente.

11.4 Exemplo de tratamento de ArithmeticExceptions e InputMismatchExceptions

O aplicativo na Figura 11.2, que é baseado na Figura 11.1, utiliza o tratamento de exceções para processar quaisquer `ArithmeticExceptions` e `InputMistmatchExceptions` que surgirem. O aplicativo ainda solicita ao usuário dois inteiros e os passa para o método `quotient`, que calcula o quociente e retorna um resultado `int`. Essa versão do aplicativo utiliza tratamento de exceções de modo que se o usuário cometer um erro, o programa captura e trata (isto é, lida com) a exceção — nesse caso, permitindo ao usuário tentar inserir a entrada novamente. Observe que `quotient` é declarado `static`, portanto `main` pode chamá-lo diretamente sem instanciar um objeto `DivideByZeroWithExceptionHandling`.

```
1  // Figura 11.2: DivideByZeroWithExceptionHandling.java
2  // Tratando ArithmeticExceptions e InputMismatchExceptions.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling
7  {
8     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
9     public static int quotient( int numerator, int denominator )
10       throws ArithmeticException
11    {
12       return numerator / denominator; // possível divisão por zero
13    } // fim do método quotient
14
15    public static void main( String[] args )
16    {
17       Scanner scanner = new Scanner( System.in ); // scanner para entrada
18       boolean continueLoop = true; // determina se mais entradas são necessárias
19
20       do
21       {
22          try // lê dois números e calcula o quociente
23          {
24             System.out.print( "Please enter an integer numerator: " );
25             int numerator = scanner.nextInt();
26             System.out.print( "Please enter an integer denominator: " );
27             int denominator = scanner.nextInt();
28
29              int result = quotient( numerator, denominator );
30             System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                denominator, result );
32             continueLoop = false; // entrada bem-sucedida; fim do loop
33          } // fim do try
34          catch ( InputMismatchException inputMismatchException )
35          {
36             System.err.printf( "\nException: %s\n",
37                inputMismatchException );
38             scanner.nextLine(); // descarta entrada para o usuário poder tentar de novo
39             System.out.println(
40                "You must enter integers. Please try again.\n" );
41          } // fim do catch
42          catch ( ArithmeticException arithmeticException )
43          {
44             System.err.printf( "\nException: %s\n", arithmeticException );
45             System.out.println(
46                "Zero is an invalid denominator. Please try again.\n" );
47          } // fim do catch
48       } while ( continueLoop ); // fim da instrução do...while
49    } // fim de main
50 } // fim da classe DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception: java.util.InputMismatchException
You must enter integers. Please try again.
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
```

Figura 11.2 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`.

A primeira execução de exemplo na Figura 11.2 é uma execução bem-sucedida que não encontra nenhum problema. Na segunda execução, o usuário insere um denominador zero e uma exceção `ArithmeticException` ocorre. Na terceira execução, o usuário insere a string `"hello"` como o denominador e uma `InputMismatchException` ocorre. Para cada exceção, o usuário é informado do erro e é solicitado a tentar novamente, então um prompt pede-lhe para inserir dois novos inteiros. Em cada execução de exemplo, o programa termina a execução com sucesso.

A classe `InputMismatchException` é importada na linha 3. A classe `ArithmeticException` não precisa ser importada porque ela está no pacote `java.lang`. A linha 18 cria a variável `boolean continueLoop`, que é `true` se o usuário ainda não inseriu uma entrada válida. As linhas 20–48 solicitam repetidamente aos usuários uma entrada até que uma entrada válida seja recebida.

Incluindo código em um bloco `try`

As linhas 22–33 contêm um **bloco `try`**, que inclui o código que pode lançar (`throw`) uma exceção e o código que não deve ser executado se ocorrer uma exceção (isto é, se ocorrer uma exceção, o código restante no bloco `try` será pulado). Um bloco `try` consiste na palavra-chave `try` seguida por um bloco de código entre chaves (`{}`). [*Nota:* o termo "bloco `try`" às vezes só refere-se ao bloco de código que segue a palavra-chave `try` (não incluindo a própria palavra-chave `try`). Para simplificar, utilizamos o termo "bloco `try`" para nos referirmos ao bloco de código que se segue à palavra-chave `try`, bem como a palavra-chave `try`.] Todas instruções que leem os inteiros a partir do teclado (linhas 25 e 27) utilizam o método `nextInt` para ler um valor `int`. O método `nextInt` lança uma `InputMismatchException` se o valor lido não for um número inteiro.

A divisão que pode causar uma `ArithmeticException` não é realizada no bloco `try`. Em vez disso, a chamada para o método `quotient` (linha 29) invoca o código que tenta a divisão (linha 12); a JVM lança um objeto `ArithmeticException` quando o denominador for zero.

Observação de engenharia de software 11.1

As exceções emergem pelo código explicitamente mencionado em um bloco `try`, por chamadas para outros métodos, por chamadas de método profundamente aninhadas iniciadas pelo código em um bloco `try` ou a partir da Java Virtual Machine à medida que ela executa os bytecodes do Java.

Capturando exceções

O bloco `try` nesse exemplo é seguido por dois blocos `catch` — um que trata uma `InputMismatchException` (linhas 34–41) e outro que trata uma `ArithmeticException` (linhas 42–47). Um **bloco `catch`** (também chamado de **cláusula `catch`** ou **rotina de tratamento de exceção**) captura (isto é, recebe) e trata uma exceção. Um bloco `catch` inicia com a palavra-chave `catch` e é seguido por um parâmetro entre parênteses (chamado parâmetro de exceção, discutido em breve) e um bloco de código entre chaves. [*Nota:* o termo "cláusula `catch`" é às vezes utilizado para referir-se à palavra-chave `catch` seguida por um bloco de código, em que o termo "bloco `catch`"

refere-se apenas ao bloco de código que se segue à palavra-chave `catch`, mas que não a inclui. Para simplificar, utilizamos o termo "bloco `catch`" para nos referirmos ao bloco de código que se segue à palavra-chave `catch`, bem como à própria palavra-chave.]

Pelo menos um bloco `catch` ou um **bloco `finally`** (discutidos na Seção 11.7) deve se seguir imediatamente ao bloco `try`. Cada bloco `catch` especifica entre parênteses um **parâmetro de exceção** que identifica o tipo de exceção que o handler pode processar. Quando ocorrer uma exceção em um bloco `try`, o bloco `catch` que é executado é o primeiro cujo tipo corresponde ao tipo da exceção que ocorreu (isto é, o tipo no bloco `catch` corresponde exatamente ao tipo de exceção lançado ou é uma superclasse dele). O nome do parâmetro de exceção permite ao bloco `catch` interagir com um objeto de exceção capturado — por exemplo, invocar implicitamente o método `toString` da exceção capturada (como nas linhas 37 e 44), que exibe informações básicas sobre a exceção. Note que utilizamos o **objeto `System.err` (fluxo de erro padrão)** para imprimir mensagens de erro. Por padrão, os métodos print de `System.err`, como aqueles de `System.out`, exibem dados no prompt de comando.

A linha 38 do primeiro bloco `catch` chama o método `Scanner` `nextLine`. Como ocorreu uma `InputMismatchException`, a chamada ao método `nextInt` nunca é lida com sucesso nos dados do usuário — então lemos essa entrada com uma chamada ao método `nextLine`. Nesse ponto, não fazemos nada com a entrada, porque sabemos que ela é inválida. Todo bloco `catch` exibe uma mensagem de erro e pede ao usuário para tentar novamente. Depois que qualquer bloco `catch` termina, o usuário recebe um prompt solicitando entrada. Logo examinaremos mais profundamente como esse fluxo de controle funciona no tratamento de exceções.

Erro comum de programação 11.1

É um erro de sintaxe colocar código entre um bloco `try` e seus blocos `catch` correspondentes.

Erro comum de programação 11.2

Cada bloco `catch` pode ter apenas um único parâmetro — especificar uma lista de parâmetros de exceção separados por vírgulas é um erro de sintaxe.

Uma **exceção não capturada** é aquela para a qual não há nenhum bloco `catch` correspondente. Vimos exceções não capturadas na segunda e terceira saídas da Figura 11.1. Lembre-se de que quando as exceções ocorreram nesse exemplo, o aplicativo terminou prematuramente (depois de exibir o rastreamento da pilha de exceções). Isso nem sempre ocorre como um resultado de exceções não capturadas. Como você aprenderá no Capítulo 26, Multithreading, o Java utiliza um modelo de múltiplas threads de execução de programação. Cada **thread** é uma atividade paralela. Um programa pode ter muitas threads. Se um programa tiver apenas uma thread, uma exceção não capturada fará com que o programa seja encerrado. Se um programa tiver múltiplas threads, uma exceção não capturada encerrará apenas a thread em que ocorreu a exceção. Nesses programas, porém, certas threads podem contar com outras e se uma thread for encerrada por causa de uma exceção não capturada, pode haver efeitos adversos no restante do programa.

Modelo de terminação de tratamento de exceções

Se ocorrer uma exceção em um bloco `try` (como uma `InputMismatchException` sendo lançada como resultado do código na linha 25 da Figura 11.2), o bloco `try` termina imediatamente e o controle do programa é transferido para o primeiro dos blocos `catch` seguintes em que o tipo do parâmetro de exceção corresponde ao tipo da exceção lançada. Na Figura 11.2, o primeiro bloco `catch` captura `InputMismatchExceptions` (que ocorrem se uma entrada inválida for fornecida) e o segundo bloco `catch` captura `ArithmeticExceptions` (que ocorrem se houver uma tentativa de dividir por zero). Depois que a exceção é tratada, o controle do programa não retorna ao ponto de lançamento, porque o bloco `try` expirou (e suas variáveis locais foram perdidas). Em vez disso, o controle retoma depois do último bloco `catch`. Isso é conhecido como o **modelo de terminação de tratamento de exceções**. Algumas linguagens utilizam o **modelo de retomada de tratamento de exceções**, em que, após uma exceção ser tratada, o controle é retomado logo depois do ponto de lançamento.

Observe que nomeamos nossos parâmetros de exceção (`inputMismatchException` e `arithmeticException`) com base em seu tipo. Os programadores Java costumam simplesmente utilizar a letra `e` como o nome de parâmetros de exceção.

Boa prática de programação 11.1

Utilizar um nome de parâmetro de exceção que reflita o tipo do parâmetro promove a clareza lembrando-lhe do tipo de exceção em tratamento.

Depois de executar um bloco `catch`, o fluxo de controle desse programa prossegue para a primeira instrução depois do último bloco `catch` (linha 48 nesse caso). A condição na instrução `do...while` é `true` (a variável `continueLoop` contém valor inicial `true`), então o controle retorna ao começo do loop e uma entrada é mais uma vez solicitada ao usuário. Essa instrução de controle fará loop até que a entrada válida seja inserida. Nesse ponto, o controle de programa alcança a linha 32, que atribui `false` à variável `continueLoop`. O bloco `try` então termina. Se nenhuma exceção for lançada no bloco `try`, os blocos `catch` são pulados e o controle continua com a primeira instrução depois dos blocos `catch` (aprenderemos outra possibilidade ao discutirmos o bloco `finally` na Seção 11.7). Agora a condição para o loop `do...while` é `false` e o método `main` é encerrado.

O bloco `try` e seus blocos `catch` e/ou `finally` correspondentes formam uma **instrução `try`**. Não confunda os termos "bloco `try`" e a "instrução `try`" — a última inclui o bloco `try` bem como os blocos `catch` e/ou bloco `finally` seguintes.

Como com qualquer outro bloco do código, quando um bloco `try` termina, as variáveis locais declaradas no bloco saem de escopo e não mais são acessíveis; assim, as variáveis locais de um bloco `try` não são acessíveis nos blocos `catch` correspondentes. Quando um bloco `catch` termina, as variáveis locais declaradas dentro do bloco `catch` (incluindo o parâmetro de exceção desse bloco `catch`) também saem de escopo e são destruídas. Quaisquer blocos `catch` restantes na instrução `try` são ignorados e a execução é retomada na primeira linha de código depois da sequência `try...catch` — essa será um bloco `finally`, se houver um presente.

Utilizando a cláusula `throws`

Agora vamos examinar o método `quotient` (Figura 11.2, linhas 9–13). A parte da declaração de método localizada na linha 10 é conhecida como uma **cláusula `throws`**. Uma cláusula `throws` especifica as exceções que o método lança. Essa cláusula aparece depois da lista de parâmetros do método e antes do corpo do método. Ela contém uma lista das exceções separadas por vírgulas que o método lançará se vários problemas ocorrerem. Essas exceções podem ser lançadas por instruções no corpo do método ou por métodos chamados a partir do corpo. Um método pode lançar exceções das classes listadas em sua cláusula `throws` ou de suas subclasses. Adicionamos a cláusula `throws` a esse aplicativo para indicar ao resto do programa que esse método pode lançar uma `ArithmeticException`. Os clientes do método `quotient` são assim informados de que o método pode lançar uma `ArithmeticException`. Você aprenderá mais sobre a cláusula `throws` na Seção 11.6.

Dica de prevenção de erro 11.2

Leia a documentação online da API para obter informações sobre um método antes de utilizar esse método em um programa. A documentação especifica as exceções lançadas pelo método (se houver alguma) e indica as razões pelas quais tais exceções podem ocorrer. Em seguida, leia a documentação online da API para as classes de exceção especificadas. A documentação para uma classe de exceção costuma conter possíveis razões por que essas exceções ocorrem. Por fim, forneça o tratamento para essas exceções em seu programa.

Quando a linha 12 executar, se o `denominator` for zero, a JVM lança um objeto `ArithmeticException`. Esse objeto será capturado pelo bloco `catch` nas linhas 42–47, que exibe informações básicas sobre a exceção invocando implicitamente o método `toString` da exceção e depois pede ao usuário para tentar novamente.

Se o `denominator` não for zero, o método `quotient` realiza a divisão e retorna o resultado ao ponto de invocação do método `quotient` no bloco `try` (linha 29). As linhas 30–31 exibem o resultado do cálculo e a linha 32 configura `continueLoop` como `false`. Nesse caso, o bloco `try` completa com sucesso, então o programa pula os blocos `catch` e falha na condição da linha 48, e o método `main` completa a execução normalmente.

Observe que quando `quotient` lança uma `ArithmeticException`, `quotient` termina e não retorna um valor, e as variáveis locais de `quotient` saem de escopo (e são destruídas). Se `quotient` contivesse variáveis locais que fossem referências a objetos e não houvesse nenhuma outra referência a esses objetos, os objetos seriam marcados para a coleta de lixo. Além disso, quando ocorre uma exceção, o bloco `try` a partir do qual `quotient` foi chamado termina antes que as linhas 30–32 possam executar. Aqui, também, se variáveis locais fossem criadas no bloco `try` antes de a exceção ser lançada, essas variáveis sairiam de escopo.

Se uma `InputMismatchException` é gerada pelas linhas 25 ou 27, o bloco `try` termina e a execução continua com o bloco `catch` nas linhas 34–41. Nesse caso, o método `quotient` não é chamado. Então o método `main` continua depois do último bloco `catch` (linha 48).

11.5 Quando usar tratamento de exceções

O tratamento de exceções é projetado para processar **erros síncronos**, que ocorrem quando uma instrução executa. Exemplos comuns que veremos por todo o livro são índices fora do intervalo de array, estouro aritmético (isto é, um valor fora do intervalo representável de valores), divisão por zero, parâmetros inválidos de método, interrupção de thread (como veremos no Capítulo 26) e alocação malsucedida de memória (devido à falta de memória). O tratamento de exceções não é projetado para processar problemas associados com os **eventos assíncronos** (por exemplo, conclusões de E/S de disco, chegadas de mensagem de rede, cliques de mouse e pressionamentos de tecla), que ocorrem paralelamente com o fluxo de controle do programa e independentemente dele.

Observação de engenharia de software 11.2

Incorpore sua estratégia de tratamento de exceções ao sistema desde o princípio do processo de design. Pode ser difícil incluir um tratamento de exceções depois que um sistema foi implementado.

Observação de engenharia de software 11.3

O tratamento de exceções fornece uma técnica única e uniforme para processamento de problemas. Isso ajuda os programadores que trabalham em grandes projetos a entender o código de processamento de erro uns dos outros.

Observação de engenharia de software 11.4

O tratamento de exceções simplifica a combinação de componentes de software e permite trabalhar em conjunto eficientemente possibilitando que os componentes predefinidos comuniquem problemas para componentes específicos ao aplicativo, que então podem processar os problemas de maneira específica ao aplicativo.

11.6 Hierarquia de exceções no Java

Todas as classes de exceção do Java herdam direta ou indiretamente da classe **`Exception`**, formando uma hierarquia de herança. Você pode estender essa hierarquia com suas próprias classes de exceção.

A Figura 11.3 mostra uma pequena parte da hierarquia de herança da classe **`Throwable`** (uma subclasse de `Object`), que é a superclasse da classe `Exception`. Só Objetos `Throwable` podem ser usados com o mecanismo de tratamento de exceções. A classe `Throwable` tem duas subclasses: `Exception` e `Error`. A classe `Exception` e suas subclasses — por exemplo, `RuntimeException` (pacote `java.lang`) e `IOException` (pacote `java.io`) — representam situações excepcionais que podem ocorrer em um programa Java e que podem ser capturadas pelo aplicativo. A classe **`Error`** e suas subclasses representam situações anormais que acontecem na JVM. `Errors` não acontecem frequentemente e não devem ser capturados pelos aplicativos — normalmente não é possível que aplicativos se recuperem de `Errors`.

A hierarquia de exceções no Java contém centenas de classes. As informações sobre classes de exceção do Java podem ser localizadas por toda a Java API. Você pode examinar a documentação de `Throwable` em `java.sun.com/javase/6/docs/api/java/lang/Throwable.html`. A partir daí, você pode examinar as subclasses dessa classe para obter informações adicionais sobre `Exceptions` e `Errors` do Java.

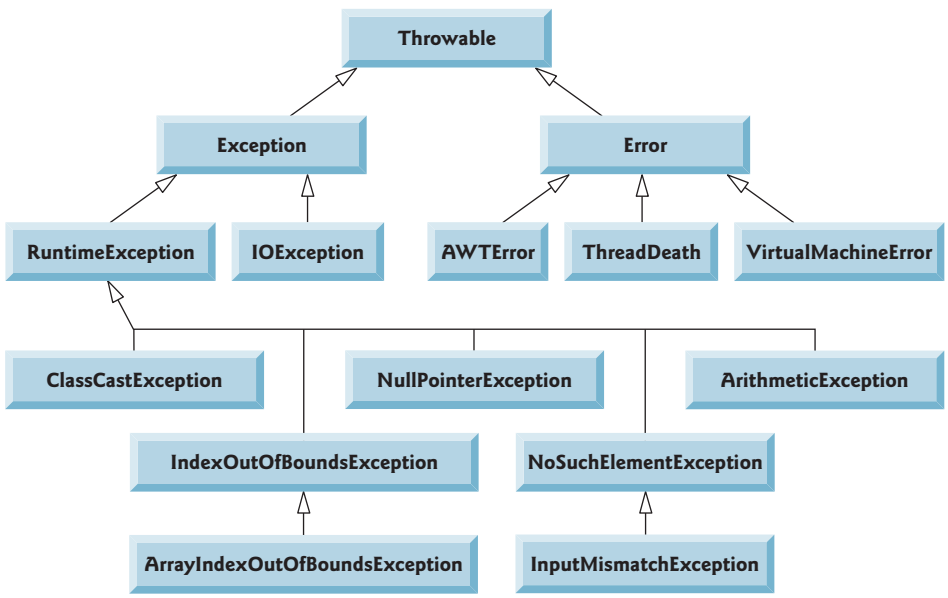


Figura 11.3 | Parte da hierarquia de herança da classe `Throwable`.

Exceções verificadas versus não verificadas

O Java distingue entre **exceções verificadas** e **exceções não verificadas**. Essa distinção é importante, porque o compilador Java impõe um **requisito "*capture ou declare*"** (*catch-or-declare*) às exceções verificadas. O tipo de uma exceção determina se a exceção é verificada ou é não verificada. Todos os tipos de exceção que são subclasses diretas ou indiretas da classe **`RuntimeException`** (pacote `java.lang`) são exceções não verificadas. Estas costumam ser causadas por deficiências no código do seu programa. Exemplos de exceções não verificadas incluem `ArrayIndexOutOfBoundsExceptions` (discutidas no Capítulo 7) e `ArithmeticExceptions` (mostradas na Figura 11.3). Todas as classes que herdam da classe `Exception` mas não da classe `RuntimeException` são consideradas exceções verificadas. Essas exceções são tipicamente causadas por condições que não estão no controle do programa — por exemplo, no processamento de arquivos, o programa não pode abrir um arquivo porque o arquivo não existe. As classes que herdam da classe `Error` são consideradas não verificadas.

O compilador *verifica* cada chamada de método e declaração de método para determinar se o método lança exceções verificadas. Se lançar, o compilador verifica se a exceção verificada é capturada ou declarada em uma cláusula `throws`. Mostramos como capturar e declarar exceções verificadas nos vários exemplos a seguir. A partir da discussão da Seção 11.4, lembre-se de que a cláusula `throws` especifica as

exceções que um método lança. Tais exceções não são capturadas no corpo do método. Para satisfazer a parte *capture* do requisito *capture ou declare*, o código que gera a exceção deve ser empacotado em um bloco `try` e fornecer um handler `catch` para o tipo de exceção verificada (ou um de seus tipos de superclasse). Para satisfazer a parte *declare* do requisito *capture ou declare*, o método contendo o código que gera a exceção deve fornecer uma cláusula `throws` contendo o tipo de exceção verificada depois de sua lista de parâmetros e antes do corpo do método. Se o requisito capture ou declare não for satisfeito, o compilador emitirá uma mensagem de erro indicando que a exceção deve ser capturada ou declarada. Isso força-lhe a pensar nos problemas que podem ocorrer quando um método que lança exceções verificadas for chamado.

Observação de engenharia de software 11.5

Você precisa lidar com exceções verificadas. Isso resulta em código mais robusto do que aquele que seria criado se você fosse capaz de simplesmente ignorar as exceções.

Erro comum de programação 11.3

Um erro de compilação ocorre se um método tentar explicitamente lançar uma exceção verificada (ou chamar outro método que lança uma exceção verificada) e essa exceção não estiver listada na cláusula `throws` do método.

Erro comum de programação 11.4

Se um método de subclasse sobrescreve um método de superclasse, é um erro o método de subclasse listar mais exceções em sua cláusula `throws` do que o método sobrescrito da superclasse. Mas a cláusula `throws` de uma subclasse pode conter um subconjunto da lista de `throws` de uma superclasse.

Observação de engenharia de software 11.6

Se o método chamar outros métodos que lançam explicitamente exceções verificadas, essas exceções devem ser capturadas ou declaradas no método. Se uma exceção pode ser significativamente tratada em um método, o método deve capturar a exceção em vez de declará-la.

Ao contrário das exceções verificadas, o compilador Java não verifica o código para determinar se uma exceção não verificada é capturada ou declarada. Em geral, pode-se impedir a ocorrência de exceções não verificadas pela codificação adequada. Por exemplo, a `ArithmeticException` não verificada lançada pelo método `quotient` (linhas 9–13) na Figura 11.2 pode ser evitada se o método assegurar que o denominador não é zero antes de tentar realizar a divisão. Não é necessário que as exceções não verificadas sejam listadas na cláusula `throws` de um método — mesmo se forem, essas exceções não precisam ser capturadas por um aplicativo.

Observação de engenharia de software 11.7

Embora o compilador não imponha o requisito capture ou declare para as exceções não verificadas, ele fornece o código de tratamento de exceções adequado quando se sabe que tais exceções são possíveis. Por exemplo, um programa deve processar a `NumberFormatException` do método `Integer parseInt`, mesmo que `NumberFormatException` (uma subclasse de `RuntimeException`) seja um tipo de exceção não verificada. Isso torna os programas mais robustos.

Capturando exceções de subclasse

Se um handler `catch` for escrito para capturar objetos de exceção de um tipo de superclasse, ele também pode capturar todos os objetos de subclasses dessa classe. Isso permite a `catch` tratar erros relacionados com uma notação concisa e permite o processamento polimorfo das exceções relacionadas. Você certamente pode capturar cada tipo de subclasse individualmente se essas exceções exigirem processamento diferente. Capturar exceções relacionadas em um bloco `catch` só faz sentido se o comportamento do tratamento for o mesmo para todas as subclasses.

Somente o primeiro `catch` correspondente executa

Se houver múltiplos blocos `catch` que correspondem a um tipo particular de exceção, somente o primeiro bloco `catch` correspondente executará na ocorrência de uma exceção desse tipo. É um erro de compilação capturar o *mesmo tipo exato* em dois blocos `catch` diferentes associados com um bloco `try` específico. Entretanto, pode haver vários blocos `catch` que correspondam a uma exceção — isto é, vários blocos `catch` cujos tipos forem os mesmos que o tipo de exceção ou uma superclasse desse tipo. Por exemplo, poderíamos seguir um bloco `catch` para o tipo `ArithmeticException` com um bloco `catch` para o tipo `Exception` — ambos corresponderiam às `ArithmeticExceptions`, mas somente o primeiro bloco `catch` correspondente executaria.

Dica de prevenção de erro 11.3

A captura de tipos de subclasse individualmente está sujeita a erro se você se esquecer de testar um ou mais dos tipos de subclasse explicitamente; capturar a superclasse garante que os objetos de todas as subclasses serão capturados. Posicionar um bloco `catch` para o tipo de superclasse depois de todos os outros blocos `catch` de subclasse para subclasses dessa superclasse assegura que todas as exceções de subclasse sejam por fim capturadas.

Erro comum de programação 11.5

Colocar um bloco `catch` para um tipo de exceção de superclasse antes de outros blocos `catch` que capturam tipos de exceção de subclasse impediria que esses blocos executem, então ocorre um erro de compilação.

11.7 Bloco finally

Os programas que obtêm certos tipos de recursos devem retorná-los ao sistema explicitamente para evitar os supostos **vazamentos de recurso**. Em linguagens de programação como C e C++, o tipo mais comum de vazamento de recurso é um vazamento de memória. O Java realiza coleta automática de lixo de memória não mais utilizada por programas, evitando assim a maioria dos vazamentos de memória. Entretanto, outros tipos de vazamentos de recurso podem ocorrer. Por exemplo, arquivos, conexões de banco de dados e conexões de rede que não são fechadas adequadamente depois que não são mais necessárias talvez não estejam disponíveis para uso em outros programas.

Dica de prevenção de erro 11.4

Uma questão sutil é que o Java não elimina inteiramente os vazamentos de memória. O Java não efetuará coleta de lixo de um objeto até que não haja nenhuma referência a ele. Portanto, se os programadores mantiverem erroneamente referências a objetos indesejáveis, vazamentos de memória podem ocorrer. Para ajudar a evitar esse problema, configure variáveis de tipo por referência como `null`, quando elas não são mais necessárias.

O bloco `finally` (que consiste na palavra-chave `finally`, seguida pelo código entre chaves), às vezes referido como a **cláusula `finally`**, é opcional. Se estiver presente, ele é colocado depois do último bloco `catch`, como na Figura 11.4.

```
try
{
   instruções
   instruções de aquisição de recursos
} // fim do try
catch ( UmTipoDeExceção exceção1 )
{
   instruções de tratamento de exceções
} // fim do catch
...
catch ( OutroDeExceção exceção2 )
{
   instruções de tratamento de exceções
} // fim do catch
finally
{
   instruções
   instruções de liberação de recursos
} // fim de finally
```

Figura 11.4 | Uma instrução `try` com um bloco `finally`.

O Java garante que o bloco `finally` executará se uma exceção for lançada no bloco `try` correspondente. O Java também garante que o bloco `finally` executará se um bloco `try` terminar utilizando uma instrução `return`, `break` ou `continue` ou simplesmente alcançando sua chave direita de fechamento. O bloco `finally` *não* executará se o aplicativo fechar antes de um bloco `try` chamando o método **`System.exit`**. Esse método, que demonstramos no Capítulo 17, encerra imediatamente um aplicativo.

Como um bloco `finally` quase sempre executa, em geral ele contém código de liberação de recursos. Suponha que um recurso é alocado em um bloco `try`. Se nenhuma exceção ocorrer, os blocos `catch` são pulados e o controle passa para o bloco `finally`, que libera o recurso. O controle então prossegue à primeira instrução depois do bloco `finally`. Se uma exceção ocorrer no bloco `try`, o bloco `try` termina. Se o programa capturar a exceção em um dos blocos `catch`, ele processa a exceção, depois o bloco `finally` libera o recurso e o controle

prossegue para a primeira instrução depois do bloco `finally`. Se o programa não capturar a exceção, o bloco `finally` ainda irá liberar o recurso e fará uma tentativa de capturar a exceção em um método chamador.

Dica de prevenção de erro 11.5

O bloco `finally` é um lugar ideal para liberar recursos adquiridos em um bloco `try` (como arquivos abertos), o que ajuda a eliminar vazamentos de recurso.

Dica de desempenho 11.2

Sempre libere um recurso explicitamente e logo que ele não for mais necessário. Isso torna os recursos disponíveis para reutilização, aprimorando assim a utilização deles.

Se uma exceção que ocorre em um bloco `try` não puder ser capturada por handlers `catch` desse bloco `try`, o programa pula o restante do bloco `try` e o controle prossegue para o bloco `finally`. Então o programa passa a exceção para o próximo bloco `try` externo — normalmente no método de chamada — onde um bloco `try` associado pode capturá-lo. Esse processo pode ocorrer pelos muitos níveis de blocos `try`. Além disso, a exceção talvez não seja capturada.

Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará. Então a exceção é passada para o próximo bloco `try` externo — novamente, em geral no método chamador.

A Figura 11.5 demonstra que o bloco `finally` executa mesmo se uma exceção não for lançada no bloco `try` correspondente. O programa contém métodos `static main` (linhas 6–18), `throwException` (linhas 21–44) e `doesNotThrowException` (linhas 47–64). Os métodos `throwException` e `doesNotThrowException` são declarados `static`, portanto `main` pode chamá-los diretamente sem instanciar um objeto `UsingExceptions`.

```
1  // Figura 11.5: UsingExceptions.java
2  // mecanismo de tratamento de exceções try...catch...finally.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          throwException(); // chama método throwException
11       } // fim do try
12       catch ( Exception exception ) // exceção lançada por throwException
13       {
14          System.err.println( "Exception handled in main" );
15       } // fim do catch
16
17       doesNotThrowException();
18    } // fim de main
19
20    // demonstra try...catch...finally
21    public static void throwException() throws Exception
22    {
23       try // lança uma exceção e imediatamente a captura
24       {
25          System.out.println( "Method throwException" );
26          throw new Exception(); // gera a exceção
27       } // fim do try
28       catch ( Exception exception ) // captura exceção lançada em try
29       {
30          System.err.println(
31             "Exception handled in method throwException" );
32          throw exception; // lança novamente para processamento adicional
33
34          // o código aqui não seria alcançado; causaria erros de compilação
35
36       } // fim do catch
37       finally // executa independentemente do que ocorre em try...catch
38       {
39          System.err.println( "Finally executed in throwException" );
40       } // fim de finally
41
```

```
42          // o código aqui não seria alcançado; causaria erros de compilação
43
44       } // fim do método throwException
45
46       // demonstra finally quando nenhuma exceção ocorrer
47       public static void doesNotThrowException()
48       {
49          try // bloco try não lança uma exceção
50          {
51             System.out.println( "Method doesNotThrowException" );
52          } // fim do try
53          catch ( Exception exception ) // não executa
54          {
55             System.err.println( exception );
56          } // fim do catch
57          finally // executa independentemente do que ocorre em try...catch
58          {
59             System.err.println(
60                "Finally executed in doesNotThrowException" );
61          } // fim de finally
62
63          System.out.println( "End of method doesNotThrowException" );
64       } // fim do método doesNotThrowException
65    } // fim da classe UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Figura 11.5 | Mecanismo de tratamento de exceções `try`...`catch`...`finally`.

Tanto `System.out` como `System.err` são **fluxos** — uma sequência de bytes. Enquanto `System.out` (conhecido como o **fluxo de saída padrão**) é utilizado para exibir uma saída do programa, `System.err` (conhecido como o **fluxo de erro padrão**) é utilizado para exibir erros de um programa. A saída desses fluxos pode ser redirecionada (isto é, enviada para algum outro lugar diferente do prompt de comando, como um arquivo, por exemplo). Utilizar dois fluxos diferentes permite-lhe separar facilmente mensagens de erro de outra saída. Por exemplo, a saída de dados de `System.err` poderia ser enviada para um arquivo de log, enquanto a saída de dados de `System.out` pode ser exibida na tela. Para simplificar, este capítulo não redirecionará a saída de `System.err`, mas exibirá essas mensagens no prompt de comando. Você aprenderá mais sobre fluxos no Capítulo 17, Arquivos, fluxos e serialização de objetos.

Lançando exceções com a instrução `throw`

O método `main` (Figura 11.5) começa a executar, entra em seu bloco `try` e imediatamente chama o método `throwException` (linha 10). O método `throwException` lança uma `Exception`. A instrução na linha 26 é conhecida como uma **instrução `throw`** — ela é executada para indicar que uma exceção ocorreu. Até agora, você só capturou exceções lançadas por métodos chamados. Você pode lançar exceções utilizando a instrução `throw`. Assim como com as exceções lançadas pelos métodos da Java API, isso indica para os aplicativos clientes que ocorreu um erro. Uma instrução `throw` especifica um objeto a ser lançado. O operando de um `throw` pode ser de qualquer classe derivada da classe `Throwable`.

Observação de engenharia de software 11.8

Quando `toString` é invocada em qualquer objeto `Throwable`, sua `string` resultante inclui a `string` descritiva que foi fornecida para o construtor, ou simplesmente o nome de classe se nenhuma `string` foi fornecida.

Observação de engenharia de software 11.9

Um objeto pode ser lançado sem conter informações sobre o problema que ocorreu. Nesse caso, simplesmente saber que uma exceção de um tipo particular ocorreu pode fornecer informações suficientes para o handler processar o problema corretamente.

Observação de engenharia de software 11.10

As exceções podem ser lançadas a partir de construtores. Quando um erro é detectado em um construtor, uma exceção deve ser lançada para evitar-se a criação de um objeto formado inadequadamente.

Relançando exceções

A linha 32 da Figura 11.5 **relança a exceção**. As exceções são relançadas quando um bloco `catch`, ao receber uma exceção, decide que não pode processar essa exceção ou que só pode processá-la parcialmente. Relançar uma exceção adia o tratamento de exceções (ou talvez uma parte dele) para outro bloco `catch` associado com uma instrução `try` externa. Uma exceção é relançada utilizando-se a **palavra-chave `throw`**, seguida por uma referência ao objeto de exceção que acabou de ser capturado. Observe que as exceções não podem ser relançadas a partir de um bloco `finally`, uma vez que o parâmetro de exceção (uma variável local) do bloco `catch` não mais existe.

Quando ocorre um relançamento, o próximo bloco `try` circundante detecta a exceção relançada e os blocos `catch` desse bloco `try` tentam tratá-la. Nesse caso, o próximo bloco `try` circundante é localizado nas linhas 8–11 do método `main`. Antes, porém, de a exceção relançada ser tratada, o bloco `finally` (linhas 37–40) executa. Então, o método `main` detecta a exceção relançada no bloco `try` e a trata no bloco `catch` (linhas 12–15).

Em seguida, `main` chama o método `doesNotThrowException` (linha 17). Nenhuma exceção é lançada no bloco `try` de `doesNotThrowException` (linhas 49–52), então o programa pula o bloco `catch` (linhas 53–56), mas, apesar disso, o bloco `finally` (linhas 57–61) executa. O controle prossegue para a instrução depois do bloco `finally` (linha 63). Então o controle retorna a `main` e o programa é encerrado.

Erro comum de programação 11.6

Se uma exceção não tiver sido capturada quando o controle entrar em um bloco `finally` e esse bloco lançar uma exceção que não será capturada por ele, a primeira exceção será perdida e a exceção do bloco será retornada ao método chamador.

Dica de prevenção de erro 11.6

Evite colocar código que possa lançar (`throw`) uma exceção em um bloco `finally`. Se esse código for necessário, inclua o código em um `try...catch` dentro do bloco `finally`.

Erro comum de programação 11.7

Assumir que uma exceção lançada de um bloco `catch` será processada por esse bloco `catch` ou qualquer outro bloco `catch` associado com a mesma instrução `try` pode resultar em erros de lógica.

Boa prática de programação 11.2

O mecanismo de tratamento de exceções do Java é projetado para remover código de processamento de erro da linha principal do código de um programa para aprimorar a clareza de programa. Não coloque `try...catch... finally` em torno de toda instrução que possa lançar uma exceção. Isso torna os programas difíceis de ler. Em vez disso, coloque um bloco `try` em torno de uma parte significativa do código, esse bloco `try` deve ser seguido por blocos `catch` que tratam cada possível exceção e os blocos `catch` devem ser seguidos por um único bloco `finally` (se algum for necessário).

11.8 Desempilhamento de pilha

Quando uma exceção é lançada mas não capturada em um escopo em particular, a pilha de chamada de método é "desempilhada" e é feita uma tentativa de capturar (`catch`) a exceção no próximo bloco `try` externo. Esse processo é chamado **desempilhamento**. Desempilhar a pilha de chamada de método significa que o método em que a exceção não foi capturada é encerrado, todas as variáveis locais nesse método saem de escopo e o controle retorna à instrução que originalmente invocou esse método. Se um bloco `try` incluir essa instrução, uma tentativa de capturar a exceção com `catch` é feita. Se um bloco `try` não confinar essa instrução ou se a exceção não for capturada, ocorre novamente uma liberação de pilha. A Figura 11.6 demonstra a liberação de pilha.

```
1  // Figura 11.6: UsingExceptions.java
2  //Desempilhamento.
3
4  public class UsingExceptions
5  {
```

```
6       public static void main( String[] args )
7       {
8          try // chama throwException para demonstrar o desempilhamento
9          {
10            throwException();
11         } // fim do try
12         catch ( Exception exception ) // exceção lançada em throwException
13         {
14            System.err.println( "Exception handled in main" );
15         } // fim do catch
16      } // fim de main
17
18      // throwException lança exceção que não é capturada nesse método
19      public static void throwException()throws Exception
20      {
21         try // lança uma exceção e a captura em main
22         {
23            System.out.println( "Method throwException" );
24            throw new Exception(); // gera a exceção
25         } // fim do try
26         catch ( RuntimeException runtimeException ) // captura o tipo correto
27         {
28            System.err.println(
29               "Exception handled in method throwException" );
30         } // fim do catch
31         finally // o bloco finally sempre executa
32         {
33            System.err.println( "Finally is always executed" );
34         } // fim de finally
35      } // fim do método throwException
36   } // fim da classe UsingExceptions
```

```
Method throwException
Finally is always executed
Exception handled in main
```

Figura 11.6 | Desempilhamento.

Quando o método `main` executa, a linha 10 no bloco `try` chama o método `throwException` (linhas 19–35). No bloco `try` do método `throwException` (linhas 21–25), a linha 24 lança uma `Exception`. Isso termina o bloco `try` imediatamente e o controle pula o bloco `catch` na linha 26, porque o tipo que está sendo capturado (`RuntimeException`) não é uma correspondência exata com o tipo lançado (`Exception`) e não é uma superclasse dele. O método `throwException` termina (mas não até seu bloco `finally` executar) e o controle retorna à linha 10 — o ponto a partir do qual ele foi chamado no programa. A linha 10 está em um bloco `try` circundante. A exceção ainda não foi tratada, então o bloco `try` termina e é feita uma tentativa de capturar a exceção na linha 12. O tipo que está sendo capturado (`Exception`) corresponde ao tipo lançado. Consequentemente, o bloco `catch` processa a exceção e o programa termina no fim de `main`. Se não houvesse nenhum bloco `catch` correspondente, e a exceção não fosse declarada em cada método que a lança, ocorreria um erro de compilação. Lembre-se de que esse nem sempre é o caso — para exceções não verificadas, o aplicativo compilará, mas executará com resultados inesperados.

11.9 printStackTrace, getStackTrace e getMessage

A partir da discussão da Seção 11.6, lembre-se de que as exceções derivam da classe `Throwable`. A classe `Throwable` oferece um método **`printStackTrace`** que envia para o fluxo de erro padrão o rastreamento da pilha (discutido na Seção 11.3). Frequentemente, isso é útil no processo de teste e depuração. A classe `Throwable` também fornece um método **`getStackTrace`** que recupera as informações sobre o rastreamento de pilha que podem ser impressas por `printStackTrace`. O método **`getMessage`** da classe `Throwable` retorna a string descritiva armazenada em uma exceção. O exemplo nesta seção demonstra esses três métodos.

Dica de prevenção de erro 11.7

Uma exceção que não é capturada em um aplicativo faz com que o handler de exceção padrão do Java execute. Isso exibe o nome da exceção, uma mensagem descritiva que indica o problema que ocorreu e um completo rastreamento da pilha de execução. Em um aplicativo com uma única thread de execução, o aplicativo termina. Em um aplicativo com múltiplas threads, a thread que causou a exceção termina.

Dica de prevenção de erro 11.8

O método `Throwable` `toString` (herdado por todas as subclasses `Throwable`) retorna uma string contendo o nome da classe da exceção e uma mensagem descritiva.

A Figura 11.7 demonstra `getMessage`, `printStackTrace` e `getStackTrace`. Se quiséssemos imprimir as informações de rastreamento de pilha para outros fluxos que não o fluxo de erro padrão, poderíamos utilizar as informações retornadas de `getStackTrace` e imprimi-las para outro fluxo ou utilizar uma das versões sobrecarregadas do método `printStackTrace`. O envio de dados a outros fluxos é discutido no Capítulo 17, Arquivos, fluxos e serialização de objetos.

```
1  // Figura 11.7: UsingExceptions.java
2  // Métodos Throwable getMessage, getStackTrace e printStackTrace.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // chama method1
11       } // fim do try
12       catch ( Exception exception ) // captura a exceção lançada em method1
13       {
14          System.err.printf( "%s\n\n", exception.getMessage() );
15          exception.printStackTrace(); // imprime rastreamento da pilha de exceções
16
17          // obtém informações de rastreamento da pilha
18          StackTraceElement[] traceElements = exception.getStackTrace();
19
20          System.out.println( "\nStack trace from getStackTrace:" );
21          System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
22
23          // faz um loop por traceElements para obter a descrição da exceção
24          for ( StackTraceElement element : traceElements )
25          {
26             System.out.printf( "%s\t", element.getClassName() );
27             System.out.printf( "%s\t", element.getFileName() );
28             System.out.printf( "%s\t", element.getLineNumber() );
29             System.out.printf( "%s\n", element.getMethodName() );
30          } // for final
31       } // fim do catch
32    } // fim de main
33
34    // chama method2; lança exceções de volta para main
35    public static void method1()throws Exception
36    {
37       method2();
38    } // fim do método method1
39
40    // chama method3; lança exceções de volta para method1
41    public static void method2()throws Exception
42    {
43       method3();
44    } // fim do método method2
45
46    // lança Exception de volta para method2
47    public static void method3()throws Exception
```

```
48      {
49         throw new Exception( "Exception thrown in method3" );
50      } // fim do método method3
51   } // fim da classe UsingExceptions
```

```
Exception thrown in method3
java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:49)
        at UsingExceptions.method2(UsingExceptions.java:43)
        at UsingExceptions.method1(UsingExceptions.java:37)
        at UsingExceptions.main(UsingExceptions.java:10)
Stack trace from getStackTrace:
Class           File                    Line    Method
UsingExceptions UsingExceptions.java    49      method3
UsingExceptions UsingExceptions.java    43      method2
UsingExceptions UsingExceptions.java    37      method1
UsingExceptions UsingExceptions.java    10      main
```

Figura 11.7 | Métodos `Throwable` `getMessage`, `getStackTrace` e `printStackTrace`.

Em `main`, o bloco `try` (linhas 8–11) chama `method1` (declarado nas linhas 35–38), que por sua vez chama `method2` (declarado nas linhas 41–44), que por sua vez chama `method3` (declarado nas linhas 47–50). A linha 49 de `method3` lança um objeto `Exception` — esse é o ponto de lançamento. Como a instrução `throw` na linha 49 não está incluída em um bloco `try`, o desempilhamento ocorre — `method3` termina na linha 49, então o controle retorna à instrução em `method2` que invocou `method3` (isto é, linha 43). Como nenhum bloco `try` inclui a linha 43, o desempilhamento ocorre novamente — `method2` termina na linha 43 e o controle retorna à instrução em `method1` que invocou `method2` (isto é, linha 37). Como nenhum bloco `try` inclui a linha 37, o desempilhamento ocorre mais uma vez — `method1` termina na linha 37 e o controle retorna à instrução em `main` que invocou `method1` (isto é, linha 10). O bloco `try` nas linhas 8–11 inclui essa instrução. A exceção não foi tratada, então o bloco `try` termina e o primeiro bloco `catch` correspondente (linhas 12–31) captura e processa a exceção.

A linha 14 invoca o método `getMessage` da exceção para obter a descrição de exceção. A linha 15 invoca o método `printStackTrace` da exceção para gerar saída do rastreamento de pilha que indica onde ocorreu a exceção. A linha 18 invoca o método `getStackTrace` da exceção para obter as informações de rastreamento de pilha como um array de objetos **`StackTraceElement`**. As linhas 24–30 obtêm cada `StackTraceElement` no array e invocam seus métodos **`getClassName`**, **`getFileName`**, **`getLineNumber`** e **`getMethodName`** para obter o nome da classe, o nome do arquivo, o número da linha e o nome de método, respectivamente, para esse `StackTraceElement`. Todo `StackTraceElement` representa uma chamada de método na pilha de chamadas de método.

A saída na Figura 11.7 mostra que as informações de rastreamento de pilha impressas por `printStackTrace` seguem o padrão: *nomeDaClasse.nomeDoMétodo* (*nomeDoArquivo:númeroDaLinha*), em que *nomeDaClasse*, *nomeDoMétodo* e *nomeDoArquivo* indicam o nome da classe, do método e do arquivo em que a exceção ocorreu, respectivamente, e *númeroDaLinha* indica onde no arquivo ocorreu a exceção. Vimos isso na saída da Figura 11.1. O método `getStackTrace` permite o processamento personalizado das informações de exceção. Compare a saída de `printStackTrace` com a saída criada a partir de `StackTraceElements` para ver que ambas contêm as mesmas informações de rastreamento de pilha.

Observação de engenharia de software 11.11

Nunca ignore uma exceção que você captura. Pelo menos utilize `printStackTrace` para gerar saída de uma mensagem de erro. Isso informará os usuários de que existe um problema, assim eles podem adotar as ações apropriadas.

11.10 Exceções encadeadas

Às vezes um método responde a uma exceção lançando um tipo de exceção diferente que é específico do aplicativo em uso. Se um bloco `catch` lançar uma nova exceção, as informações da exceção original e do rastreamento de pilha serão perdidas. As primeiras versões do Java não forneciam nenhum mecanismo para empacotar as informações da exceção original com as informações da nova exceção a fim de fornecer um rastreamento de pilha completo que mostre onde o problema original ocorreu. Isso torna a depuração desses problemas particularmente difícil. **Exceções encadeadas** permitem a um objeto exceção manter informações de rastreamento de pilha completas a partir da exceção original. A Figura 11.8 demonstra exceções encadeadas.

```
1  // Figura 11.8: UsingChainedExceptions.java
2  // Exceções encadeadas.
3
4  public class UsingChainedExceptions
```

```
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // chama method1
11       } // fim do try
12       catch ( Exception exception ) // exceções lançadas de method1
13       {
14          exception.printStackTrace();
15       } // fim do catch
16    } // fim de main
17
18    // chama method2 ; lança exceções para main
19    public static void method1()throws ExceptionW
20    {
21       try
22       {
23          method2(); // chama method2
24       } // fim do try
25       catch ( Exception exception ) // exceção lançada de method2
26       {
27        throw new Exception( "Exception thrown in method1", exception );
28       } // fim do catch
29    } // fim do método method1
30
31    // chama method3; lança exceções de volta para method1
32    public static void method2()throws Exception
33    {
34       try
35       {
36          method3(); // chama method3
37       } // fim do try
38       catch ( Exception exception ) // exceção lançada de method3
39       {
40        throw new Exception( "Exception thrown in method2", exception );
41       } // fim do catch
42    } // fim do método method2
43
44    // lança Exception de volta para method2
45    public static void method3() throws Exception
46    {
47       throw new Exception( "Exception thrown in method3" );
48    } // fim do método method3
49 } // fim da classe UsingChainedExceptions
```

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:27)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:10)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:40)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:23)
       ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:47)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:36)
       ... 2 more
```

Figura 11.8 | Exceções encadeadas.

O programa consiste em quatro métodos — `main` (linhas 6–16), `method1` (linhas 19–29), `method2` (linhas 32–42) e `method3` (linhas 45–48). A linha 10 no bloco `try` do método `main` chama `method1`. A linha 23 no bloco `try` do `method1` chama `method2`. A linha 36 no bloco `try` do `method2` chama `method3`. Em `method3`, a linha 47 lança uma nova `Exception`. Como essa instrução não está em um bloco `try`, `method3` termina e a exceção é retornada ao método chamador (`method2`) na linha 36. Essa instrução *está* em um bloco `try`; portanto, o bloco `try` termina e a exceção é capturada nas linhas 38–41. A linha 40 no bloco `catch` lança uma nova exceção. Nesse caso, o

construtor `Exception` com dois argumentos é chamado. O segundo argumento representa a exceção que era a causa original do problema. Nesse programa, essa exceção ocorreu na linha 47. Como uma exceção é lançada a partir do bloco `catch`, `method2` termina e retorna a nova exceção ao método chamador (`method1`) na linha 23. Mais uma vez, essa instrução está em um bloco `try`, então o bloco `try` termina e a exceção é capturada nas linhas 25–28. A linha 27 no bloco `catch` lança uma nova exceção e utiliza a exceção que foi capturada como o segundo argumento para o construtor `Exception`. Como uma exceção é lançada a partir do bloco `catch`, `method1` termina e retorna a nova exceção ao método chamador (`main`) na linha 10. O bloco `try` em `main` termina e a exceção é capturada nas linhas 12–15. A linha 14 imprime um rastreamento de pilha.

Note na saída do programa que as primeiras três linhas mostram a exceção mais recente que foi lançada (isto é, aquela de `method1` na linha 27). As próximas quatro linhas indicam a exceção que foi lançada a partir de `method2` na linha 40. Por fim, as quatro últimas linhas representam a exceção que foi lançada de `method3` na linha 47. Note também que, se você ler a saída no sentido inverso, ela mostra quantas exceções encadeadas restam.

11.11 Declarando novos tipos de exceção

A maioria dos programadores em Java utiliza as classes existentes da Java API, fornecedores independentes e bibliotecas de classe livremente disponíveis (normalmente descarregáveis a partir da Internet) para construir aplicativos Java. Em geral, os métodos dessas classes são declarados para lançar exceções apropriadas se ocorrer algum problema. Você escreve código que processa essas exceções existentes para tornar os programas mais robustos.

Se construir classes que outros programadores irão utilizar, seria útil declarar suas próprias classes de exceção que são específicas aos problemas que podem ocorrer quando outro programador utiliza suas classes reutilizáveis.

Observação de engenharia de software 11.12

Se possível, indique as exceções provenientes de seus métodos utilizando classes de exceção existentes, em vez de criar novas. A Java API contém muitas classes de exceção que podem ser adequadas ao tipo de problema que seus métodos precisam indicar.

Uma nova classe de exceção deve estender uma classe de exceção existente para assegurar que a classe pode ser utilizada com o mecanismo de tratamento de exceções. Como qualquer outra classe, uma classe de exceção pode conter campos e métodos. Mas uma nova classe de exceção típica contém somente quatro construtores: um que não recebe nenhum argumento e passa uma mensagem de erro padrão `String` para o construtor de superclasse; um que recebe uma mensagem de erro personalizada como uma `String` e passa-a para o construtor de superclasse; um que recebe uma mensagem de erro personalizada como uma `String` e um `Throwable` (para exceções encadeadas) e passa ambos para o construtor de superclasse; e um que recebe um `Throwable` (para exceções encadeadas) e passa-o para o construtor de superclasse.

Boa prática de programação 11.3

Associar cada tipo de malfuncionamento sério em tempo de execução com uma classe `Exception` apropriadamente identificada aprimora a clareza do programa.

Observação de engenharia de software 11.13

Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na Java API e tente estender uma classe de exceção relacionada. Por exemplo, se estiver criando uma nova classe para representar quando um método tenta uma divisão por zero, você poderia estender a classe `ArithmeticException`, porque a divisão por zero ocorre durante a aritmética. Se as classes existentes não forem superclasses apropriadas para sua nova classe de exceção, decida se a nova classe deve ser uma classe de exceção verificada ou não verificada. A nova classe de exceção deve ser uma exceção verificada (isto é, estende `Exception` mas não `RuntimeException`) se clientes precisarem tratar a exceção. A aplicação cliente deve ser razoavelmente capaz de se recuperar de tal exceção. A nova classe de exceção deve estender `RuntimeException` se o código do cliente deve ser capaz de ignorar a exceção (isto é, a exceção é uma exceção não verificada).

No Capítulo 22, Estruturas de dados genéricas personalizadas, fornecemos um exemplo de uma classe de exceção personalizada. Declaramos uma classe reutilizável chamada `List` que é capaz de armazenar uma lista de referências a objetos. Algumas operações normalmente executadas em uma `List` não são permitidas se `List` estiver vazia, como remover um item da parte da frente ou de trás da lista. Por essa razão, alguns métodos `List` lançam exceções da classe de exceções `EmptyListException`.

Boa prática de programação 11.4

Por convenção, todos os nomes de classe de exceções devem terminar com a palavra `Exception`.

11.12 Pré-condições e pós-condições

Os programadores passam longos períodos de tempo mantendo e depurando código. Para facilitar essas tarefas e aprimorar todo o design, eles podem especificar os estados esperados antes e depois da execução de um método. Esses estados são chamados pré-condições e pós-condições, respectivamente.

Uma **pré-condição** deve ser verdadeira quando um método é invocado. Pré-condições descrevem restrições nos parâmetros de método e quaisquer outras expectativas que o método tem sobre o estado atual de um programa um pouco antes de ele começar a executar. Se as pré-condições não forem atendidas, o comportamento do método será então indefinido — ele pode lançar uma exceção, prosseguir com um valor ilegal ou tentar recuperar-se do erro. Você nunca deve esperar um comportamento consistente se as pré-condições não forem satisfeitas.

Uma **pós-condição** é verdadeira depois que o método retorna com sucesso. As pós-condições descrevem as restrições sobre o valor de retorno e quaisquer outros efeitos colaterais que o método possa apresentar. Ao chamar um método, você pode assumir que um método cumpre todas as suas pós-condições. Se estiver escrevendo seu próprio método, você deve documentar todas as pós-condições para que outros saibam o que esperar quando eles chamam seu método, e você deve certificar-se de que seu método cumpre todas as pós-condições se as pré-condições forem de fato satisfeitas.

Se as pré-condições ou pós-condições não forem satisfeitas, os métodos costumam lançar exceções. Como um exemplo, examine método `String` `charAt`, que tem um parâmetro `int` — um índice em `String`. Para uma pré-condição, o método `charAt` assume que `index` é maior que ou igual a zero e menor que o comprimento da `String`. Se a pré-condição é atendida, a pós-condição declara que o método retornará o caractere à posição da `String` especificada pelo parâmetro `index`. Caso contrário, o método lança uma `IndexOutOfBoundsException`. Confiamos que o método `charAt` satisfaz sua pós-condição, desde que atendamos à pré-condição. Não precisamos nos preocupar com os detalhes de como o método realmente recupera o caractere no índice.

Alguns programadores declaram as pré-condições e pós-condições informalmente como parte da especificação de método geral, enquanto outros preferem uma abordagem mais formal definindo-as explicitamente. Ao projetar seus próprios métodos, você deve declarar as pré-condições e pós-condições em um comentário antes da declaração de método da maneira que preferir. Declarar as pré-condições e pós-condições antes de escrever um método também ajudará a orientá-lo durante a implementação do método.

11.13 Assertivas

Ao implementar e depurar uma classe, às vezes é útil declarar as condições que devem ser verdadeiras em um ponto particular de um método. Essas condições, chamadas de **assertivas**, ajudam a assegurar a validade de um programa capturando bugs potenciais e identificando possíveis erros de lógica durante o desenvolvimento. As pré-condições e as pós-condições são dois tipos de assertivas. As pré-condições são assertivas sobre seu estado quando um método é invocado e as pós-condições são assertivas sobre o estado de um programa depois do encerramento de um método.

Embora as assertivas possam ser declaradas como comentários para orientar-lhe durante o desenvolvimento, o Java inclui duas versões da instrução **`assert`** para validar as assertivas programaticamente. A instrução `assert` avalia uma expressão `boolean` e, se `false`, lança um **`AssertionError`** (uma subclasse de `Error`). A primeira forma da instrução `assert` é

```
assert expressão;
```

que lança um `AssertionError` se *a expressão* for `false`. A segunda forma é

```
assert expressão1 : expressão2;
```

que avalia *expressão1* e lança um `AssertionError` com *expressão2* como a mensagem de erro se *expressão1* for `false`.

Você pode utilizar assertivas para implementar programaticamente as pré-condições e pós-condições ou verificar qualquer outro estado intermediário que ajude a assegurar que o código está funcionando corretamente. A Figura 11.9 demonstra a instrução `assert`. A linha 11 apresenta um prompt pedindo ao usuário para inserir um número entre 0 e 10 e, então, a linha 12 lê o número. A linha 15 determina se o usuário inseriu um número dentro do intervalo válido. Se o número estiver fora do intervalo, a instrução `assert` informará um erro; do contrário, o programa prossegue normalmente.

```
1  // Figura 11.9: AssertTest.java
2  / Verificando com assert que um valor está dentro do intervalo
3  import java.util.Scanner;
4
```

```
5  public class AssertTest
6  {
7     public static void main( String[] args )
8     {
9        Scanner input = new Scanner( System.in );
10
11       System.out.print( "Enter a number between 0 and 10: " );
12       int number = input.nextInt();
13
14       // faz a assertiva de que o valor é >= 0 e <= 10
15       assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17       System.out.printf( "You entered %d\n", number );
18    } // fim de main
19 } // fim da classe AssertTest
```

```
Enter a number between 0 and 10: 5
You entered 5
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Figura 11.9 | Verificando com `assert` se um valor está dentro do intervalo.

Você utiliza assertivas principalmente para depurar e identificar erros de lógica em um aplicativo. Você deve ativar explicitamente as assertivas ao executar um programa, porque elas reduzem o desempenho e são desnecessárias ao usuário do programa. Para fazer isso, utilize a opção de linha de comando `-ea` do comando `java`, como em

```
java -ea AssertTest
```

Os usuários não devem encontrar quaisquer `AssertionErrors` pela execução normal de um programa adequadamente escrito. Esses erros devem apenas indicar bugs na implementação. Como resultado, você nunca deve capturar um `AssertionError`. Em vez disso, deve permitir que o programa termine quando ocorrer erros, de modo que você possa ver a mensagem de erro e, então, localizar e corrigir a fonte do problema. Como os usuários do aplicativo podem escolher não ativar assertivas em tempo de execução, você não deve utilizar `assert` para indicar problemas de tempo de execução no código de produção. Em vez disso, deve utilizar o mecanismo de exceção para esse propósito.

11.14 Conclusão

Neste capítulo, você aprendeu a utilizar o tratamento de exceções para lidar com erros. Você aprendeu que o tratamento de exceções permite-lhe remover código de tratamento de erro a partir da "linha principal" da execução do programa. Mostramos como utilizar blocos `try` para incluir código que pode lançar uma exceção e como utilizar blocos `catch` para lidar com possíveis exceções. Você aprendeu sobre o modelo de terminação de tratamento de exceções, que determina que depois de uma exceção ser tratada, o controle de programa não retorna ao ponto de lançamento. Discutimos exceções verificadas versus não verificadas e como especificar com a cláusula `throws` as exceções que um método poderia lançar. Você aprendeu como utilizar o bloco `finally` para liberar recursos, se ocorrer uma exceção ou não. Você também aprendeu como lançar e relançar exceções. Mostramos como obter informações sobre uma exceção utilizando os métodos `printStackTrace`, `getStackTrace` e `getMessage`. Em seguida, apresentamos exceções encadeadas, que permitem empacotar as informações originais sobre uma exceção com novas informações sobre uma exceção. Então, mostramos como criar suas próprias classes de exceção. Introduzimos as pré-condições e pós-condições para ajudar os programadores que utilizam seus métodos a entender as condições que devem ser verdadeiras quando o método é chamado e quando ele retorna. Se as pré-condições e pós-condições não forem satisfeitas, os métodos costumam lançar exceções. Por fim, discutimos a instrução `assert` e como ela pode ser utilizada para ajudar a depurar programas. Especialmente, `assert` pode ser utilizada para assegurar que as pré-condições e pós-condições são satisfeitas. No próximo capítulo, começaremos nosso estudo de caso opcional de dois capítulos sobre o projeto orientado a objetos com a UML.

Resumo

Seção 11.1 Introdução

- Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa.
- O tratamento de exceções permite aos programadores criar aplicativos que podem resolver exceções.

Seção 11.2 Visão geral do tratamento de erros

- O tratamento de exceções permite-lhe remover da "linha principal" de execução do programa o código de tratamento de erro, aprimorando a clareza do programa e destacando sua capacidade de modificação.

Seção 11.3 Exemplo de divisão por zero sem tratamento de exceções

- As exceções são lançadas quando um método detecta um problema e é incapaz de tratá-lo.
- O rastreamento de pilha de uma exceção inclui o nome da exceção em uma mensagem que indica o tipo de problema que ocorreu e a pilha completa de chamadas de método no momento em que a exceção ocorreu.
- O ponto no programa em que uma exceção ocorre é chamado de ponto de lançamento.

Seção 11.4 Exemplo de tratamento de `ArithmeticExceptions` e `InputMismatchExceptions`

- Um bloco `try` inclui o código que talvez lance (`throw`) uma exceção e o código que não deve executar se essa exceção ocorrer.
- As exceções podem emergir por meio de código explicitamente mencionado em um bloco `try`, por chamadas para outros métodos ou até mesmo pelas chamadas de método profundamente aninhadas iniciadas pelo código no bloco `try`.
- Um bloco `catch` inicia com a palavra-chave `catch` e um parâmetro de exceção seguido por um bloco de código que trata a exceção. Esse código executa quando o bloco `try` detecta a exceção.
- Uma exceção não capturada é uma exceção que ocorre para a qual não há nenhum bloco `catch` correspondente.
- Uma exceção não capturada fará com que um programa termine antes da hora se esse programa contiver somente uma thread. Caso contrário, somente a thread em que a exceção ocorreu terminará. O restante do programa executará, mas possivelmente com resultados adversos.
- Pelo menos um bloco `catch` ou um bloco `finally` deve seguir imediatamente o bloco `try`.
- Um bloco `catch` especifica entre parênteses um parâmetro de exceção que identifica o tipo de exceção a tratar. O nome do parâmetro de exceção permite ao bloco `catch` interagir com um objeto de exceção capturado.
- Se uma exceção ocorrer em um bloco `try`, o bloco `try` terminará imediatamente e o controle do programa irá passar para o primeiro bloco `catch` com um tipo de parâmetro que corresponde ao tipo da exceção lançada.
- Depois que uma exceção é tratada, o controle de programa não retorna ao ponto de lançamento, porque o bloco `try` expirou. Isso é conhecido como o modelo de terminação de tratamento de exceções.
- Se houver múltiplos blocos `catch` correspondentes quando uma exceção ocorrer, somente o primeiro é executado.
- Uma cláusula `throws` especifica uma lista separada por vírgula das exceções que o método poderia lançar e aparece depois da lista de parâmetros do método e antes do corpo do método.

Seção 11.5 Quando usar tratamento de exceções

- O tratamento de exceções processa erros síncronos, que ocorrem quando uma instrução executa.
- O tratamento de exceções não é projetado para processar problemas associados com eventos assíncronos, que ocorrem paralelamente com o fluxo do programa de controle e independentemente dele.

Seção 11.6 Hierarquia de exceções no Java

- Todas as classes de exceção do Java herdam direta ou indiretamente da classe `Exception`.
- Os programadores podem estender a hierarquia de exceções do Java com suas próprias classes de exceção.
- A classe `Throwable` é a superclasse da classe `Exception` e, portanto, também é a superclasse de todas as exceções. Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.
- A classe `Throwable` tem duas subclasses: `Exception` e `Error`.
- A classe `Exception` e suas subclasses representam problemas que poderiam ocorrer em um programa Java e ser capturados pelo aplicativo.
- A classe `Error` e suas subclasses representam problemas que poderiam acontecer no sistema de tempo de execução do Java. `Errors` raramente acontecem e, em geral, não devem ser capturados por um aplicativo.
- O Java distingue entre duas categorias de exceções: verificadas e não verificadas.

- O compilador do Java não verifica para determinar se uma exceção não verificada foi capturada ou declarada. Em geral, pode-se impedir a ocorrência de exceções não verificadas pela codificação adequada.
- Subclasses de `RuntimeException` representam exceções não verificadas. Todos os tipos de exceção que herdam da classe `Exception` mas não da `RuntimeException` são exceções verificadas.
- Se um bloco `catch` é escrito para capturar objetos de exceção de um tipo de superclasse, ele também pode capturar todos os objetos das subclasses dessa classe. Isso permite processamento polimórfico de exceções relacionadas.

Seção 11.7 Bloco `finally`

- Os programas que obtêm certos tipos de recursos devem retorná-los ao sistema explicitamente para evitar os supostos vazamentos de recurso. O código de liberação de recurso é geralmente colocado em um bloco `finally`.
- O bloco `finally` é opcional. Se estiver presente, ele é colocado depois do último bloco `catch`.
- O Java garante que o bloco `finally` executará se uma exceção for lançada no bloco `try` correspondente ou em qualquer um de seus blocos `catch` correspondentes.
- Se uma exceção não puder ser capturada por um dos handlers `catch` associados ao bloco `try`, o controle passa para o bloco `finally`. A exceção é então transferida para o próximo bloco `try` externo.
- Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará. A exceção é então transferida para o próximo bloco `try` externo.
- Uma instrução `throw` pode lançar qualquer objeto `Throwable`.

Seção 11.8 Desempilhamento de pilha

- As exceções são relançadas quando um bloco `catch`, ao receber uma exceção, decide que não pode processar essa exceção ou que só pode processá-la parcialmente. Relançar uma exceção adia o tratamento de exceções (ou talvez uma parte dele) para outro bloco `catch`.
- Quando ocorre um relançamento, o próximo bloco `try` circundante detecta a exceção relançada e os blocos `catch` desse bloco `try` tentam tratá-la.
- Quando uma exceção é lançada mas não é capturada em um escopo particular, a pilha de chamadas de método é desempilhada e uma tentativa de capturar (`catch`) a exceção é feita na próxima instrução `try` externa.

Seção 11.9 `printStackTrace`, `getStackTrace` e `getMessage`

- A classe `Throwable` oferece um método `printStackTrace` que imprime a pilha de chamadas de método. Frequentemente, isso é útil no processo de teste e depuração.
- A classe `Throwable` também fornece um método `getStackTrace` que obtém as mesmas informações de rastreamento de pilha que são impressas por `printStackTrace`.
- O método `getMessage` da classe `Throwable` retorna a string descritiva armazenada em uma exceção.
- O método `getStackTrace` obtém as informações de rastreamento de pilha como um array de objetos `StackTraceElement`. Todo `StackTraceElement` representa uma chamada de método na pilha de chamadas de método.
- Os métodos `StackTraceElement` `getClassName`, `getFileName`, `getLineNumber` e `getMethodName` obtêm o nome de classe, nome de arquivo, número de linha e nome de método, respectivamente.

Seção 11.10 Exceções encadeadas

- As exceções encadeadas permitem que um objeto de exceção mantenha as informações do rastreamento de pilha completo, incluindo as informações sobre exceções anteriores que causaram a exceção atual.

Seção 11.11 Declarando novos tipos de exceção

- Uma nova classe de exceção deve estender uma classe de exceção existente para assegurar que a classe pode ser utilizada com o mecanismo de tratamento de exceções.

Seção 11.12 Pré-condições e pós-condições

- A pré-condição de um método é uma condição que deve ser verdadeira quando o método é invocado.
- A pós-condição de um método é uma condição que é verdadeira depois de o método retornar com sucesso.
- Ao projetar seus próprios métodos, você deve declarar as pré-condições e pós-condições em um comentário antes da declaração de método.

Seção 11.13 Assertivas

- Assertivas ajudam a assegurar a validade de um programa capturando potenciais bugs e identificando possíveis erros de lógica.
- O Java inclui duas versões de uma instrução `assert` para validar assertivas programaticamente.
- Para permitir assertivas em tempo de execução, utilize a switch `-ea` ao executar o comando `java`.

Terminologia

aplicativo robusto, 336
`ArithmeticException`, classe, 337
`assert`, instrução, 354
`AssertionError`, classe, 354
assertiva, 354
capture ou declare, requisito, 343
`catch`, bloco, 340
`catch`, cláusula, 340
`catch`, palavra-chave, 340
desempilhando, 348
erro síncrono, 342
`Error`, classe, 343
evento assíncrono, 342
exceção, 336
exceção encadeada, 351
exceção não interceptada, 341
exceção não verificada, 343
exceção verificada, 343
`Exception`, classe, 343
`exit`, método da classe `System`, 345
`finally`, bloco, 341
`finally`, cláusula, 345
`finally`, palavra-chave, 341
fluxo, 347
fluxo de erro padrão, 341
fluxo de saída padrão, 347
`getClassName`, método da classe `StackTraceElement`, 351
`getFileName`, método da classe `StackTraceElement`, 351
`getLineNumber`, método da classe `StackTraceElement`, 351
`getMessage`, método da classe `Throwable`, 349
`getMethodName`, método da classe `StackTraceElement`, 351
`getStackTrace`, método da classe `Throwable`, 349
handler de exceção, 340
`InputMismatchException`, classe, 338
lançar uma exceção, 337
modelo de retomada de tratamento de exceções, 341
modelo de retomada do tratamento de exceções, 341
parâmetro de exceção, 341
ponto de lançamento, 338
pós-condição, 354
pré-condição, 354
`printStackTrace`, método da classe `Throwable`, 349
programa tolerante a falha, 336
rastreamento de pilha, 337
relançar uma exceção, 348
`RuntimeException`, classe, 343
`StackTraceElement`, classe, 351
`System.err`, (fluxo de erro padrão), 341
thread, 341
`throw`, instrução, 347
`throw`, palavra-chave, 348
`Throwable`, classe, 343
`throws`, cláusula, 342
`throws`, palavra-chave, 342
tolerante a falhas, 336
tratamento de exceções, 336
`try`, bloco, 340
`try`, instrução, 342
`try`, palavra-chave, 340
vazamento de recurso, 345

Exercícios de autorrevisão

11.1 Liste cinco exemplos de exceções comuns.

11.2 Apresente várias razões pelas quais as técnicas de tratamento de exceções não devem ser usadas para o controle convencional de programa.

11.3 Por que as exceções são particularmente adequadas para lidar com erros produzidos por métodos de classes na Java API?

11.4 O que é um "vazamento de recurso"?

11.5 Se nenhuma exceção é lançada em um bloco `try`, onde o controle prossegue quando o bloco `try` completa a execução?

11.6 Cite a principal vantagem de usar `catch( Exception` *nomeDaExceção* `)`.

11.7 Um aplicativo convencional deve capturar objetos `Error`? Explique.

11.8 O que acontece se nenhum handler `catch` corresponder ao tipo de um objeto lançado?

11.9 O que acontece se vários blocos `catch` correspondem ao tipo do objeto lançado?

11.10 Por que um programador especificaria um tipo de superclasse como o tipo em um bloco `catch`?

11.11 Qual é a razão-chave para utilizar blocos `finally`?

11.12 O que acontece quando um bloco `catch` lança uma `Exception`?

11.13 O que a instrução `throw` *referênciaDaExceção* faz em um bloco `catch`?

11.14 O que acontece com uma referência local em um bloco `try` quando esse bloco lança uma `Exception`?

Respostas dos exercícios de autorrevisão

11.1 Esgotamento de memória, índice de array fora dos limites, estouro aritmético, divisão por zero, parâmetros de método inválidos.

11.2 (a) O tratamento de exceções é projetado para tratar situações raras que costumam resultar na terminação do programa, não situações que surgem o tempo todo. (b) O fluxo de controle com estruturas de controle convencionais é geralmente mais claro e mais eficiente do que as exceções. (c) As exceções adicionais podem entrar no caminho de verdadeiras exceções do tipo erro. Fica mais difícil para você monitorar o número maior de casos de exceção resultante.

11.3 É improvável que os métodos de classes na Java API possam realizar processamento de erro que atenda às necessidades particulares de todos os usuários.

11.4 Um "vazamento de recurso" ocorre quando um programa em execução não libera adequadamente um recurso quando ele não é mais necessário.

11.5 Os blocos `catch` para essa instrução `try` são pulados e o programa retoma a execução depois do último bloco `catch`. Se houver um bloco `finally`, ele será executado primeiro; então o programa retomará a execução depois do bloco `finally`.

11.6 A forma `catch( Exception` *nomeDaExceção* `)` captura qualquer tipo de exceção lançada em um bloco `try`. Uma vantagem é que nenhuma `Exception` lançada pode passar sem ser capturada. Voce pode então decidir tratar a exceção ou possivelmente relançá-la.

11.7 `Errors` são problemas normalmente sérios com o sistema Java subjacente; a maioria dos programas não quer capturar `Errors` porque não será capaz de se recuperar deles.

11.8 Isso faz com que a pesquisa por uma correspondência continue na próximo instrução `try` circundante. Se houver um bloco `finally`, ele será executado antes de a exceção ir para a próxima instrução `try` circundante. Se não houver nenhuma instrução `try` circundante para a qual existem blocos `catch` correspondentes, e as exceções forem declaradas (ou não verificadas), um rastreamento de pilha é impresso e a thread atual termina antes. Se as exceções forem verificadas, mas não capturadas ou declaradas, ocorrerão erros de compilação.

11.9 O primeiro bloco `catch` correspondente depois do bloco `try` é executado.

11.10 Isso permite que um programa capture tipos de exceções relacionadas e os processe de maneira uniforme. Entretanto, costuma ser útil processar os tipos de subclasse individualmente para obter um tratamento de exceções mais preciso.

11.11 O bloco `finally` é o meio preferido de liberar recursos para impedir vazamentos de recurso.

11.12 Primeiro, o controle passa para o bloco `finally` se houver algum. Em seguida, a exceção será processada por um bloco `catch` (se houver algum) associado com um bloco `try` envolvente (se houver algum).

11.13 Ele relança a exceção para o processamento por um handler de exceção de uma instrução `try` envolvente, depois de o bloco `finally` da instrução `try` atual executar.

11.14 A referência sai de escopo. Se o objeto referenciado tornar-se inalcançável, o objeto poderá sofrer uma coleta de lixo.

Exercícios

11.15 Liste as várias condições excepcionais que ocorreram em programas por todo este livro até agora. Liste o maior número possível de condições excepcionais adicionais que você puder. Para cada uma destas, descreva de maneira resumida como um programa trataria tipicamente a exceção utilizando as técnicas de tratamento de exceções discutidas neste capítulo. As exceções típicas incluem a divisão por zero e índice de array fora dos limites.

11.16 Até este capítulo, descobrimos que lidar com erros detectados pelos construtores é um pouco difícil. Explique por que o tratamento de exceções é um meio eficaz de lidar com falha de construtor.

11.17 **(*Capturando exceções com superclasses*)** Utilize herança para criar uma superclasse de exceção (chamada `ExceptionA`) e subclasses de exceção `ExceptionB` e `ExceptionC`, em que `ExceptionB` herda de `ExceptionA` e `ExceptionC` herda de `ExceptionB`. Escreva um programa para demonstrar que o bloco `catch` para tipo `ExceptionA` captura exceções de tipos `ExceptionB` e `ExceptionC`.

11.18 **(*Capturando exceções com a classe `Exception`*)** Escreva um programa que demonstra como várias exceções são capturadas com

```
catch (Exception exception )
```

Desta vez, defina as classes `ExceptionA` (que herda da classe `Exception`) e `ExceptionB` (que herda da classe `ExceptionA`). Em seu programa, crie blocos `try` que lançam exceções de tipos `ExceptionA`, `ExceptionB`, `NullPointerException` e `IOException`. Todas as exceções devem ser capturadas com blocos `catch` para especificar o tipo `Exception`.

11.19 **(*Ordem de blocos `catch`*)** Escreva um programa que mostra que a ordem de blocos `catch` é importante. Se você tentar capturar um tipo de exceção de superclasse antes de um tipo de subclasse, o compilador deve gerar erros.

11.20 **(*Falha de construtor*)** Escreva um programa que mostra um construtor que passa informações sobre a falha do construtor para um handler de exceção. Defina a classe `SomeClass`, que lança um `Exception` no construtor. O seu programa deve tentar criar um objeto do tipo `SomeClass` e capturar a exceção que é lançada do construtor.

11.21 **(*Relançando exceções*)** Escreva um programa que ilustra o relançamento de uma exceção. Defina os métodos `someMethod` e `someMethod2`. O método `someMethod2` deve lançar inicialmente uma exceção. O método `someMethod` deve chamar `someMethod2`, capturar a exceção e relançá-la. Chame `someMethod` a partir do método `main` e capture a exceção relançada. Imprima o rastreamento de pilha dessa exceção.

11.22 **(*Capturando exceções com escopos externos*)** Escreva um programa que mostra que um método com seu próprio bloco `try` não precisa capturar todo possível erro gerado dentro do `try`. Algumas exceções podem escorregar para, e serem tratadas em, outros escopos.

12

A ação fala mais do que as palavras, mas nem sempre.
— **Mark Twain**

Sempre projete algo considerando-o em seu próximo, mais amplo, contexto.
— **Eliel Saarinen**

Oh, a vida é um ciclo glorioso de canções.
— **Dorothy Parker**

O projeto dos irmãos Wright… permitiu-lhes sobreviver o bastante para aprender a voar.
— **Michael Potts**

Estudo de caso de ATM, Parte 1: projeto orientado a objetos com UML

Objetivos

Neste capítulo, você aprenderá:

- Uma metodologia de projeto orientado a objetos simples.
- O que é um documento de requisitos.
- A identificar classes e atributos de classe de um documento de requisitos.
- A identificar estados, atividades e operações de objetos a partir de um documento de requisitos.
- A determinar as colaborações entre objetos em um sistema.
- A trabalhar com o caso de uso, classe, estado, atividade, comunicação e diagramas de sequência da UML para modelar graficamente um sistema orientado a objetos.

Sumário

12.1 Introdução ao estudo de caso
12.2 Examinando o documento de requisitos
12.3 Identificando classes em um documento de requisitos
12.4 Identificando atributos de classe
12.5 Identificando estados e atividades dos objetos
12.6 Identificando operações de classe
12.7 Indicando colaboração entre objetos
12.8 Conclusão
Respostas dos exercícios de autorrevisão

12.1 Introdução ao estudo de caso

Agora iniciaremos a parte opcional de nosso estudo de caso de design e implementação orientados a objetos. Neste capítulo e no Capítulo 13, você projetará e implementará um sistema de software de caixa eletrônico (*Automated Teller Machine* — ATM) orientado a objetos. O estudo de caso fornece-lhe uma experiência de design e implementação concisa, passo a passo e completa. Nas seções 12.2 a 12.7 e 13.2 a13.3, você seguirá os passos de um processo de projeto orientado a objetos (*Object-Oriented Design* — OOD) que utiliza a UML ao mesmo tempo em que relaciona esses passos aos conceitos orientados a objetos discutidos nos capítulos 2 a 10. Neste capítulo, você trabalhará com seis tipos populares de diagramas de UML para representar o projeto graficamente. No Capítulo 13, você ajustará o projeto com a herança, então implementará inteiramente o ATM em um aplicativo Java de 673 linhas (Seção 13.4).

Isso não é um exercício; em vez disso, é uma experiência de aprendizagem completa que conclui com uma revisão detalhada do código Java completo que implementa nosso design. Você irá se familiarizar com os tipos de problema encontrados nessa área.

Esses capítulos podem ser estudados como uma unidade contínua depois de você completar a introdução à programação orientada a objetos nos capítulos 8 a 11. Ou, você pode avançar passo a passo pelas seções uma após outra depois dos capítulos 2 a 8 e 10. Cada seção do estudo de caso começa com uma nota informando o capítulo depois do qual ela pode ser abordada.

12.2 Examinando o documento de requisitos

[*Nota:* esta seção pode ser ensinada depois do Capítulo 2.]

Começamos nosso processo de design apresentando um **documento de requisitos** que especifica o objetivo do sistema ATM e *o que* ele deve fazer. Em todas as partes do estudo de caso, referimo-nos frequentemente a esse documento de requisitos.

Documento de requisitos

Um banco local pretende instalar um novo caixa eletrônico (*Automated Teller Machine* — ATM) para permitir que os usuários (isto é, clientes do banco) realizem transações financeiras básicas (Figura 12.1). Cada usuário pode ter somente uma conta no banco. Os usuários do ATM devem ser capazes de visualizar seus saldos bancários, sacar dinheiro (isto é, retirar dinheiro de uma conta) e depositar fundos (isto é, colocar dinheiro em uma conta). A interface com o usuário do caixa eletrônico contém:

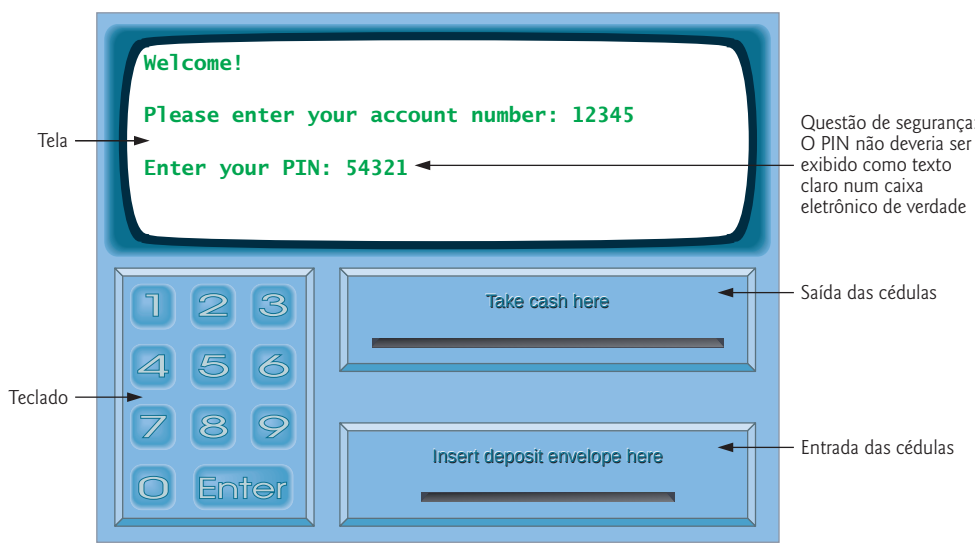


Figura 12.1 | A interface com o usuário do caixa eletrônico.

- uma tela que exibe as mensagens para o usuário
- um teclado que recebe a entrada numérica do usuário
- um dispensador de cédulas que disponibiliza o dinheiro para o usuário e
- uma abertura para depósito que recebe os envelopes com o depósito do usuário.

O dispensador de cédulas é carregado diariamente com US$ 500 em notas de US$ 20. [*Nota:* devido ao escopo limitado desse estudo de caso, certos elementos do ATM descritos aqui não simulam exatamente aqueles de um ATM real. Por exemplo, um ATM real normalmente contém um dispositivo que lê o número da conta de um usuário a partir de um cartão ATM, enquanto este ATM solicita que o usuário digite o número da conta no teclado. Normalmente, um ATM real também imprime um recibo no fim de uma sessão, mas toda a saída deste ATM aparece na tela.]

O banco quer que você desenvolva um software para realizar as transações financeiras iniciadas pelos clientes do banco por meio do ATM. O banco integrará o software com o hardware do ATM em um momento posterior. O software deve encapsular a funcionalidade dos dispositivos de hardware (por exemplo, o dispensador de cédulas, a abertura para depósito) dentro dos componentes de software, mas ele próprio não precisa se preocupar com a maneira como esses dispositivos realizam suas tarefas. O hardware do ATM ainda não foi desenvolvido, assim, em vez de escrever seu software para ser executado no ATM, você deve desenvolver uma primeira versão para executar em um computador pessoal. Essa versão deve utilizar o monitor do computador para simular a tela do ATM e o teclado do computador para simular o teclado do ATM.

Uma sessão do ATM consiste na autenticação de um usuário (isto é, provar a identidade do usuário) com base em um número de conta e número de identificação pessoal (*Personal Identification Number* — PIN) para então criar e executar as transações financeiras. Para autenticar um usuário e realizar as transações, o ATM deve interagir com o banco de dados das informações da conta bancária (isto é, uma coleção organizada de dados armazenada em um computador; estudaremos o acesso a banco de dados no Capítulo 28). Para cada conta bancária, o banco de dados armazena um número de conta, um PIN e um saldo que indica a quantidade de dinheiro na conta. [*Nota:* supomos que o banco planeja construir somente um ATM, portanto não precisamos nos preocupar com múltiplos ATMs acessando esse banco de dados ao mesmo tempo. Além disso, supomos que o banco não faz nenhuma alteração nas informações no banco de dados enquanto um usuário está acessando o ATM. Além disso, qualquer sistema de negócios como um ATM depara-se com questões de segurança razoavelmente complexas que estão além do escopo de um primeiro ou segundo curso de programação. Entretanto, fazemos a suposição simplificadora de que o banco confia no ATM para acesso e manipulação das informações no banco de dados sem medidas de segurança significativas.]

Na primeira abordagem do ATM (supondo que ninguém o está utilizando atualmente), o usuário deve experimentar a seguinte sequência de eventos (mostrada na Figura 12.1):

1. a tela exibe `Welcome!` e pede para o usuário inserir um número de conta.
2. o usuário insere um número de cinco dígitos da conta utilizando o teclado.
3. a tela solicita que o usuário insira o PIN associado com o número da conta especificada.
4. o usuário insere um PIN de cinco dígitos utilizando o teclado.[1]
5. se o usuário inserir um número de conta válida e o PIN correto para essa conta, a tela exibe o menu principal (Figura 12.2). Se o usuário inserir um número inválido de conta ou um PIN incorreto, a tela exibe uma mensagem apropriada e então o ATM retorna ao *Passo 1* para reiniciar o processo de autenticação.

Depois de o ATM autenticar o usuário, o menu principal (Figura 12.2) deve conter uma opção numerada para cada um dos três tipos de transações: Consulta de saldos (opção 1), retirada (opção 2) e depósito (opção 3). Ele também deve conter uma opção para permitir ao usuário sair do sistema (opção 4). O usuário então opta por realizar uma transação (inserindo 1, 2 ou 3) ou sair do sistema (inserindo 4).

Se o usuário inserir 1 para fazer uma consulta de saldos, a tela exibe o saldo da conta do usuário. Para fazer isso, o ATM deve recuperar o saldo a partir do banco de dados do banco.

Os passos a seguir descrevem o que ocorre quando o usuário insere 2 para fazer um saque:

1. a tela exibe um menu (mostrado na Figura 12.3) que contém valores padrão de saque: US$ 20 (opção 1), US$ 40 (opção 2), US$ 60 (opção 3), US$ 100 (opção 4) e US$ 200 (opção 5). O menu também contém uma opção para permitir que o usuário cancele a transação (opção 6).
2. o usuário entra em uma seleção de menu utilizando o teclado.
3. se o valor do saque escolhido for maior que o saldo da conta do usuário, a tela exibe uma mensagem declarando isso e pedindo ao usuário para escolher um valor menor. O ATM então retorna ao *Passo 1*. Se o valor do saque escolhido for menor ou igual ao saldo da conta do usuário (isto é, um valor aceitável), o ATM prossegue para o *Passo 4*. Se o usuário optar por cancelar a transação (opção 6), o ATM exibe o menu principal e espera pela entrada do usuário.

[1] Neste ATM simples, baseado em texto e linha de comando, quando você digita o PIN, ele aparece na tela. Isso é uma violação de segurança óbvia — você não iria querer que alguém olhasse pelas costas em um ATM e visse o seu PIN exibido na tela. No Capítulo 14, apresentamos o componente GUI `JPasswordField`, que exibe asteriscos como os tipos de usuário — tornando-o mais apropriado para inserir números PIN e senhas. O Exercício 14.18 solicita a construção de uma versão baseada em GUI do ATM e o uso de um `JPasswordField` para obter o PIN do usuário.

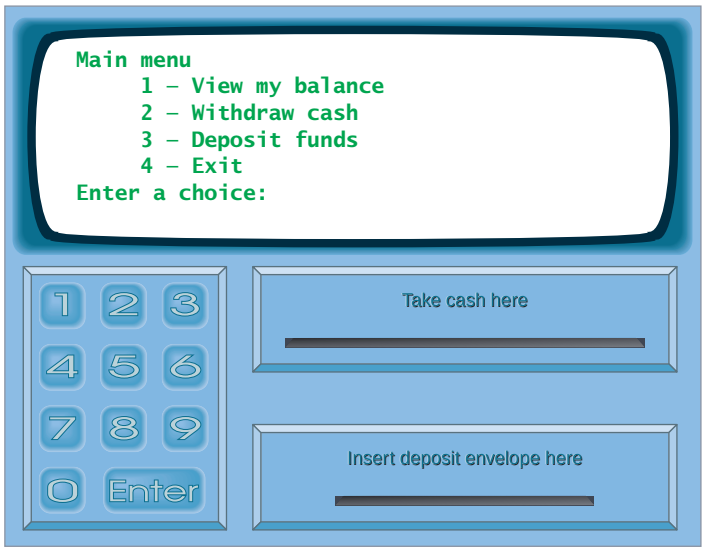


Figura 12.2 | Menu principal do ATM.

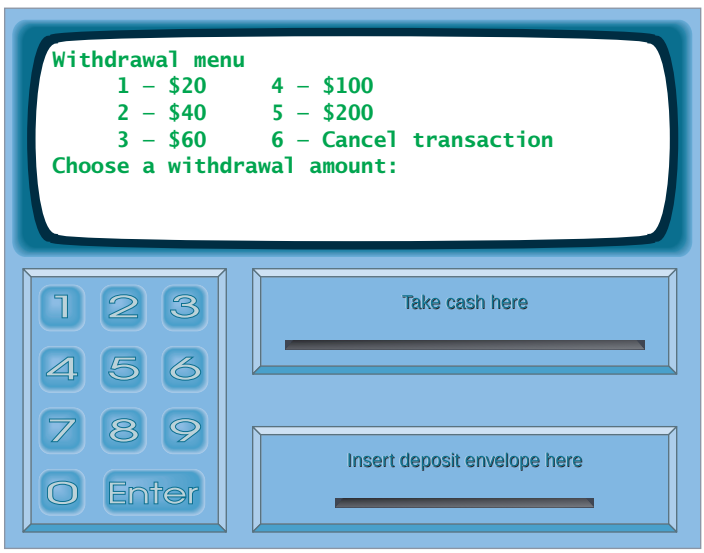


Figura 12.3 | Menu de saque do ATM.

4. se o dispensador de cédulas contiver dinheiro suficiente para atender a solicitação, o ATM prossegue para o *Passo 5*. Do contrário, a tela exibe uma mensagem indicando o problema e instruindo o usuário a escolher um valor de saque menor. O ATM retorna então ao *Passo 1*.
5. o ATM debita o valor do saque na conta do usuário no banco de dados do banco (isto é, subtrai o valor do saque do saldo da conta do usuário).
6. o dispensador de cédulas entrega o valor desejado para o usuário.
7. a tela exibe uma mensagem lembrando o usuário de pegar o dinheiro.

Os passos a seguir descrevem as ações que ocorrem quando o usuário insere 3 (examinando o menu principal da Figura 12.2) para fazer um depósito:

1. a tela solicita que o usuário insira um valor de depósito ou que digite 0 (zero) para cancelar.
2. o usuário insere um valor de depósito ou 0 utilizando o teclado. [*Nota:* o teclado não contém um ponto de fração decimal nem um sinal de cifrão, portanto o usuário não pode digitar um valor monetário real (por exemplo, US$ 27,25). Em vez disso, o usuário deve inserir um valor de depósito como um número de centavos (por exemplo, 2725). O ATM divide então esse número por 100 para obter um número que represente um valor monetário (por exemplo, 2725 ÷ 100 = 27.25).]

3. se o usuário especificar um valor de depósito, o ATM prossegue para o *Passo 4*. Se o usuário optar por cancelar a transação (inserindo 0), o ATM exibe o menu principal e espera pela entrada do usuário.
4. a tela exibe uma mensagem pedindo para o usuário inserir um envelope de depósito.
5. se a abertura de depósito receber um envelope de depósito dentro de dois minutos, o ATM credita o valor do depósito na conta do usuário no banco de dados do banco (isto é, adiciona o valor do depósito ao saldo da conta do usuário). [*Nota:* esse valor não permanece imediatamente disponível para saque. Primeiro o banco deve verificar fisicamente o valor do dinheiro no envelope de depósito e quaisquer cheques no envelope devem ser transferidos do emissor do cheque para a conta do depositário. Quando qualquer um desses eventos ocorre, o banco atualiza apropriadamente o saldo do usuário armazenado no seu banco de dados. Isso ocorre independentemente do sistema ATM.] Se a abertura para depósito não receber o envelope de um depósito dentro desse período, a tela exibe uma mensagem informando que o sistema cancelou a transação devido à inatividade. O ATM exibe então o menu principal e espera a entrada do usuário.

Depois de o sistema executar uma transação com sucesso, ele deve retornar ao menu principal de modo que o usuário possa realizar transações adicionais. Se o usuário sair do sistema, a tela exibe uma mensagem de agradecimento e então exibe a mensagem de boas-vindas para o próximo usuário.

Analisando o sistema ATM

A instrução anterior é um exemplo simplificado de um documento de requisitos. Em geral, esse documento é o resultado de um processo detalhado de **coleta de requisitos**, que poderia incluir entrevistas com possíveis usuários do sistema e especialistas em campos relacionados ao sistema. Por exemplo, um analista de sistemas que é contratado para preparar um documento de requisitos para software de operações bancárias (por exemplo, o sistema ATM descrito aqui) poderia entrevistar especialistas em bancos para obter um melhor entendimento sobre o que o software deve fazer. O analista utilizaria as informações obtidas para compilar uma lista de **requisitos de sistema** a fim de orientar os projetistas de sistemas à medida que eles projetam o sistema.

O processo de coleta de requisitos é uma tarefa-chave da primeira etapa do ciclo de vida de software. O **ciclo de vida de software** especifica as etapas pelas quais o software passa desde o momento em que é inicialmente concebido até o momento em que é retirado de uso. Essas etapas em geral incluem: análise, design, implementação, teste e depuração, implantação, manutenção e aposentadoria. Há vários modelos de ciclo de vida de software, cada um com suas próprias preferências e especificações para o momento e a frequência com que os engenheiros de software devem realizar cada uma dessas etapas. **Modelos em cascata** realizam cada etapa uma vez em sucessão, enquanto **modelos iterativos** podem repetir uma ou mais etapas várias vezes por todo um ciclo de vida do produto.

A etapa de análise focaliza a definição do problema a ser resolvido. Ao projetar qualquer sistema, deve-se *solucionar corretamente o problema*, mas, igualmente importante, deve-se *solucionar o problema correto*. Os analistas de sistemas coletam os requisitos que indicam o problema específico a ser solucionado. Nosso documento de requisitos descreve os requisitos do nosso sistema ATM em detalhes suficientes para que você não precise passar por uma extensa etapa de análise — isso já foi feito para você.

Para capturar o que um sistema proposto deve fazer, os desenvolvedores costumam empregar uma técnica conhecida como **modelagem de caso de uso**. Esse processo identifica os **casos de uso** do sistema, cada qual representando uma capacidade diferente que o sistema fornece a seus clientes. Por exemplo, ATMs em geral têm vários casos de uso, como "Visualizar saldo de conta", "Sacar dinheiro", "Depositar fundos", "Transferir fundos entre contas" e "Comprar selos de postagem". O sistema ATM simplificado que construímos nesse estudo de caso permite somente os três primeiros.

Cada caso de uso descreve um cenário típico pelo qual o usuário utiliza o sistema. Você já leu descrições dos casos de uso do sistema ATM no documento de requisitos; as listas de passos necessários para realizar cada tipo de transação (isto é, consulta de saldo, saque e depósito) na verdade descreveram os três casos de uso do nosso ATM — "Visualizar saldo em conta", "Sacar dinheiro" e "Depositar fundos", respectivamente.

Diagramas de casos de uso

Agora, introduziremos o primeiro dos vários diagramas da UML ao estudo de caso. Criamos um **diagrama de casos de uso** para modelar as interações entre os clientes de um sistema (nesse estudo de caso, clientes do banco) e seus casos de uso. O objetivo é mostrar os tipos de interações entre usuários e um sistema sem fornecer os detalhes — estes são fornecidos em outros diagramas da UML (que apresentamos por todo esse estudo de caso). Os diagramas de caso de uso são frequentemente acompanhados por texto informal que apresenta mais detalhes — como o texto que aparece no documento de requisitos. Os diagramas de casos de uso são produzidos durante a etapa de análise do ciclo de vida do software. Em sistemas maiores, diagramas de casos de uso são ferramentas indispensáveis que ajudam projetistas de sistema a permanecerem concentrados na satisfação das necessidades dos usuários.

A Figura 12.4 mostra o diagrama dos casos de uso para nosso sistema ATM. A garatuja representa um **ator**, que define os papéis que uma entidade externa — como uma pessoa ou um outro sistema — reproduz ao interagir com o sistema. Para nosso ATM, o ator é o Usuário que pode visualizar um saldo em conta, sacar dinheiro e depositar fundos no ATM. O Usuário não é uma pessoa real, mas em vez disso abrange os papéis que uma pessoa real — ao reproduzir a parte de um Usuário — pode reproduzir ao interagir com o ATM. Observe que um diagrama de caso de uso pode incluir múltiplos atores. Por exemplo, o diagrama de caso de uso para o sistema ATM de um banco real poderia também incluir um ator chamado Administrador que recarrega o dispensador de cédulas todos os dias.

Nosso documento de requisitos fornece os atores — "Usuários do ATM devem ser capazes de visualizar seus saldos em contas, sacar dinheiro e depositar fundos". Portanto, o ator em cada um dos três casos de uso é o usuário que interage com o ATM. Uma entidade externa — uma pessoa real — desempenha o papel do usuário para realizar transações financeiras. A Figura 12.4 mostra um ator, cujo nome, Usuário, aparece abaixo do ator no diagrama. A UML modela cada caso de uso como uma oval conectada a um ator com uma linha sólida.

Os engenheiros de software (mais precisamente, os projetistas de sistemas) devem analisar o documento de requisitos ou um conjunto de casos de uso e projetar o sistema antes de os programadores implementá-lo em uma linguagem de programação particular. Durante a etapa de análise, os projetistas de sistemas se concentram no entendimento do documento de requisitos para produzir uma especificação de alto nível que descreve *o que* o sistema deve fazer. A saída da etapa de design — uma **especificação de design** — deve especificar claramente *como* o sistema deve ser construído a fim de satisfazer esses requisitos. Nas várias seções a seguir, realizamos os passos do processo de um design simples orientado a objetos (OOD) no sistema ATM para produzir uma especificação de design que contém uma coleção de diagramas da UML e texto de suporte.

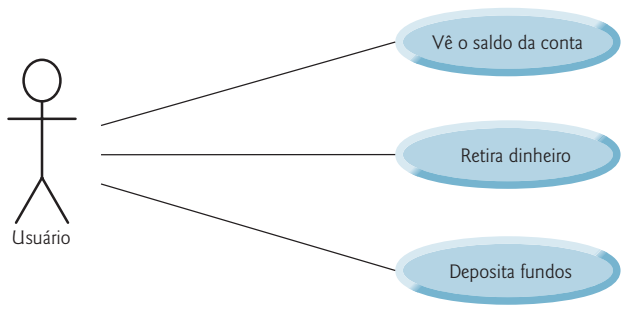


Figura 12.4 | Diagrama de casos de uso para o sistema ATM da perspectiva do Usuário.

A UML é projetada para utilização com qualquer processo OOD. Existem muitos desses processos, o mais conhecido é o Rational Unified Process™ (RUP) desenvolvido pela Rational Software Corporation, agora parte da IBM. O RUP é um processo rico concebido para projetar aplicativos com "poder industrial". Para esse estudo de caso, apresentamos nosso próprio processo simplificado de design, concebido para alunos no primeiro e segundo cursos de programação.

Projetando o sistema ATM

Agora, começamos a etapa de design do nosso sistema ATM. Um **sistema** é um conjunto de componentes que interage para resolver um problema. Por exemplo, para que o sistema ATM realize as tarefas projetadas, nosso sistema ATM tem uma interface com o usuário (Figura 12.1) e contém um software que executa as transações financeiras e interage com um banco de dados das informações de conta bancária. A **estrutura do sistema** descreve os objetos do sistema e seus inter-relacionamentos. O **comportamento do sistema** descreve como o sistema muda à medida que seus objetos interagem entre si.

Cada sistema apresenta tanto uma estrutura como um comportamento — os projetistas devem especificar ambos. Há vários tipos de estruturas e comportamentos de sistema. Por exemplo, as interações entre objetos no sistema diferem daquelas entre o usuário e o sistema, contudo ambas constituem uma parte do comportamento de sistema.

A UML 2 padrão especifica 13 tipos de diagrama para documentar os modelos de sistema. Cada um modela uma característica distinta da estrutura ou comportamento de um sistema — seis diagramas se relacionam com a estrutura do sistema, os sete remanescentes com o comportamento do sistema. Listamos aqui apenas os seis tipos de diagrama utilizados no nosso estudo de caso — um modela a estrutura de sistema; os outros cinco modelam o comportamento de sistema. Fornecemos uma visão geral dos demais sete tipos de diagrama de UML no Apêndice P, UML 2: Tipos de diagrama adicionais.

1. **Diagramas de casos de uso**, como o da Figura 12.4, modelam as interações entre um sistema e suas entidades externas (atores) em termos de casos de uso (capacidades do sistema, como "Visualizar saldo em conta", "Sacar dinheiro" e "Depositar fundos").
2. **Diagramas de classe**, que você estudará na Seção 12.3, modelam as classes, ou os "blocos de construção" utilizados em um sistema. Cada substantivo ou "aspecto" descrito no documento de requisitos é candidato a ser uma classe no sistema (por exemplo, `Account`, `Keypad`). Os diagramas de classe nos ajudam a especificar os relacionamentos estruturais entre partes do sistema. Por exemplo, o diagrama de classe do sistema ATM especificará que o ATM é fisicamente composto de uma tela, teclado, dispensador de cédulas e uma abertura para depósito.
3. **Diagramas de estados de máquina**, que você estudará na Seção 12.5, modelam como um objeto muda de estado. O **estado** de um objeto é indicado pelos valores de todos os seus atributos em um determinado momento. Quando um objeto muda de estado, ele pode comportar-se diferentemente no sistema. Por exemplo, depois de validar o PIN de um usuário, o ATM passa do estado "usuário não autenticado" para o estado "usuário autenticado", quando permite ao usuário realizar transações financeiras (por exemplo, visualizar o saldo da conta, sacar dinheiro, depositar fundos).
4. **Diagramas de atividades**, que você também estudará na Seção 12.5, modelam a **atividade** de um objeto — é o fluxo de trabalho (sequência de eventos) durante a execução do programa. Um diagrama de atividades modela as ações que o objeto realiza e especifica

a ordem em que ele as realiza. Por exemplo, um diagrama de atividades mostra que o ATM deve obter o saldo em conta do usuário (a partir do banco de dados com as informações da conta bancária) *antes* de a tela poder exibir o saldo para o usuário.

5. **Diagramas de comunicação** (chamados de **diagramas de colaboração** nas versões anteriores da UML) modelam as interações entre os objetos em um sistema, com ênfase em *quais* interações ocorrem. Você aprenderá na Seção 12.7 que esses diagramas mostram quais objetos devem interagir para realizar uma transação no ATM. Por exemplo, o ATM deve se comunicar com o banco de dados com as informações da conta bancária para obter o saldo em uma conta.
6. **Diagramas de sequência** também modelam as interações entre os objetos em um sistema, mas, diferentemente dos diagramas de comunicação, eles enfatizam *quando* as interações ocorrem. Você aprenderá na Seção 12.7 que esses diagramas ajudam a mostrar a ordem em que as interações ocorrem ao executar uma transação financeira. Por exemplo, a tela solicita que o usuário insira um valor de saque antes de o dinheiro ser dispensado.

Na Seção 12.3, continuamos a projetar nosso sistema ATM identificando as classes no documento de requisitos. Realizamos isso extraindo substantivos e locuções substantivas chave do documento de requisitos. Utilizando essas classes, desenvolveremos nosso primeiro rascunho do diagrama de classe que modela a estrutura do nosso sistema ATM.

Recursos da Web

Criamos um extenso Centro de Recursos de UML que contém muitos links para informações adicionais, incluindo introduções, tutoriais, blogs, livros, certificação, conferências, ferramentas de desenvolvedor, documentação, e-books, FAQs, fóruns, grupos, UML em Java, podcasts, segurança, ferramentas, downloads, cursos de treinamentos, vídeos e muito mais. Nós o encorajamos a navegar pelo nosso Centro de Recursos de UML em `www.deitel.com/UML/` para aprender mais.

Exercícios de autorrevisão da Seção 12.2

12.1 Suponha que permitimos aos usuários do nosso sistema ATM transferir dinheiro entre duas contas bancárias. Modifique o diagrama de caso de uso da Figura 12.4 para refletir essa alteração.

12.2 O (A) _____ modela as interações entre objetos em um sistema com ênfase em *quando* essas interações ocorrem.

a) Diagramas de classe
b) Diagramas de sequência
c) Diagramas de comunicação
d) Diagramas de atividade

12.3 Qual das opções a seguir lista as etapas de um ciclo de vida de software típico em uma ordem sequencial?

a) design, análise, implementação, teste
b) design, análise, teste, implementação
c) análise, design, teste, implementação
d) análise, design, implementação, teste

12.3 Identificando classes em um documento de requisitos

[*Nota:* esta seção pode ser ensinada depois do Capítulo 3.]

Agora iniciamos o desenho do sistema ATM. Nesta seção, identificamos as classes que são necessárias para construir o sistema analisando os *substantivos* e *frases substantivas* que aparecem no documento de requisitos. Introduzimos os diagramas de classe UML para modelar essas classes. Este é um primeiro passo importante na definição da estrutura do sistema.

Identificando as classes em um sistema

Iniciamos nosso processo OOD identificando as classes necessárias para construir o sistema ATM. Por fim, descrevemos essas classes utilizando diagramas de classe UML e implementamos essas classes em Java. Inicialmente, revisamos o documento de requisitos da Seção 12.2 e identificamos substantivos e locuções substantivas para nos ajudar a identificar classes que abrangem o sistema ATM. Podemos decidir que alguns desses substantivos e locuções substantivas são na verdade atributos de outras classes no sistema. Também podemos concluir que alguns substantivos não correspondem a partes do sistema e, portanto, simplesmente não devem ser modelados. As classes adicionais podem tornar-se visíveis à medida que avançamos no processo de design.

A Figura 12.5 lista os substantivos e locuções substantivas encontrados no documento de requisitos da Seção 12.2. Nós os listamos da esquerda para a direita na ordem em que os encontramos inicialmente. Listamos apenas a forma singular de cada um.

Criamos classes apenas para os substantivos e as locuções substantivas que têm importância no sistema ATM. Não modelamos o "banco" como uma classe, porque o banco não é uma parte do sistema ATM — o banco simplesmente quer que o ATM seja construído. "Cliente" e "usuário" também representam entidades exteriores — são importantes porque *interagem* com o nosso sistema ATM, mas não precisamos modelá-los como classes no software ATM. Lembre-se de que modelamos um usuário ATM (isto é, um cliente de banco) como o ator no diagrama de caso de uso da Figura 12.4.

Substantivos e locuções substantivas no documento de requisitos do ATM			
banco	dinheiro / fundos	número de conta	ATM
tela	PIN	usuário	teclado
banco de dados do banco	cliente	dispensador de cédulas (*cash dispenser*)	pesquisa de saldo
transação	cédula de US$ 20 / dinheiro	retirada / saque	conta
entrada de depósito	depósito	saldo	envelope de depósito

Figura 12.5 | Substantivos e locuções substantivas no documento de requisitos do ATM.

Não modelamos a "cédula de US$ 20" nem o "envelope de depósito" como classes. Esses são objetos físicos no mundo real, mas não fazem parte do que é automatizado. Podemos representar adequadamente a presença de contas no sistema utilizando um atributo da classe que modela o dispensador de cédulas (*cash dispenser*). (Damos atributos para as classes de sistema ATM na Seção 12.4.) Por exemplo, o dispensador de cédulas mantém uma contagem do número de contas que ele contém. O documento de requisitos não diz nada sobre o que o sistema deve fazer com os envelopes de depósito depois de recebê-los. Podemos supor que simplesmente reconhecer o recebimento de um envelope — uma operação realizada pela classe que modela a entrada de depósito (*deposit slot*) — é suficiente para representar a presença de um envelope no sistema. (Atribuímos operações às classes de sistema ATM na Seção 12.6.)

Em nosso sistema ATM simplificado, parece mais apropriado representar as várias quantidades de "dinheiro", incluindo o "saldo" de uma conta, como atributos de classes. Da mesma forma, os substantivos "número de conta" e "PIN" representam partes significativas das informações no sistema ATM. Esses são atributos importantes de uma conta bancária. Mas não exibem comportamentos. Portanto, você pode modelá-los mais apropriadamente como atributos de uma classe de conta.

Embora o documento de requisitos costume descrever uma "transação" em um sentido geral, não modelamos a noção ampla de uma transação financeira nesse momento. Em vez disso, modelamos os três tipos de transações (isto é, "consulta de saldo", "saque" e "depósito") como classes individuais. Essas classes possuem atributos específicos necessários para executar as transações que eles representam. Por exemplo, uma retirada precisa saber o valor da retirada. Uma consulta de saldo, porém, não requer nenhum dado adicional além do número da conta. Além disso, as três classes de transação exibem comportamentos únicos. Uma retirada inclui liberar dinheiro para o usuário, enquanto um depósito envolve receber envelopes de depósito do usuário. [*Nota:* na Seção 13.3, "fatoramos" recursos comuns de todas as transações em uma classe de "transação" geral utilizando o conceito de herança orientado a objetos.]

Determinamos as classes para nosso sistema baseado nos substantivos e frases substantivas restantes da Figura 12.5. Cada um desses refere-se a um ou mais dos itens a seguir:

- ATM
- tela
- teclado
- dispensador de cédulas (*cash dispenser*)
- entrada de depósito
- conta
- banco de dados do banco
- pesquisa de saldo
- retirada/saque
- depósito

É provável que os elementos dessa lista sejam as classes necessárias para implementar nosso sistema.

Agora podemos modelar as classes em nosso sistema com base na lista que criamos. Colocamos os nomes de classe no processo de design em letras maiúsculas — uma convenção UML — assim como quando escrevemos o código Java real que implementa nosso design. Se o nome de uma classe contiver mais de uma palavra, unimos as palavras e colocamos a letra inicial de cada palavra em maiúscula (por exemplo, `MultipleWordName`). Utilizando essa convenção, criamos as classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` e `Deposit`. Construímos nosso sistema utilizando essas classes como blocos de construção. Mas, antes de iniciarmos a construção do sistema, devemos obter um melhor entendimento de como as classes se relacionam.

Modelando classes

A UML permite modelar, via **diagramas de classe**, as classes no sistema ATM e seus inter-relacionamentos. A Figura 12.6 representa a classe `ATM`. Cada classe é modelada como um retângulo com três compartimentos. O superior contém o nome da classe centralizada horizontalmente em negrito. O compartimento do meio contém os atributos da classe. (Discutimos os atributos nas seções 12.4 e 12.5.) O compartimento inferior contém as operações da classe (discutidas na Seção 12.6). Na Figura 12.6, os compartimentos do meio e do fundo estão vazios porque ainda não determinamos atributos e operações dessa classe.

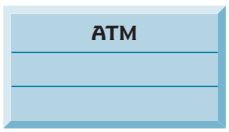


Figura 12.6 | Representando uma classe na UML utilizando um diagrama de classe.

Os diagramas de classe também mostram os relacionamentos entre as classes do sistema. A Figura 12.7 mostra como nossas classes `ATM` e `Withdrawal` se relacionam. Por ora, para simplificar, escolhemos modelar apenas esse subconjunto de classes. Apresentamos um diagrama de classe mais completo mais adiante nesta seção. Observe que os retângulos que representam classes nesse diagrama não estão subdivididos em compartimentos. A UML permite a supressão de atributos e operações de classe dessa maneira para criar diagramas mais legíveis, quando apropriado. Diz-se que esse diagrama é um **diagrama elidido** — em que algumas informações, como o conteúdo do segundo e terceiro compartimentos, não são modeladas. Colocaremos informações sobre esses compartimentos nas seções 12.4 a 12.6.

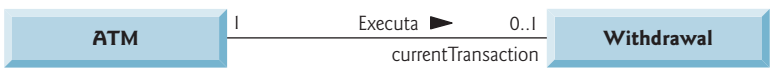


Figura 12.7 | Diagrama de classe que mostra uma associação entre classes.

Na Figura 12.7, a linha sólida que conecta as duas classes representa uma **associação** — um relacionamento entre classes. Os números próximos de cada extremidade da linha são valores de **multiplicidade**, que indicam quantos objetos de cada classe participam da associação. Nesse caso, seguir a linha da esquerda para a direita revela que, a qualquer dado momento, um objeto `ATM` participa de uma associação com zero ou um objeto `Withdrawal` — zero se o usuário atual não estiver realizando atualmente uma transação ou tiver solicitado um tipo de transação diferente e um se o usuário tiver solicitado uma retirada. A UML pode modelar muitos tipos de multiplicidade. A Figura 12.8 lista e explica os tipos de multiplicidade.

Símbolo	Significado
0	Nenhum
1	Um
m	Um valor de inteiro
0..1	Zero ou um
m, *n*	*m* ou *n*
m..*n*	Pelo menos *m*, mas não mais do que *n*
*	Qualquer número inteiro não negativo (zero ou mais)
0..*	Zero ou mais (idêntico a *)
1..*	Um ou mais

Figura 12.8 | Tipos de multiplicidade.

Uma associação pode ser nomeada. Por exemplo, a palavra `Executes` acima da linha que conecta as classes `ATM` e `Withdrawal` na Figura 12.7 indica o nome dessa associação. Essa parte do diagrama exibe "um objeto da classe `ATM` que executa zero ou um objeto da classe `Withdrawal`". Observe que os nomes de associação são direcionais, como indicado pela ponta da seta preenchida — então seria inadequado, por exemplo, ler a associação anterior da direita para a esquerda como "zero ou um objeto de classe `Withdrawal` que executa um objeto de classe `ATM`".

A palavra `currentTransaction` ao lado de `Withdrawal` na linha de associação da Figura 12.7 é um **nome de papel**, que identifica o papel que o objeto `Withdrawal` desempenha em seu relacionamento com o `ATM`. Um nome de papel adiciona significado a uma associação entre classes identificando o papel que uma classe desempenha no contexto de uma associação. Uma classe pode desempenhar diversos papéis no mesmo sistema. Por exemplo, no sistema pessoal de uma escola, uma pessoa pode desempenhar o papel de "professor" ao se relacionar com alunos. A mesma pessoa pode assumir o papel de "colega" ao associar-se com outro professor e de "treinador", ao treinar alunos atletas. Na Figura 12.7, o nome de papel `currentTransaction` indica que o objeto `Withdrawal` que participa da associação `Executes` com um objeto de classe `ATM` representa a transação atual que o ATM processa. Em outros contextos, um objeto `Withdrawal` pode assumir outros papéis (por exemplo, a "transação anterior"). Observe que não especificamos um nome de papel para a extremidade `ATM` da associação `Executes`. Os nomes de papel em diagramas de classe costumam ser omitidos quando o significado de uma associação é claro sem eles.

Além de indicar relacionamentos simples, as associações podem especificar relacionamentos mais complexos, como objetos de uma classe que são compostos de objetos de outras classes. Considere um caixa eletrônico do mundo real. Que "peças" um fabricante monta para construir um ATM funcional? Nosso documento de requisitos informa que o ATM é composto de uma tela, um teclado, um dispensador de cédulas (*cash dispenser*) e uma entrada de depósito (*deposit slot*).

Na Figura 12.9, os **losangos sólidos** anexados às linhas de associação da classe `ATM` indicam que a `ATM` tem um relacionamento de **composição** com as classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`. A composição implica um relacionamento do todo/parte. A classe que tem o símbolo de composição (o losango sólido) em sua extremidade da linha de associação é o todo (nesse caso, `ATM`) e as classes na extremidade das linhas de associação são as partes — nesse caso, `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`. As composições na Figura 12.9 indicam que um objeto da classe `ATM` é formado de um objeto da classe `Screen` , um objeto da classe `CashDispenser`, um objeto da classe `Keypad` e um objeto da classe `DepositSlot`. O ATM *tem uma* tela, um teclado, um dispensador de cédulas e uma entrada de depósito. (Como vimos no Capítulo 9, o relacionamento *é um* define a herança. Veremos na Seção 13.3 que há uma ótima oportunidade de utilizar a herança no projeto de sistema ATM.)

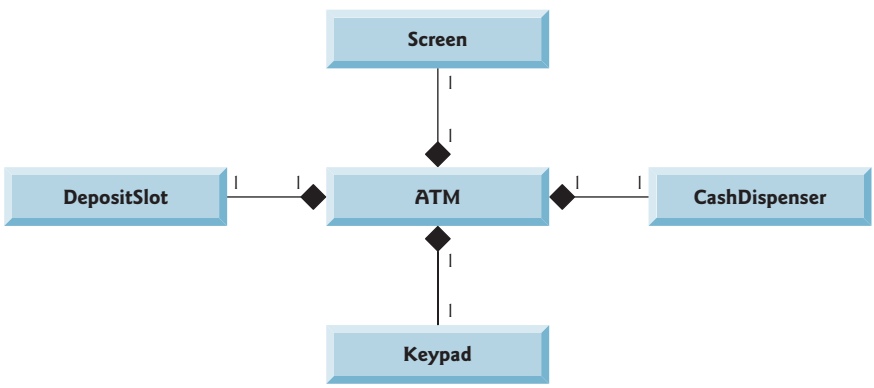


Figura 12.9 | Diagrama de classe mostrando os relacionamentos de composição.

De acordo com a especificação UML (`www.omg.org/technology/documents/formal/uml.htm`), os relacionamentos de composição têm as seguintes propriedades:

1. somente uma classe no relacionamento pode representar o todo (isto é, o losango pode ser colocado somente no final da linha de associação). Por exemplo, a tela é parte do ATM ou o ATM é parte da tela, mas a tela e o ATM não podem representar o todo no relacionamento.
2. as partes no relacionamento de composição só existem enquanto o todo existir, e o todo é responsável pela criação e destruição de suas partes. Por exemplo, o ato de construir um ATM inclui manufaturar suas partes. Além disso, se o ATM é destruído, sua tela, teclado, dispensador de cédulas e entrada de depósito também são destruídos.
3. uma parte pode pertencer a somente um todo de cada vez, embora possa ser removida e anexada a outro todo, que então assume a responsabilidade pela parte.

Os losangos sólidos em nossos diagramas de classe indicam relacionamentos de composição que satisfazem essas propriedades. Se um relacionamento *tem um* não satisfaz um ou mais desses critérios, a UML especifica que os **losangos sem preenchimento** são anexados às extremidades de linhas de associação para indicar **agregação** — uma forma mais fraca de composição. Por exemplo, um computador pessoal e um monitor de computador participam de um relacionamento de agregação — o computador *tem um* monitor, mas as duas partes podem existir independentemente e o mesmo monitor pode ser anexado a múltiplos computadores de uma vez, violando assim a segunda e a terceira propriedades de composição.

A Figura 12.10 mostra um diagrama de classe para o sistema ATM. Esse diagrama modela a maioria das classes que identificamos anteriormente nesta seção, bem como as associações entre elas que podemos inferir do documento de requisitos. [*Nota:* as classes `Balance-`

`Inquiry` e `Deposit` participam de associações semelhantes àquelas da classe `Withdrawal`, então escolhemos omiti-las desse diagrama para mantê-lo simples. Na Seção 13.3, expandimos nosso diagrama de classe para incluir todas as classes no sistema ATM.]

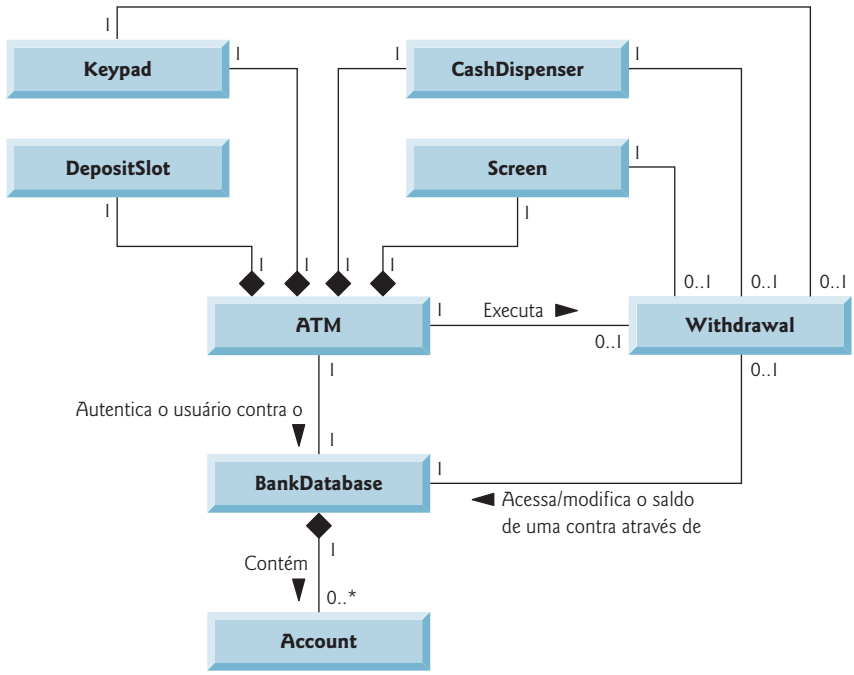


Figura 12.10 | O diagrama de classe para o modelo de sistema ATM.

A Figura 12.10 apresenta um modelo gráfico da estrutura de sistema ATM. Isso inclui as classes `BankDatabase` e `Account` e várias associações que não estavam presentes nas figuras 12.7 ou 12.9. Mostra que a classe `ATM` tem um **relacionamento de um para um** com a classe `BankDatabase` — um objeto `ATM` autentica usuários em um objeto `BankDatabase`. Na Figura 12.10, também modelamos o fato de que o banco de dados do banco contém informações sobre muitas contas — um objeto `BankDatabase` participa de um relacionamento de composição com zero ou mais objetos `Account`. O valor de multiplicidade 0..* na extremidade `Account` da associação entre a classe `BankDatabase` e a classe `Account` indica que zero ou mais objetos de classe `Account` fazem parte da associação. A classe `BankDatabase` tem um **relacionamento de um para muitos** com a classe `Account` — o `BankDatabase` contém muitas `Accounts`. Semelhantemente, a classe `Account` tem um **relacionamento de muitos para um** com a classe `BankDatabase` — pode haver muitas `Accounts` armazenadas no `BankDatabase`. [*Nota:* a partir da Figura 12.8, lembre-se de que o valor de multiplicidade * é idêntico ao 0..*. Incluímos 0..* em nossos diagramas de classe para tornar isso mais claro.]

A Figura 12.10 também indica que a qualquer momento 0 ou 1 objetos `Withdrawal` podem existir. Se o usuário estiver realizando uma retirada, "um objeto da classe `Withdrawal` acessa/modifica o saldo de uma conta por meio de um objeto da classe `BankDatabase`". Podemos ter criado uma associação diretamente entre a classe `Withdrawal` e a classe `Account`. O documento de requisitos, porém, afirma que o "ATM deve interagir com o banco de dados de informações sobre a conta do banco" para realizar transações. Uma conta bancária contém informações sensíveis e os engenheiros de sistemas devem sempre considerar a segurança de dados pessoais ao projetar um sistema. Portanto, somente o `BankDatabase` pode acessar e manipular uma conta diretamente. Todas as outras partes do sistema devem interagir com o banco de dados para recuperar ou atualizar informações da conta (por exemplo, o saldo da conta).

O diagrama de classe na Figura 12.10 também modela associações entre a classe `Withdrawal` e as classes `Screen`, `CashDispenser` e `Keypad`. Uma transação de retirada inclui solicitar ao usuário a escolha de uma quantia em dinheiro a ser retirada e receber a entrada numérica. Essas ações requerem o uso da tela e teclado, respectivamente. Além disso, liberar dinheiro para o usuário requer acesso ao dispensador de cédulas.

As classes `BalanceInquiry` e `Deposit`, embora não mostradas na Figura 12.10, fazem parte de diversas associações com as outras classes do sistema ATM. Semelhante à classe `Withdrawal`, cada uma dessas classes se associa com as classes `ATM` e `BankDatabase`. Um objeto de classe `BalanceInquiry` também se associa com um objeto de classe `Screen` para exibir o saldo de uma conta para o usuário. A classe `Deposit` se associa com as classes `Screen`, `Keypad` e `DepositSlot`. Semelhantes aos saques, as transações de depósito exigem o uso da tela e do teclado para exibir prompts e receber entrada, respectivamente. Para receber envelopes de depósito, um objeto de classe `Deposit` acessa o slot de depósito.

Agora identificamos as classes iniciais no nosso sistema ATM — podemos descobrir outras à medida que avançamos com o design e a implementação. Na Seção 12.4, determinamos os atributos de cada uma dessas classes e, na Seção 12.5, utilizamos esses atributos para examinar como o sistema muda ao longo do tempo.

Exercícios de autorrevisão da Seção 12.3

12.4 Suponha que tivéssemos uma classe `Car` que representasse um carro. Pense em algumas peças diferentes que um fabricante ligaria para montar um carro inteiro. Crie um diagrama de classe (semelhante ao da Figura 12.9) que modela alguns relacionamentos de composição da classe `Car`.

12.5 Suponha que tivéssemos uma classe `File` que representasse um documento eletrônico em um computador independente conectado em rede representado pela classe `Computer`. Que tipo de associação existe entre a classe `Computer` e a classe `File`?

a) A classe `Computer` tem um relacionamento de um para um com a classe `File`.

b) A classe `Computer` tem um relacionamento de muitos para um com a classe `File`.

c) A classe `Computer` tem um relacionamento de um para muitos com a classe `File`.

d) A classe `Computer` tem um relacionamento de muitos para muitos com a classe `File`.

12.6 Determine se a seguinte sentença é *verdadeira* ou *falsa* e, se *falsa*, explique por quê: diz-se que um diagrama UML em que o segundo e o terceiro compartimentos da classe não são modelados é um diagrama elidido.

12.7 Modifique o diagrama de classe da Figura 12.10 para incluir a classe `Deposit` em vez da classe `Withdrawal`.

12.4 Identificando atributos de classe

[*Nota:* esta seção pode ser ensinada depois do Capítulo 4.]

Na Seção 12.3, começamos a primeira etapa de um design orientado a objetos (*Object-Oriented Design* — OOD) para nosso sistema ATM — analisando o documento de requisitos e identificando as classes necessárias para implementar o sistema. Listamos os substantivos e locuções substantivas no documento de requisitos e identificamos uma classe para cada uma que desempenha um papel significativo no sistema ATM. Em seguida, modelamos as classes e seus relacionamentos em um diagrama de classes UML (Figura 12.10).

As classes contêm atributos (dados) e operações (comportamentos). Os atributos de classe são implementados nos programas Java como campos e as operações de classe são implementadas como métodos. Nesta seção, determinamos muitos dos atributos necessários no sistema ATM. Na Seção 12.5, examinaremos como esses atributos representam o estado de um objeto. Na Seção 12.6, determinaremos as operações de classe.

Identificando atributos

Considere os atributos de alguns objetos do mundo real: Os atributos de uma pessoa incluem a altura, peso e se ela é canhota, destra ou ambidestra. Os atributos de um rádio incluem a sua estação, volume e configurações de FM ou AM. Os atributos de um carro incluem as leituras do velocímetro e do hodômetro, a quantidade de gasolina em seu tanque e a marcha em que está. Os atributos de um computador pessoal incluem seu fabricante (por exemplo, Dell, Sun, Apple ou IBM), tipo de tela (por exemplo, LCD ou CRT), tamanho da memória principal e tamanho do disco rígido.

Podemos identificar muitos atributos das classes no nosso sistema procurando palavras e frases descritivas no documento de requisitos. Para cada palavra ou frase encontrada que desempenha um papel significativo no sistema ATM, criamos um atributo e o atribuímos a uma ou mais classes identificadas na Seção 12.3. Também criamos atributos para representar quaisquer dados adicionais de que uma classe talvez precise, à medida que essas necessidades tornam-se claras por todo o processo de design.

A Figura 12.11 lista as palavras ou frases no documento de requisitos que descrevem cada classe. Formamos essa lista lendo o documento de requisitos e identificando todas as palavras ou frases que se referem às características das classes no sistema. Por exemplo, o documento de requisitos descreve os passos seguidos para obter uma "quantia de saque" ("*withdrawal amount*"), assim listamos "*amount*" ao lado da classe `Withdrawal`.

A Figura 12.11 nos leva a criar um atributo da classe `ATM`. A classe `ATM` mantém as informações sobre o estado do ATM. A frase "usuário é autenticado" descreve um estado do ATM (introduzimos estados na Seção 12.5), portanto incluímos `userAuthenticated` como um **atributo booleano** (isto é, um atributo com um valor `true` ou `false`) à classe `ATM`. Observe que o tipo de atributo `Boolean` na UML é equivalente ao tipo `boolean` em Java. Esse atributo indica se o ATM autenticou o usuário atual com sucesso — `userAuthenticated` deve ser `true` para que o sistema autorize o usuário a realizar transações e acessar as informações sobre a conta. Esse atributo ajuda a garantir a segurança dos dados no sistema.

As classes `BalanceInquiry`, `Withdrawal` e `Deposit` compartilham um atributo. Cada transação envolve um "número de conta" que corresponde à conta do usuário que faz a transação. Atribuímos um atributo inteiro `accountNumber` a cada classe de transação para identificar a conta a qual um objeto da classe se aplica.

Palavras e frases descritivas no documento de requisitos também sugerem algumas diferenças nos atributos requeridos por cada classe de transação. O documento de requisitos indica que, para sacar dinheiro ou depositar fundos, os usuários devem inserir uma "quantia" (*amount*) específica a ser sacada ou depositada, respectivamente. Portanto, atribuímos às classes `Withdrawal` e `Deposit` um atributo `amount` para armazenar o valor fornecido pelo usuário. Os valores relacionados a um saque e depósito são características definidoras das transações que o sistema requer para que essas transações aconteçam. A classe `BalanceInquiry`, porém, não precisa de nenhum dado adicional para realizar sua tarefa — ela requer somente um número de conta para indicar a conta cujo saldo deve ser recuperado.

Classe	Palavras e frases descritivas
`ATM`	usuário é autenticado
`BalanceInquiry`	número de conta
`Withdrawal`	número de conta valor
`Deposit`	número de conta valor
`BankDatabase`	*[nenhuma palavra ou frase descritiva]*
`Account`	número de conta PIN saldo
`Screen`	*[nenhuma palavra ou frase descritiva]*
`Keypad`	*[nenhuma palavra ou frase descritiva]*
`CashDispenser`	inicia cada dia carregado com 500 cédulas de US$ 20
`DepositSlot`	*[nenhuma palavra ou frase descritiva]*

Figura 12.11 | Palavras e frases descritivas do documento de requisitos do ATM.

A classe `Account` tem vários atributos. O documento de requisitos declara que cada conta bancária deve ter um "número de conta" e "PIN", que o sistema utiliza para identificar contas e autenticar usuários. Atribuímos à classe `Account` dois atributos de inteiro: `accountNumber` e `pin`. O documento de requisitos também especifica que uma conta deve manter um "saldo" ("*balance*") do valor na conta e que esse valor depositado pelo usuário não se torna disponível para saque até que o banco verifique-o no envelope de depósito e que todos os cheques no envelope sejam compensados. Uma conta, porém, ainda deve registrar o valor que um usuário deposita. Portanto, decidimos que uma conta deve representar um saldo utilizando dois atributos: `availableBalance` e `totalBalance`. O atributo `availableBalance` armazena o valor que um usuário pode sacar da conta. O atributo `totalBalance` refere-se ao valor total que o usuário tem "em depósito" (isto é, o valor disponível mais o valor que está esperando para ser verificado ou compensado). Por exemplo, suponha que um usuário do ATM deposite US$ 50,00 em uma conta zerada. O atributo `totalBalance` aumentaria para US$ 50,00 a fim de registrar o depósito, mas o `availableBalance` permaneceria em US$ 0.

[*Nota*: assumimos que o banco atualiza o atributo `availableBalance` de uma `Account` depois que a transação no ATM ocorre, em resposta à confirmação de que US$ 50 ou cheques foram encontrados no envelope de depósito. Assumimos que essa atualização ocorre por meio de uma transação que um funcionário do banco realiza utilizando algum software do banco diferente do ATM. Portanto, não discutimos essa transação no nosso estudo de caso.]

A classe `CashDispenser` tem um atributo. O documento de requisitos afirma que o dispensador de cédulas "começa todos os dias carregado com 500 cédulas de US$ 20". O dispensador de cédulas deve manter um registro do número de cédulas que ele contém para determinar se há dinheiro suficiente à disposição para satisfazer as solicitações de saque. Atribuímos à classe `CashDispenser` um atributo inteiro `count`, inicialmente configurado como `500`.

Para problemas reais na indústria, não há garantias de que os documentos de requisitos serão precisos o suficiente para que o projetista de sistemas orientados a objetos determine todos os atributos ou mesmo todas as classes. A necessidade de classes, atributos e comportamentos adicionais pode tornar-se clara à medida que o design avança. À medida que progredimos nesse estudo de caso, continuaremos a adicionar, modificar e excluir as informações sobre as classes no nosso sistema.

Modelando atributos

O diagrama de classes na Figura 12.12 lista alguns atributos das classes no nosso sistema — as palavras e frases descritivas na Figura 12.11 nos levam a identificar esses atributos. Por uma questão de simplicidade, a Figura 12.12 não mostra as associações entre classes — nós as mostramos na Figura 12.10. Essa é uma prática comum de projetistas de sistemas quando os projetos estão em desenvolvimento. A partir da Seção 12.3, lembre-se de que, na UML, os atributos de uma classe são colocados no compartimento central do retângulo da classe. Listamos cada nome e tipo do atributo separado por um dois-pontos (`:`), seguido, em alguns casos, por um sinal de igual (`=`) e de um valor inicial.

Considere o atributo `userAuthenticated` da classe `ATM`:

```
userAuthenticated : Boolean = false
```

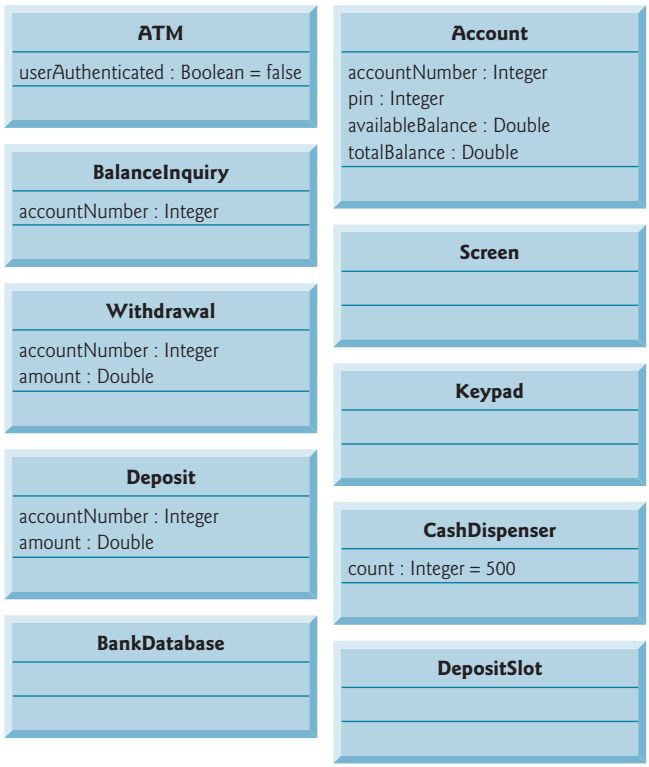


Figura 12.12 | Classes com atributos.

Essa declaração de atributo contém três informações sobre o atributo. O **nome do atributo** é `userAuthenticated`. O **tipo do atributo** é `Boolean`. No Java, um atributo pode ser representado por um tipo primitivo, como `boolean`, `int` ou `double`, ou um tipo de referência como uma classe. Escolhemos modelar somente atributos de tipo primitivo na Figura 12.12 — discutimos o raciocínio por trás dessa decisão a seguir. [*Nota:* os tipos de atributos na Figura 12.12 estão na notação UML. Associaremos os tipos `Boolean`, `Integer` e `Double` no diagrama UML com os tipos primitivos `boolean`, `int` e `double` em Java, respectivamente.]

Também podemos indicar um valor inicial para um atributo. O atributo `userAuthenticated` na classe `ATM` tem um valor inicial de `false`. Isso indica que o sistema inicialmente não considera o usuário como autenticado. Se um atributo não contiver nenhum valor inicial especificado, somente seu nome e tipo (separado por um dois-pontos) são mostrados. Por exemplo, o atributo `accountNumber` da classe `BalanceInquiry` é um inteiro. Aqui, não mostramos nenhum valor inicial, pois o valor desse atributo é um número que ainda não conhecemos. Esse número será determinado em tempo de execução com base no número de conta inserido pelo usuário atual do ATM.

A Figura 12.12 não inclui atributos para as classes `Screen`, `Keypad` e `DepositSlot`. Essas são componentes importantes do nosso sistema, para os quais o nosso processo de design ainda não revelou nenhum atributo. Entretanto, é possível descobrir alguns nas fases restantes do design ou ao implementarmos essas classes em Java. Isso é perfeitamente normal.

Observação de engenharia de software 12.1

Nas etapas iniciais do processo de design, frequentemente faltam nas classes atributos (e operações). Essas classes, porém, não devem ser eliminadas, pois os atributos (e operações) podem tornar-se evidentes nas fases posteriores do design e implementação.

Observe que a Figura 12.12 também não inclui nenhum atributo para a classe `BankDatabase`. Lembre-se de que os atributos no Java podem ser representados por tipos primitivos ou tipos de referência. Optamos por incluir somente os atributos de tipo primitivo no diagrama de classes na Figura 12.12 (e nos diagramas de classes semelhantes por todo o estudo de caso). Um atributo de tipo por referência é modelado mais claramente como uma associação entre a classe que contém a referência e a classe do objeto para o qual a referência aponta. Por exemplo, o diagrama de classes na Figura 12.10 indica que a classe `BankDatabase` participa de um relacionamento de composição com zero ou mais objetos `Account`. A partir dessa composição, podemos determinar que, ao implementarmos o sistema ATM em Java, será necessário criar um atributo da classe `BankDatabase` para armazenar referências a zero ou mais objetos de `Account`. De maneira semelhante, podemos determinar os atributos de tipo por referência da classe `ATM` que correspondem aos seus relacionamentos de composição com as classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`. Esses atributos baseados em composição seriam redundantes se modelados na Figura 12.12, porque as composições modeladas na Figura 12.10 já comunicam o fato de que o banco de dados contém as informações sobre zero ou mais contas e que um ATM é composto de uma tela, teclado, dispensador de cédulas e uma abertura para depósito. Em geral,

os desenvolvedores de software modelam esses relacionamentos integrais/parciais como composições em vez de como atributos requeridos para implementar os relacionamentos.

O diagrama de classes na Figura 12.12 fornece uma base sólida para a estrutura do nosso modelo, mas o diagrama não está completo. Na Seção 12.5, identificamos os estados e atividades dos objetos no modelo e, na Seção 12.6, identificamos as operações que os objetos realizam. À medida que apresentamos outras informações sobre a UML e o design orientado a objetos, continuaremos a fortalecer a estrutura do nosso modelo.

Exercícios de autorrevisão da Seção 12.4

12.8 Em geral identificamos os atributos das classes no nosso sistema analisando os (as) _____ no documento de requisitos.
a) substantivos simples e substantivos compostos
b) palavras e frases descritivas
c) verbos e frases com verbos
d) Todos os acima.

12.9 Qual dos seguintes itens não é um atributo de um avião?
a) comprimento
b) envergadura da asa
c) voo
d) número de poltronas

12.10 Descreva o significado da seguinte declaração de atributo da classe `CashDispenser` no diagrama de classes da Figura 12.12:
`count : Integer = 500`

12.5 Identificando estados e atividades dos objetos

[*Nota:* esta seção pode ser ensinada depois do Capítulo 5.]

Na Seção 12.4, identificamos muitos dos atributos de classe necessários para implementar o sistema ATM e os adicionamos ao diagrama de classes na Figura 12.12. Agora mostramos como esses atributos representam o estado de um objeto. Identificamos alguns estados-chave que nossos objetos podem ocupar e discutimos como os objetos mudam de estado em resposta a vários eventos que ocorrem no sistema. Também discutimos o fluxo de trabalho, ou **atividades**, que os objetos realizam no sistema ATM, e apresentamos as atividades dos objetos de transação `BalanceInquiry` e `Withdrawal`.

Diagramas de estado de máquina

Todo objeto em um sistema passa por uma série de estados. O estado de um objeto é indicado pelos valores dos seus atributos em determinado momento. Os **diagramas de estado de máquina** (comumente chamados de **diagramas de estado**) modelam vários estados de um objeto e mostram sob que circunstâncias o objeto muda de estado. Ao contrário dos diagramas de classes apresentados nas seções anteriores do estudo de caso, que focalizaram principalmente a estrutura do sistema, os diagramas de estado modelam algum comportamento do sistema.

A Figura 12.13 é um diagrama de estado simples que modela alguns estados de um objeto de classe `ATM`. A UML representa cada estado em um diagrama de estado como um **retângulo arredondado** com o nome do estado posicionado dentro dele. Um **círculo sólido** com uma seta (→) designa o **estado inicial**. Lembre-se de que modelamos essas informações de estado como o atributo `Boolean userAuthenticated` no diagrama de classes da Figura 12.12. Esse atributo é inicializado como `false`, ou o estado "User not authenticated", de acordo com o diagrama de estado.

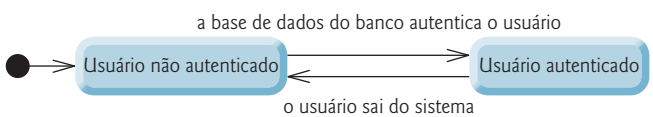


Figura 12.13 | Diagrama de estado do objeto `ATM`.

As setas indicam **transições** entre estados. Um objeto pode transitar de um estado para outro em resposta a vários eventos que ocorrem no sistema. O nome ou descrição do evento que causa uma transição é escrito perto da linha que corresponde à transição. Por exemplo, o objeto `ATM` muda do estado "Usuário não autenticado" para "Usuário autenticado" depois que o banco de dados autentica o usuário. A partir do documento de requisitos, lembre-se de que o banco de dados autentica um usuário comparando o número de conta e PIN inseridos pelo usuário com os de uma conta no banco de dados. Se o usuário inseriu um número de conta válida e o PIN correto, o objeto `ATM` transita

para o estado "Usuário autenticado" e muda seu atributo `userAuthenticated` para um valor de `true`. Quando o usuário sair do sistema escolhendo a opção "exit" (sair) do menu principal, o objeto `ATM` retorna ao estado "Usuário não autenticado".

Observação de engenharia de software 12.2

Em geral, os engenheiros de software não criam diagramas de estado que mostram cada estado possível e transição de estado para todos os atributos — há simplesmente muitos deles. Os diagramas de estado em geral mostram apenas estados-chave e as principais transições de estado.

Diagramas de atividade

Como um diagrama de estado, um **diagrama de atividade** modela aspectos do comportamento do sistema. Ao contrário de um diagrama de estado, um diagrama de atividade modela o **fluxo de trabalho** de um objeto (sequência de eventos) durante a execução de programa. Um diagrama de atividade modela as **ações** que o objeto realizará e em que ordem. O diagrama de atividade na Figura 12.14 modela as ações envolvidas na execução de uma transação de consulta de saldo. Assumimos que um objeto `BalanceInquiry` já foi inicializado e recebeu um número de conta válida (aquela do usuário atual), então o objeto sabe que saldo recuperar. O diagrama inclui as ações que ocorrem depois de o usuário selecionar uma consulta de saldo do menu principal e antes de o ATM retornar o usuário para o menu principal — um objeto `BalanceInquiry` não realiza nem inicia essas ações, então não as modelamos aqui. O diagrama inicia com a recuperação do saldo da conta do banco de dados. Em seguida, o `BalanceInquiry` exibe o saldo na tela. Essa ação completa a execução da transação. Lembre-se de que escolhemos representar o saldo de uma conta tanto com o atributo `availableBalance` como com `totalBalance` da classe `Account`, então as ações modeladas na Figura 12.14 referem-se à recuperação e exibição dos *dois* atributos de saldo.

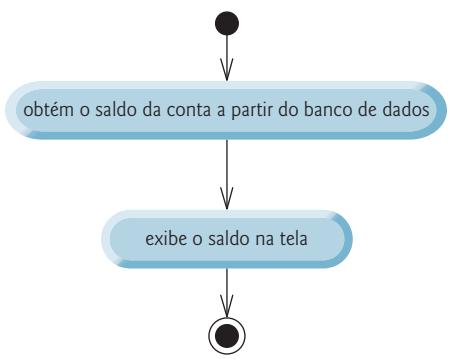


Figura 12.14 | Diagrama de atividades de um objeto `BalanceInquiry`.

A UML representa uma ação em um diagrama de atividade como um estado de ação modelado por um retângulo com seus lados esquerdo e direito substituídos por arcos que se curvam para fora. Cada estado de ação contém uma expressão de ação — por exemplo, "obtém o saldo da conta a partir do banco de dados" — que especifica uma ação a ser realizada. Uma seta (—>) conecta dois estados de ação, indicando a ordem em que ocorrem as ações representadas pelos estados de ação. O círculo sólido (na parte superior da Figura 12.14) representa o estado inicial da atividade — o começo do fluxo de trabalho antes de o objeto realizar as ações modeladas. Nesse caso, a transação executa primeiro a expressão de ação "obtém o saldo da conta a partir do banco de dados". A transação então exibe ambos os saldos na tela. O círculo sólido dentro de um círculo vazado (na parte inferior da Figura 12.14) representa o estado final — o fim do fluxo de trabalho depois de o objeto realizar as ações modeladas. Utilizamos os diagramas de atividade UML para ilustrar o fluxo de controle das instruções de controle apresentadas nos capítulos 4 e 5.

A Figura 12.15 mostra um diagrama de atividade de uma transação de saque. Assumimos que um número de conta válida foi atribuído ao objeto `Withdrawal`. Não modelamos o usuário que seleciona a opção de saque do menu principal ou a ATM que retorna o usuário para o menu principal porque essas não são ações realizadas por um objeto `Withdrawal`. A transação exibe primeiro um menu de quantias padrão de saque (mostradas na Figura 12.3) e uma opção para cancelar a transação. A transação então recebe uma seleção de menu do usuário. O fluxo de atividade agora chega a uma decisão (uma bifurcação indicada pelo símbolo de um pequeno losango). [*Nota:* uma decisão foi conhecida como uma variação nas primeiras versões da UML.] Esse ponto determina a próxima ação com base na condição de guarda associada (em colchetes perto da transição), que declara que a transição ocorre se essa condição de guarda for satisfeita. Se o usuário cancelar a transação escolhendo a opção "cancel" do menu, o fluxo de atividade pula imediatamente para o estado final. Observe a mescla (indicada pelo símbolo do pequeno losango) em que o fluxo de cancelamento de atividade liga-se ao fluxo principal de atividade antes de alcançar o estado final da atividade. Se o usuário selecionar uma quantia de saque do menu, `Withdrawal` configura `amount` (um atributo modelado originalmente na Figura 12.12) como o valor escolhido pelo usuário.

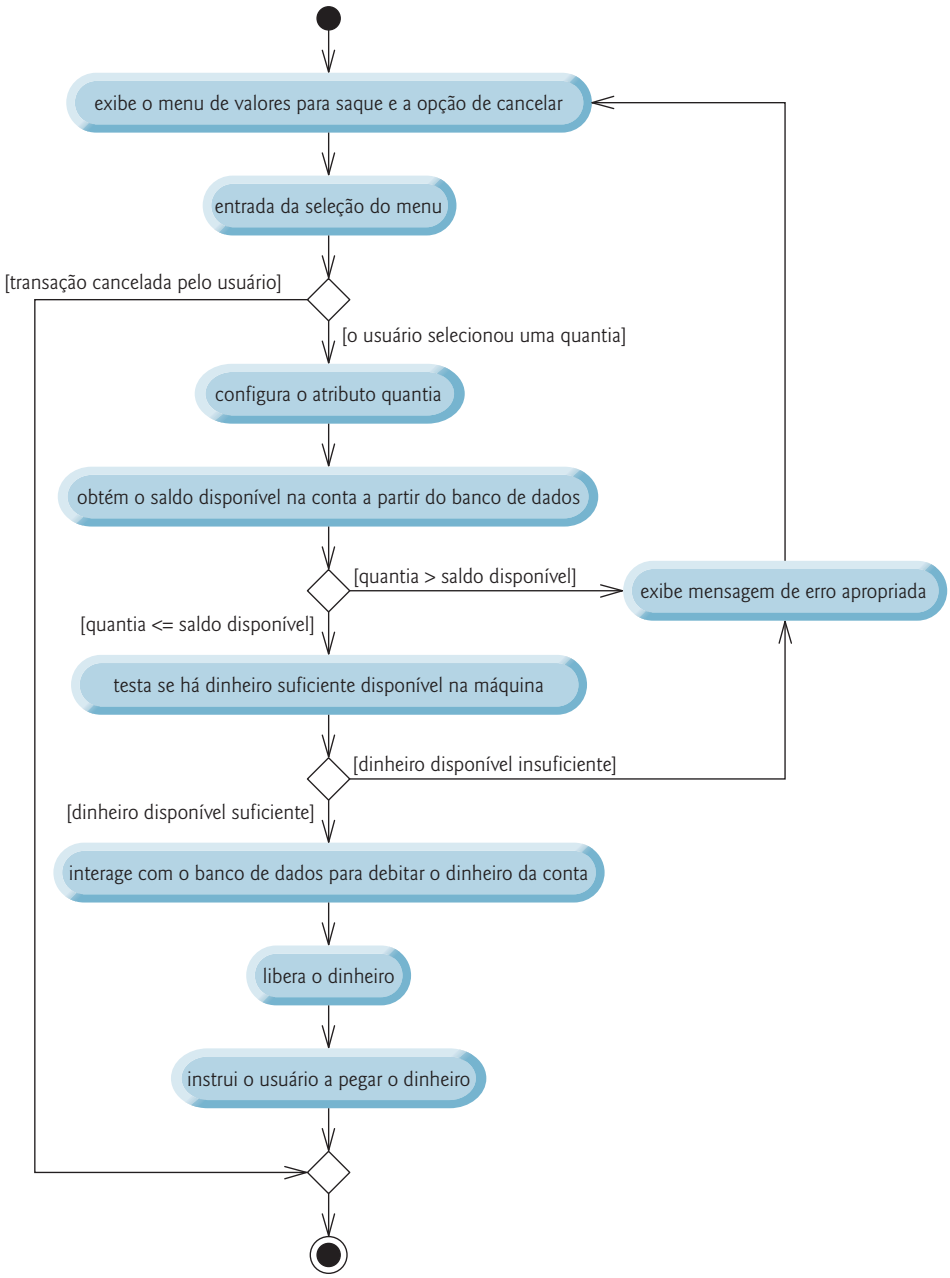


Figura 12.15 | Diagrama de atividades para uma transação de retirada.

Depois de configurar a quantidade de retirada ou saque, a transação recupera o saldo disponível da conta de usuário (isto é, o atributo `availableBalance` do objeto `Account` do usuário) a partir do banco de dados. O fluxo de atividade então chega a outra decisão. Se a quantidade de saque solicitada exceder o saldo disponível do usuário, o sistema exibe uma mensagem de erro apropriada informando o problema ao usuário, então retorna ao começo do diagrama de atividade e pede para o usuário inserir uma nova quantidade. Se a quantia retirada solicitada for menor que ou igual ao saldo disponível do usuário, a transação continua. A transação seguinte testa se o dispensador de cédulas tem dinheiro suficiente para atender a solicitação de retirada. Se não tiver, a transação exibe uma mensagem de erro apropriada, depois retorna ao começo do diagrama de atividade e solicita ao usuário para escolher uma nova quantidade. Se houver dinheiro suficiente disponível, a transação interage com o banco de dados para debitar o valor do saque da conta do usuário (isto é, subtrair a quantia tanto do atributo `availableBalance` como do `totalBalance` do objeto `Account` do usuário). A transação então dispensa a quantia desejada de dinheiro e instrui o usuário a pegá-la. Por fim, o fluxo principal de atividade mescla-se com o fluxo de cancelamento de atividade antes de alcançar o estado final.

Demos os primeiros passos na modelagem do comportamento do sistema de software ATM e mostramos como os atributos de um objeto participam da realização das atividades do objeto. Na Seção 12.6, investigamos os comportamentos de todas as classes para dar uma interpretação mais exata do comportamento do sistema preenchendo os terceiros compartimentos das classes no nosso diagrama de classe.

Exercícios de autorrevisão da Seção 12.5

12.11 Determine se a seguinte sentença é *verdadeira* ou *falsa* e, se *falsa*, explique por quê: diagramas de estado modelam aspectos estruturais de um sistema.

12.12 Um diagrama de atividades modela os (as)_____ que um objeto realiza e a ordem em que ele os (as) realiza.

a) ações
b) atributos
c) estados
d) transições de estado

12.13 Com base no documento de requisitos, crie um diagrama de atividades para uma transação de depósito.

12.6 Identificando operações de classe

[Nota: esta seção pode ser ensinada depois do Capítulo 6.]

Nesta seção, determinamos algumas operações de classe (ou comportamentos) necessárias para implementar o sistema ATM. Uma operação é um serviço que os objetos de uma classe fornecem aos clientes (usuários) da classe. Pense nas operações de alguns objetos do mundo real. As operações de um rádio incluem configurar sua estação e volume (em geral invocado por uma pessoa que ajusta os controles do rádio). As operações de um carro incluem aceleração (invocado pelo motorista ao pressionar o pedal do acelerador), desaceleração (invocado pelo motorista ao pressionar o pedal do freio ou soltar o pedal do acelerador), mudança de direção e troca de marchas. Os objetos de software também podem oferecer operações — por exemplo, um objeto de um software gráfico poderia oferecer operações para desenhar um círculo, uma linha, um quadrado etc. Um objeto de um software de planilha poderia oferecer operações como imprimir a planilha, somar os elementos em uma linha ou coluna e diagramar informações na planilha como um gráfico de barras ou gráfico de torta.

Podemos derivar várias operações da classe examinando os verbos e frases com verbos-chave no documento de requisitos. Então relacionamos esses verbos e frases verbais a classes no nosso sistema (Figura 12.16). As frases com verbos na Figura 12.16 ajudam a determinar as operações de cada classe.

Classe	Verbos e frases com verbos
`ATM`	executar transações financeiras
`BalanceInquiry`	*[nenhuma no documento de requisitos]*
`Withdrawal`	*[nenhuma no documento de requisitos]*
`Deposit`	*[nenhuma no documento de requisitos]*
`BankDatabase`	autentica um usuário, recupera um saldo em conta, credita uma quantia depositada em uma conta, debita um valor sacado de uma conta
`Account`	recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia sacada de uma conta
`Screen`	exibe uma mensagem para o usuário
`Keypad`	recebe entrada numérica do usuário
`CashDispenser`	fornece o dinheiro, indica se contém dinheiro suficiente para satisfazer uma solicitação de saque
`DepositSlot`	recebe um envelope de depósito

Figura 12.16 | Os verbos e frases com verbo para cada classe no sistema ATM.

Modelando operações

Para identificar operações, examinamos as frases verbais listadas sobre cada classe na Figura 12.16. A frase "executa transações financeiras" associada com a classe `ATM` implica que a classe `ATM` instrui as transações a executar. Portanto, as classes `BalanceInquiry`, `Withdrawal` e `Deposit` precisam de uma operação para fornecer esse serviço ao ATM. Colocamos essa operação (que nomeamos `execute`) no terceiro compartimento das três classes de transação no diagrama de classe atualizado da Figura 12.17. Durante uma sessão ATM, o objeto `ATM` invocará essas operações de transação de acordo com a necessidade.

A UML representa operações (implementadas como métodos em Java) listando o nome da operação, seguido por uma lista separada por vírgulas de parâmetros entre parênteses, um dois-pontos e o tipo de retorno:

nomeDaOperação(*parâmetro1*, *parâmetro2*, ..., *parâmetroN*) : *tipo de retorno*

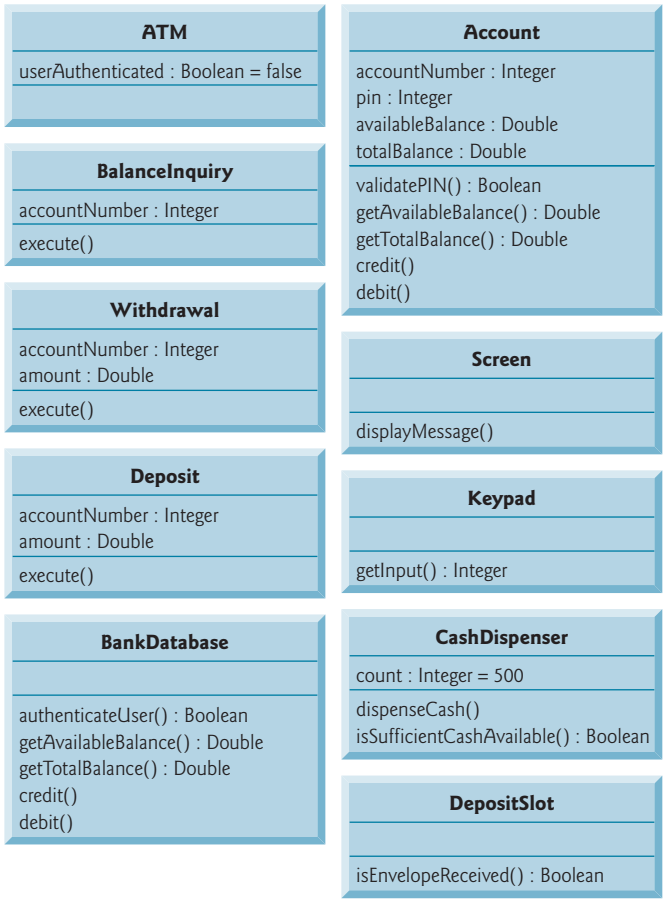


Figura 12.17 | Classes no sistema ATM com atributos e operações.

Cada parâmetro na lista separada por vírgulas de parâmetros consiste em um nome de parâmetro, seguido por um dois-pontos e o tipo de parâmetro:

nomeDoParâmetro : *tipoDoParâmetro*

Por enquanto, não listamos os parâmetros das nossas operações — identificaremos e modelaremos alguns deles a seguir. Para algumas operações, ainda não conhecemos os tipos de retorno, portanto também iremos omitimo-los do diagrama. Essas omissões são perfeitamente normais nesse ponto. À medida que o nosso design e implementação avançam, adicionaremos os tipos de retorno remanescentes.

Autenticando um usuário

A Figura 12.16 lista a frase "autentica um usuário" ao lado da classe `BankDatabase` — o banco de dados é o objeto que contém informações sobre uma conta necessárias para determinar se o número da conta e PIN inseridos por um usuário correspondem àqueles de uma conta mantida pelo banco. Portanto, a classe `BankDatabase` precisa de uma operação que forneça um serviço de autenticação ao ATM. Colocamos a operação `authenticateUser` no terceiro compartimento da classe `BankDatabase` (Figura 12.17). Entretanto, um objeto da classe `Account`, não da classe `BankDatabase`, armazena o número da conta e o PIN que devem ser acessados para autenticar um usuário, dessa forma a classe `Account` deve fornecer um serviço para validar um PIN, obtido por meio da entrada do usuário, contra um PIN armazenado em um objeto `Account`. Portanto, adicionamos uma operação `validatePIN` à classe `Account`. Note que especificamos um tipo de retorno de `Boolean` para as operações `authenticateUser` e `validatePIN`. Cada operação retorna um valor indicando que a operação foi bem-sucedida ao realizar sua tarefa (isto é, um valor de retorno de `true`) ou não (isto é, um valor de retorno de `false`).

Outras operações `BankDatabase` e `Account`

A Figura 12.16 lista várias frases com verbos adicionais da classe `BankDatabase`: "recupera um saldo de conta", "credita uma quantia depositada em uma conta" e "debita um valor sacado de uma conta". Como ocorre com "autentica um usuário", essas frases restantes se referem aos serviços que o banco de dados deve fornecer ao ATM, porque o banco de dados armazena todos os dados de uma conta utilizados para autenticar um usuário e realizar as transações no ATM. Entretanto, objetos da classe `Account` na verdade realizam as operações às quais essas frases se referem. Portanto, atribuímos uma operação à classe `BankDatabase` e à classe `Account` para que elas correspondam

a cada uma dessas frases. Lembre-se a partir do que foi discutido na Seção 12.3 de que, como a conta bancária contém informações sigilosas, não permitimos que o ATM acesse contas diretamente. O banco de dados atua como um intermediário entre o ATM e os dados da conta, evitando assim acesso não autorizado. Como veremos na Seção 12.7, a classe `ATM` invoca as operações da classe `BankDatabase`, cada uma das quais por sua vez invoca a operação com o mesmo nome na classe `Account`.

Obtendo os saldos

A frase "recupera o saldo de uma conta" sugere que as classes `BankDatabase` e `Account` precisam de uma operação `getBalance`. Contudo, lembre-se de que criamos dois atributos na classe `Account` para representar um saldo — `availableBalance` e `totalBalance`. Uma consulta de saldo requer acesso aos dois atributos de saldo para poder exibi-los ao usuário, mas um saque precisa verificar somente o valor de `availableBalance`. Para permitir que objetos no sistema obtenham cada atributo de saldo individualmente, adicionamos operações `getAvailableBalance` e `getTotalBalance` ao terceiro compartimento das classes `BankDatabase` e `Account` (Figura 12.17). Especificamos um tipo de retorno de `Double` para essas operações porque os atributos de saldo que elas recuperam são do tipo `Double`.

Fazendo créditos e débitos em uma `Account`

As frases "credita uma quantia depositada em uma conta" e "debita um valor sacado de uma conta" indicam que as classes `BankDatabase` e `Account` devem realizar operações para atualizar uma conta durante um depósito e saque, respectivamente. Portanto, atribuímos as operações `credit` e `debit` às classes `BankDatabase` e `Account`. Lembre-se de que creditar a uma conta (como em um depósito) só adiciona uma quantia monetária ao atributo `totalBalance`. Debitar de uma conta (como em um saque), por outro lado, subtrai a quantia dos dois atributos de saldo. Ocultamos esses detalhes de implementação dentro da classe `Account`. Esse é um bom exemplo do encapsulamento e ocultamento de informações.

Confirmações de depósito feitas por outro sistema bancário

Se isso fosse um sistema ATM real, as classes `BankDatabase` e `Account` também forneceriam um conjunto de operações para permitir que um outro sistema de operações bancárias atualizasse um saldo na conta do usuário depois de uma confirmação ou rejeição de todo ou parte de um depósito. A operação `confirmDepositAmount`, por exemplo, adicionaria uma quantia monetária ao atributo `availableBalance`, tornando assim os fundos depositados disponíveis para saque. A operação `rejectDepositAmount` subtrairia uma quantia monetária do atributo `totalBalance` para indicar que um valor monetário especificado, recém-depositado por meio do ATM e adicionado ao `totalBalance`, não foi encontrado no envelope de depósito. O banco invocaria essa operação depois de determinar que o usuário não incluiu a quantia monetária correta ou que um cheque não foi compensado (isto é, ele "voltou"). Adicionar essas operações torna nosso sistema mais completo, porém não as incluímos nos nossos diagramas de classe nem na nossa implementação porque elas estão além do escopo desse estudo de caso.

Exibição de mensagens

A classe `Screen` "exibe uma mensagem para o usuário" em vários momentos de uma sessão no ATM. Toda a saída visual ocorre por meio da tela do ATM. O documento de requisitos descreve muitos tipos de mensagens (por exemplo, uma mensagem de boas-vindas, uma mensagem de erro, uma mensagem de agradecimento) que a tela exibe para o usuário. O documento de requisitos também indica que a tela exibe prompts e menus para o usuário. Entretanto, um prompt na verdade é apenas uma mensagem que descreve o que o usuário deve inserir em seguida e um menu é essencialmente um tipo de prompt que consiste em uma série de mensagens (isto é, opções de menu) exibida consecutivamente. Portanto, em vez de atribuir a classe `Screen` a uma operação individual a fim de exibir cada tipo de mensagem, prompt e menu, simplesmente criamos uma operação que possa exibir qualquer mensagem especificada por um parâmetro. Colocamos essa operação (`displayMessage`) no terceiro compartimento da classe `Screen` no nosso diagrama de classe (Figura 12.17). Observe que, nesse momento, não nos preocupamos com o parâmetro dessa operação — ele será modelado mais tarde nesta seção.

Entrada de teclado

Na frase "recebe entrada numérica do usuário" listada pela classe `Keypad` na Figura 12.16, concluímos que a classe `Keypad` deve realizar uma operação `getInput`. Como o teclado do ATM, diferentemente de um teclado de computador, contém somente os números de 0 a 9, especificamos que essa operação retorna um valor inteiro. Lembre-se de que no documento de requisitos, em diferentes situações, talvez seja necessário que o usuário insira um tipo diferente de número (por exemplo, um número de conta, um PIN, o número de uma opção de menu, uma quantia de depósito como um número de centavos). A classe `Keypad` simplesmente obtém um valor numérico para um cliente da classe — ela não determina se o valor atende quaisquer critérios específicos. Qualquer classe que utilize essa operação deve verificar se o usuário inseriu um número apropriado em uma dada situação e então responder de maneira correspondente (isto é, exibir uma mensagem de erro via classe `Screen`). [*Nota:* quando implementamos o sistema, simulamos o teclado do ATM com um teclado de computador e, por simplicidade, supomos que o usuário não irá inserir uma entrada não numérica utilizando teclas do teclado de computador que não aparecem no teclado do ATM.]

Liberação do dinheiro

A Figura 12.16 lista "liberar dinheiro" para a classe `CashDispenser`. Portanto, criamos a operação `dispenseCash` e a listamos abaixo da classe `CashDispenser` na Figura 12.17. A classe `CashDispenser` também "indica se ela contém dinheiro suficiente para satisfazer uma solicitação de retirada". Portanto, incluímos `isSufficientCashAvailable`, uma operação que retorna um valor do tipo `Boolean`, da UML, na classe `CashDispenser`.

A Figura 12.16 também lista "recebe um envelope de depósito" para a classe `DepositSlot`. A fenda para depósito deve indicar se recebeu um envelope, portanto colocamos uma operação `isEnvelopeReceived`, que retorna um valor `Boolean`, no terceiro compartimento da classe `DepositSlot`. [*Nota:* um hardware real da fenda para depósito provavelmente enviaria um sinal ao ATM para indicar que um envelope foi recebido. Entretanto, simulamos esse comportamento com uma operação na classe `DepositSlot` para que a classe `ATM` possa ser invocada a fim de descobrir se a fenda para depósito recebeu um envelope.]

Classe `ATM`

Não listamos nenhuma operação para a classe `ATM` nesse momento. Ainda não estamos cientes de nenhum serviço que a classe `ATM` fornece para outras classes no sistema. Entretanto, ao implementarmos o sistema com código Java, operações dessa classe, e operações adicionais das outras classes no sistema, podem surgir.

Identificando e modelando parâmetros de operação para a classe `BankDatabase`

Até esse momento, não nos preocupamos com os parâmetros das nossas operações — tentamos adquirir apenas um entendimento básico sobre as operações de cada classe. Agora, vamos examinar mais detalhadamente alguns parâmetros de operação. Identificamos os parâmetros de uma operação examinando quais dados a operação requer para realizar sua tarefa atribuída.

Considere a operação `authenticateUser` de `BankDatabase`. Para autenticar um usuário, essa operação deve conhecer o número de conta e o PIN fornecido pelo usuário. Então especificamos que `authenticateUser` aceita os parâmetros do tipo inteiro `userAccountNumber` e `userPIN`, que a operação deve comparar com o número de conta de um objeto `Account` e PIN no banco de dados. Prefixamos esses nomes de parâmetro com o "usuário" para evitar confusão entre os nomes de parâmetro da operação e os nomes de atributo da classe `Account`. Listamos esses parâmetros no diagrama de classe na Figura 12.18 que modela somente a classe `BankDatabase`. [*Nota:* é perfeitamente normal modelar apenas uma classe. Nesse caso, estamos examinando os parâmetros dessa classe, portanto omitimos outras classes. Nos diagramas de classe posteriores nesse estudo de caso, em que parâmetros não são mais o foco, omitimos esses parâmetros para economizar espaço. Mas não se esqueça de que as operações listadas nesses diagramas ainda contêm parâmetros.]

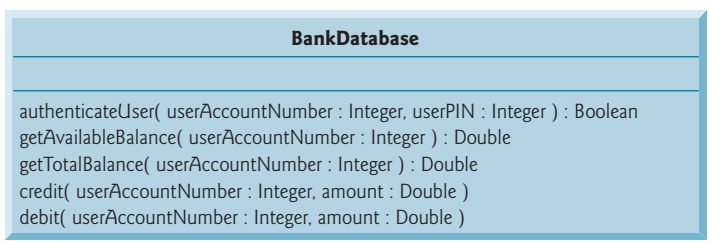


Figura 12.18 | A classe `BankDatabase` com parâmetros de operação.

Lembre-se de que a UML modela cada parâmetro em uma lista de parâmetros separada por vírgulas da operação e listando o nome do parâmetro seguido por um caractere de dois-pontos e pelo tipo de parâmetro (na notação da UML). A Figura 12.18 assim especifica que a operação `authenticateUser` aceita dois parâmetros — `userAccountNumber` e `userPIN`, ambos do tipo `Integer`. Quando implementamos o sistema em Java, representaremos esses parâmetros com os valores `int`.

As operações `getAvailableBalance`, `getTotalBalance`, `credit` e `debit` da classe `BankDatabase` também requerem, cada uma, um parâmetro `userAccountNumber` para identificar a conta à qual o banco de dados deve aplicar as operações, desse modo incluímos esses parâmetros no diagrama de classe da Figura 12.18. Além disso, as operações `credit` e `debit` requerem individualmente um parâmetro `Double` em `amount` para especificar a quantia em dinheiro a ser creditada ou debitada, respectivamente.

Identificando e modelando parâmetros de operação para a classe `Account`

A Figura 12.19 modela os parâmetros de operação da classe `Account`. A operação `validatePIN` requer somente um parâmetro `userPIN`, que contém o PIN especificado pelo usuário a ser comparado com o PIN da conta. Como ocorre com suas contrapartes na classe `BankDatabase`, as operações `credit` e `debit` na classe `Account` requerem um parâmetro `Double` em `amount` que indica a quantia monetária envolvida na operação. As operações `getAvailableBalance` e `getTotalBalance` na classe `Account` não requerem nenhum dado adicional para realizar suas tarefas. Observe que as operações da classe `Account` não requerem um parâmetro de número de conta para distinguir entre `Accounts`, porque essas operações só podem ser invocadas em um objeto `Account` específico.

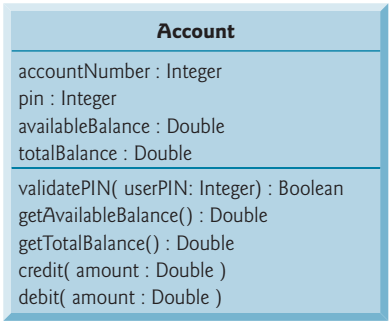


Figura 12.19 | A classe `Account` com parâmetros de operação.

Identificando e modelando parâmetros de operação para a classe `Screen`

A Figura 12.20 modela a classe `Screen` com um parâmetro especificado para a operação `displayMessage`. Essa operação requer somente um parâmetro `String` `message` que indica o texto a ser exibido. Lembre-se de que os tipos de parâmetros listados nos nossos diagramas de classe estão em notação UML, portanto o tipo `String` listado na Figura 12.20 se refere ao tipo UML. Quando implementamos o sistema em Java, utilizaremos a classe Java `String` para representar esse parâmetro.

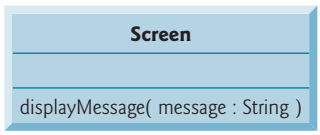


Figura 12.20 | A classe `Screen` com parâmetros de operação.

Identificando e modelando parâmetros de operação para a classe `CashDispenser`

A Figura 12.21 especifica que a operação `dispenseCash` da classe `CashDispenser` recebe um parâmetro `Double`, `amount`, para indicar a quantia monetária (em dólares) a ser entregue. A operação `isSufficientCashAvailable` também recebe um parâmetro `Double` em `amount` para indicar a quantia monetária em questão.

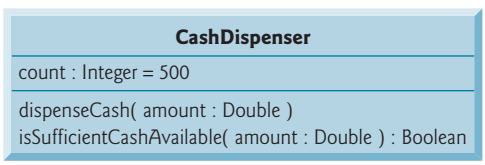


Figura 12.21 | A classe `CashDispenser` com parâmetros de operação.

Identificando e modelando parâmetros de operação para outras classes

Note que não discutimos os parâmetros para a operação `execute` das classes `BalanceInquiry`, `Withdrawal` e `Deposit`, a operação `getInput` da classe `Keypad` e a operação `isEnvelopeReceived` da classe `DepositSlot`. Nessa fase do nosso processo de design, não podemos determinar se essas operações exigem dados adicionais, assim deixamos suas listas de parâmetros vazias. Mais tarde, podemos decidir adicionar parâmetros.

Nesta seção, determinamos várias operações realizadas pelas classes no sistema ATM. Identificamos os parâmetros e os tipos de retorno de algumas operações. À medida que continuamos o nosso processo de design, o número de operações que pertencem a cada classe pode variar — poderíamos descobrir que novas operações são necessárias ou que algumas operações atuais são desnecessárias. Também poderíamos determinar que algumas das nossas operações de classe precisam de parâmetros adicionais e tipos de retorno diferentes ou que alguns parâmetros são desnecessários ou requerem tipos diferentes.

Exercícios de autorrevisão da Seção 12.6

12.14 Qual dos seguintes itens não é um comportamento?

a) ler dados a partir de um arquivo

b) imprimir a saída

c) saída de texto

d) obter a entrada do usuário

12.15 Se você fosse adicionar ao sistema ATM uma operação que retornasse o atributo `amount` da classe `Withdrawal`, como e onde você especificaria essa operação no diagrama de classe da Figura 12.17?

12.16 Descreva o significado da listagem da operação a seguir que poderia aparecer em um diagrama de classe em um design orientado a objetos de uma calculadora:

```
add( x : Integer, y : Integer ) : Integer
```

12.7 Indicando colaboração entre objetos

[*Nota:* esta seção pode ser ensinada depois do Capítulo 7.]

Nesta seção, focalizamos as colaborações (interações) entre objetos. Quando dois objetos se comunicam para realizar uma tarefa, diz-se que eles são **colaboradores** — objetos fazem isso invocando as operações um do outro. Uma **colaboração** consiste em um objeto de uma classe enviar uma **mensagem** a um objeto de outra classe. As mensagens são enviadas no Java via chamadas de método.

Na Seção 12.6, determinamos muitas das operações das classes em nosso sistema. Nesta seção, focalizamos as mensagens que invocam essas operações. Para identificar as colaborações no sistema, retornamos ao documento de requisitos da Seção 12.2. Lembre-se de que esse documento especifica a série de atividades que ocorre durante uma sessão ATM (por exemplo, autenticar um usuário, realizar transações). Os passos utilizados para descrever como o sistema deve realizar cada uma dessas tarefas são nossa primeira indicação das colaborações em nosso sistema. À medida que avançamos nesta seção e no Capítulo 13, podemos descobrir colaborações adicionais.

Identificando as colaborações em um sistema

Identificamos as colaborações no sistema lendo cuidadosamente as seções do documento de requisitos que especificam o que o ATM deve fazer para autenticar um usuário e realizar todo tipo de transação. Para cada ação ou passo descrito, decidimos quais objetos em nosso sistema devem interagir para alcançar o resultado desejado. Identificamos um objeto como o objeto emissor e outro como o objeto receptor. Então selecionamos uma das operações do objeto receptor (identificadas na Seção 12.6) que devem ser invocadas pelo objeto emissor para produzir o comportamento adequado. Por exemplo, o ATM exibe uma mensagem de boas-vindas quando desocupado. Sabemos que um objeto da classe `Screen` exibe uma mensagem para o usuário via sua operação `displayMessage`. Portanto, decidimos que o sistema pode exibir uma mensagem de boas-vindas empregando uma colaboração entre `ATM` e `Screen` em que `ATM` envia uma mensagem `displayMessage` para `Screen` invocando a operação `displayMessage` da classe `Screen`. [*Nota:* para evitar repetir a frase "um objeto da classe...", referenciamos um objeto utilizando seu nome de classe precedido por um artigo (como "um" ou "o") — por exemplo, "o `ATM`" referencia um objeto da classe `ATM`.]

A Figura 12.22 lista as colaborações que podem ser derivadas do documento de requisitos. Para cada objeto emissor, listamos as colaborações na ordem em que elas ocorrem primeiro durante uma sessão ATM (isto é, a ordem em que são discutidas no documento de requisitos). Listamos cada colaboração que envolve um único emissor, mensagem e receptor apenas uma vez, mesmo que as colaborações possam ocorrer em diversas ocasiões diferentes por toda uma sessão ATM. Por exemplo, a primeira linha na Figura 12.22 indica que `ATM` colabora com `Screen` sempre que `ATM` precisar exibir uma mensagem para o usuário.

Vamos considerar as colaborações na Figura 12.22. Antes de permitir a um usuário realizar qualquer transação, o ATM deve pedir-lhe para inserir o número de conta e, em seguida, um PIN. Ele realiza essas tarefas enviando uma mensagem `displayMessage` para `Screen`. Ambas as ações referem-se à mesma colaboração entre o `ATM` e a `Screen`, que já está listada na Figura 12.22. O `ATM` obtém a entrada em resposta a um prompt enviando uma mensagem `getInput` para o `Keypad`. Em seguida, o ATM deve determinar se o número da conta especificado pelo usuário e o PIN correspondem àqueles de uma conta no banco de dados. Ele faz isso enviando uma mensagem `authenticateUser` para `BankDatabase`. Lembre-se de que `BankDatabase` não pode autenticar um usuário diretamente — somente a `Account` do usuário (isto é, a `Account` que contém o número da conta especificado pelo usuário) pode acessar o PIN do usuário em registro para autenticar o usuário. Portanto, a Figura 12.22 lista uma colaboração em que `BankDatabase` envia uma mensagem `validatePIN` para uma `Account`.

Um objeto da classe...	envia a mensagem...	para um objeto de classe...
ATM	displayMessage	Screen
	getInput	Keypad
	authenticateUser	BankDatabase
	execute	BalanceInquiry
	execute	Withdrawal-
	execute	Deposit
BalanceInquiry	getAvailableBalance	BankDatabase
	getTotalBalance	BankDatabase
	displayMessage	Screen
Withdrawal	displayMessage	Screen
	getInput	Keypad
	getAvailableBalance	BankDatabase
	isSufficientCashAvailable	CashDispenser
	debit	BankDatabase
	dispenseCash	CashDispenser
Deposit	displayMessage	Screen
	getInput	Keypad
	isEnvelopeReceived	DepositSlot
	credit	BankDatabase
BankDatabase	validatePIN	Account
	getAvailableBalance	Account
	getTotalBalance	Account
	debit	Account
	credit	Account

Figura 12.22 | Colaborações no sistema ATM.

Depois que o usuário é autenticado, `ATM` exibe o menu principal enviando uma série de mensagens `displayMessage` para `Screen` e obtém a entrada que contém uma seleção de menu enviando uma mensagem `getInput` para `Keypad`. Já explicamos essas colaborações, então não adicionamos nada à Figura 12.22. Depois que o usuário escolhe um tipo da transação, o `ATM` executa a transação enviando uma mensagem `execute` a um objeto da classe de transação apropriada (isto é, um `BalanceInquiry`, um `Withdrawal` ou um `Deposit`). Por exemplo, se o usuário escolher realizar uma consulta de saldo, o `ATM` envia uma mensagem `execute` para um `BalanceInquiry`.

O exame mais detalhado do documento de requisitos revela as colaborações envolvidas na execução de cada tipo de transação. Um `BalanceInquiry` recupera a quantia de dinheiro disponível na conta do usuário enviando uma mensagem `getAvailableBalance` para `BankDatabase`, que responde enviando uma mensagem `getAvailableBalance` ao `Account` do usuário. De maneira semelhante, `BalanceInquiry` recupera o valor do depósito enviando uma mensagem `getTotalBalance` para `BankDatabase`, que envia a mesma mensagem ao `Account` do usuário. Para exibir ambas as partes do saldo da conta do usuário ao mesmo tempo, `BalanceInquiry` envia uma mensagem `displayMessage` para `Screen`.

Um `Withdrawal` responde a uma mensagem `execute` enviando mensagens `displayMessage` a `Screen` para exibir um menu de valores de saque padrão (isto é, US$ 20, US$ 40, US$ 60, US$ 100, US$ 200). `Withdrawal` envia uma mensagem `getInput` para `Keypad` para obter a seleção do usuário. Em seguida, `Withdrawal` determina se a quantia solicitada é menor ou igual ao saldo da conta do usuário. O `Withdrawal` pode obter a quantia em dinheiro disponível enviando uma mensagem `getAvailableBalance` ao `BankDatabase`. `Withdrawal` testa se o dispensador de cédulas contém dinheiro suficiente enviando uma mensagem `isSufficientCashAvailable` para `CashDispenser`. Um `Withdrawal` envia uma mensagem `debit` para `BankDatabase` para diminuir o saldo da conta do usuário. O `BankDatabase` por sua vez envia a mesma mensagem ao `Account` apropriado, que diminui tanto o `totalBalance` como o `availableBalance`. Para liberar o valor solicitado, `Withdrawal` envia uma mensagem `dispenseCash` para `CashDispenser`. Por fim, `Withdrawal` envia uma mensagem `displayMessage` para `Screen`, que instrui o usuário a pegar o dinheiro.

Um `Deposit` responde a uma mensagem `execute` enviando primeiro uma mensagem `displayMessage` para `Screen` para solicitar ao usuário o valor do depósito. `Deposit` envia uma mensagem `getInput` para `Keypad` para obter a entrada do usuário. `Deposit` então envia uma mensagem `displayMessage` para `Screen` para instruir o usuário a inserir um envelope de depósito. Para determinar se a abertura de depósito recebeu um envelope de depósito, `Deposit` envia uma mensagem `isEnvelopeReceived` para `DepositSlot`. `Deposit` atualiza a conta do usuário enviando uma mensagem `credit` para `BankDatabase`, que subsequentemente envia uma mensagem `credit` ao `Account` do usuário. Lembre-se de que creditar fundos em um `Account` aumenta `totalBalance`, mas não o `availableBalance`.

Diagramas de interação

Agora que identificamos as possíveis colaborações entre os nossos objetos de sistema ATM, vamos modelar graficamente essas interações usando a UML. A UML fornece vários tipos de **diagramas de interação** que modelam o comportamento de um sistema modelando a maneira como os objetos interagem. O **diagrama de comunicação** enfatiza *quais objetos* participam das colaborações. [*Nota:* os diagramas de comunicação eram chamados de **diagramas de colaboração** nas versões anteriores da UML.] Como o diagrama de comunicação, o **diagrama de sequência** mostra as colaborações entre objetos, mas enfatiza *quando* as mensagens são enviadas entre objetos *ao longo do tempo*.

Diagramas de comunicação

A Figura 12.23 mostra um diagrama de comunicação que modela o `ATM` para executar um `BalanceInquiry`. Os objetos são modelados na UML como retângulos que contêm nomes na forma *`nomeDoObjeto : NomeDaClasse`*. Nesse exemplo, que envolve apenas um objeto de cada tipo, não damos importância ao nome de objeto e listamos apenas um dois-pontos seguido pelo nome da classe. [*Nota:* ao modelar múltiplos objetos do mesmo tipo, recomenda-se especificar o nome de cada objeto em um diagrama de comunicação.] Os objetos que se comunicam são conectados com linhas sólidas e as mensagens são passadas entre os objetos ao longo dessas linhas na direção mostrada por setas. O nome da mensagem, que aparece junto à seta, é o nome de uma operação (isto é, um método em Java) que pertence ao objeto receptor — considere o nome como um "serviço" que o objeto receptor fornece para enviar objetos (seus clientes).

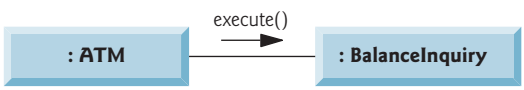


Figura 12.23 | Diagrama de comunicação do ATM que executa uma consulta de saldo.

A seta sólida representa uma mensagem — ou **chamada síncrona** — na UML e uma chamada de método no Java. Essa seta indica que o fluxo de controle é do objeto que envia (o `ATM`) ao objeto que recebe (um `BalanceInquiry`). Visto que essa é uma chamada síncrona, o objeto emissor pode não enviar outra mensagem, ou não fazer coisa alguma, até que o objeto receptor processe a mensagem e retorne o controle para o objeto emissor. O emissor simplesmente espera. Na Figura 12.23, o `ATM` chama o método `BalanceInquiry` `execute` e pode não enviar outra mensagem até `execute` terminar e retornar o controle ao `ATM`. [*Nota:* se isso fosse uma **chamada assíncrona**, representada por uma seta (—≫), o objeto emissor não teria de esperar o objeto receptor para retornar o controle — ele continuaria enviando mensagens adicionais imediatamente após a chamada assíncrona. As chamadas assíncronas são implementadas no Java utilizando uma técnica chamada multithreading, que é discutida no Capítulo 26, Multithreading.]

Sequência de mensagens em um diagrama de comunicação

A Figura 12.24 mostra um diagrama de comunicação que modela as interações entre os objetos do sistema quando um objeto da classe `BalanceInquiry` é executado. Assumimos que o atributo `accountNumber` do objeto contém o número da conta do usuário atual. As colaborações na Figura 12.24 iniciam depois de o `ATM` enviar uma mensagem `execute` para um `BalanceInquiry` (isto é, a interação modelada na Figura 12.23). O número à esquerda de um nome de mensagem indica a ordem em que a mensagem é passada. A **sequência de mensagens** em um diagrama de comunicação progride em ordem numérica do menor para o maior. Nesse diagrama, a numeração inicia com a mensagem `1` e termina com a mensagem `3`. `BalanceInquiry` primeiro envia uma mensagem `getAvailableBalance` ao `BankDatabase` (mensagem `1`) e, então, envia uma mensagem `getTotalBalance` ao `BankDatabase` (mensagem `2`). Dentro dos parênteses que se seguem a um nome de mensagem, podemos especificar uma lista separada por vírgulas dos nomes dos parâmetros enviados com a mensagem (isto é, argumentos em uma chamada de método Java) — o atributo `BalanceInquiry` passa `accountNumber` com suas mensagens para o `BankDatabase` para indicar quais informações de saldo de `Account` recuperar. A partir da Figura 12.18, lembre-se de que as operações `getAvailableBalance` e `getTotalBalance` da classe `BankDatabase` exigem ambas um parâmetro para identificar uma conta. O próximo `BalanceInquiry` exibe `availableBalance` e `totalBalance` para o usuário passando uma mensagem `displayMessage` para `Screen` (mensagem `3`) que inclui um parâmetro que indica `message` a ser exibida.

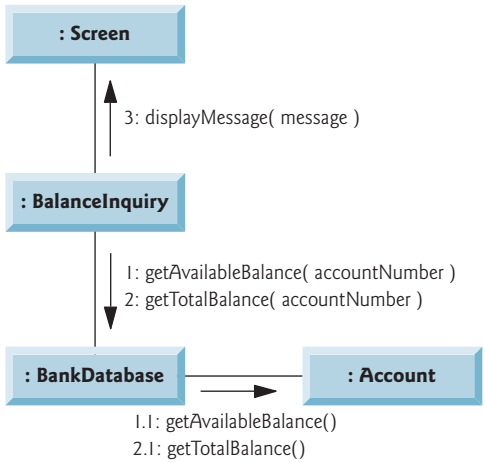


Figura 12.24 | Diagrama de comunicação para executar uma consulta de saldo.

Observe que a Figura 12.24 modela duas mensagens adicionais que passam do `BankDatabase` para uma `Account` (mensagem `1.1` e mensagem `2.1`). Para fornecer o `ATM` dos dois saldos da `Account` do usuário (como solicitado pelas mensagens `1` e `2`), o `BankDatabase` deve passar uma mensagem `getAvailableBalance` e `getTotalBalance` para a `Account` do usuário. Essas mensagens passadas dentro do tratamento de outra mensagem são chamadas de **mensagens aninhadas**. A UML recomenda utilizar um esquema de numeração decimal para indicar mensagens aninhadas. Por exemplo, a mensagem `1.1` é a primeira mensagem aninhada na mensagem `1` — `BankDatabase` passa uma mensagem `getAvailableBalance` durante o processamento por `BankDatabase` de uma mensagem com o mesmo nome. [*Nota*: se `BankDatabase` precisasse passar uma segunda mensagem aninhada durante o processamento da mensagem `1`, a segunda mensagem seria numerada `1.2`.] Uma mensagem só pode ser passada quando todas as mensagens aninhadas da mensagem anterior forem passadas. Por exemplo, `BalanceInquiry` passa a mensagem `3` somente depois que as mensagens `2` e `2.1` forem passadas, nessa ordem.

O esquema aninhado de numeração utilizado nos diagramas de comunicação ajuda a esclarecer precisamente quando e em que contexto cada mensagem é passada. Por exemplo, se numerássemos as mensagens na Figura 12.24 utilizando um esquema de numeração simples (isto é, `1`, `2`, `3`, `4`, `5`), alguém examinando o diagrama poderia não ser capaz de determinar que `BankDatabase` passa a mensagem `getAvailableBalance` (mensagem `1.1`) para um `Account` *durante* o processamento por `BankDatabase` da mensagem `1`, em oposição a *depois* de completar o processamento da mensagem `1`. Os números decimais aninhados deixam claro que a segunda mensagem `getAvailableBalance` (mensagem `1.1`) é passada para um `Account` dentro do tratamento da primeira mensagem `getAvailableBalance` (mensagem `1`) pelo `BankDatabase`.

Diagramas de sequência

Os diagramas de comunicação enfatizam os participantes de colaborações, mas modelam sua sincronização de uma maneira um pouco desajeitada. Um diagrama de sequência ajuda a modelar a sincronização de colaborações mais claramente. A Figura 12.25 mostra um diagrama de sequência modelando a sequência de interações que ocorre quando um `Withdrawal` executa. A linha pontilhada que se estende para baixo do retângulo de um objeto é a **linha da vida** desse objeto, que representa a progressão de tempo. As ações ocorrem ao longo da linha da vida de um objeto na ordem cronológica de cima para baixo — uma ação próxima da parte superior acontece antes de uma próxima da parte inferior.

A passagem de mensagem em diagramas de sequência é semelhante à passagem de mensagem em diagramas de comunicação. Uma seta sólida com uma seta preenchida que se estende do objeto emissor para o objeto receptor representa uma mensagem entre dois objetos. A seta aponta para uma ativação na linha da vida do objeto receptor. Uma **ativação**, mostrada como um pequeno retângulo vertical, indica que um objeto está em execução. Quando um objeto retorna o controle, uma mensagem de retorno, representada como uma linha tracejada com uma seta (⟶), estende-se desde a ativação do objeto que está retornando o controle até a ativação do objeto que inicialmente enviou a mensagem. Para eliminar a poluição visual, omitimos as setas de retorno de mensagem — a UML permite essa prática para tornar os diagramas mais legíveis. Como os diagramas de comunicação, os diagramas de sequência podem indicar parâmetros de mensagem entre os parênteses que se seguem a um nome de mensagem.

A sequência de mensagens na Figura 12.25 inicia quando um `Withdrawal` solicita ao usuário para escolher um valor de saque enviando uma mensagem `displayMessage` para `Screen`. Em seguida, `Withdrawal` envia uma mensagem `getInput` para `Keypad`, que obtém a entrada do usuário. Já modelamos a lógica de controle envolvida. Você pode modelar a lógica de controle em um diagrama de sequência com quadros de UML (que não são abrangidos nesse estudo de caso). Para obter uma breve visão geral dos quadros de UML, visite `www.agilemodeling.com/style/frame.htm`.

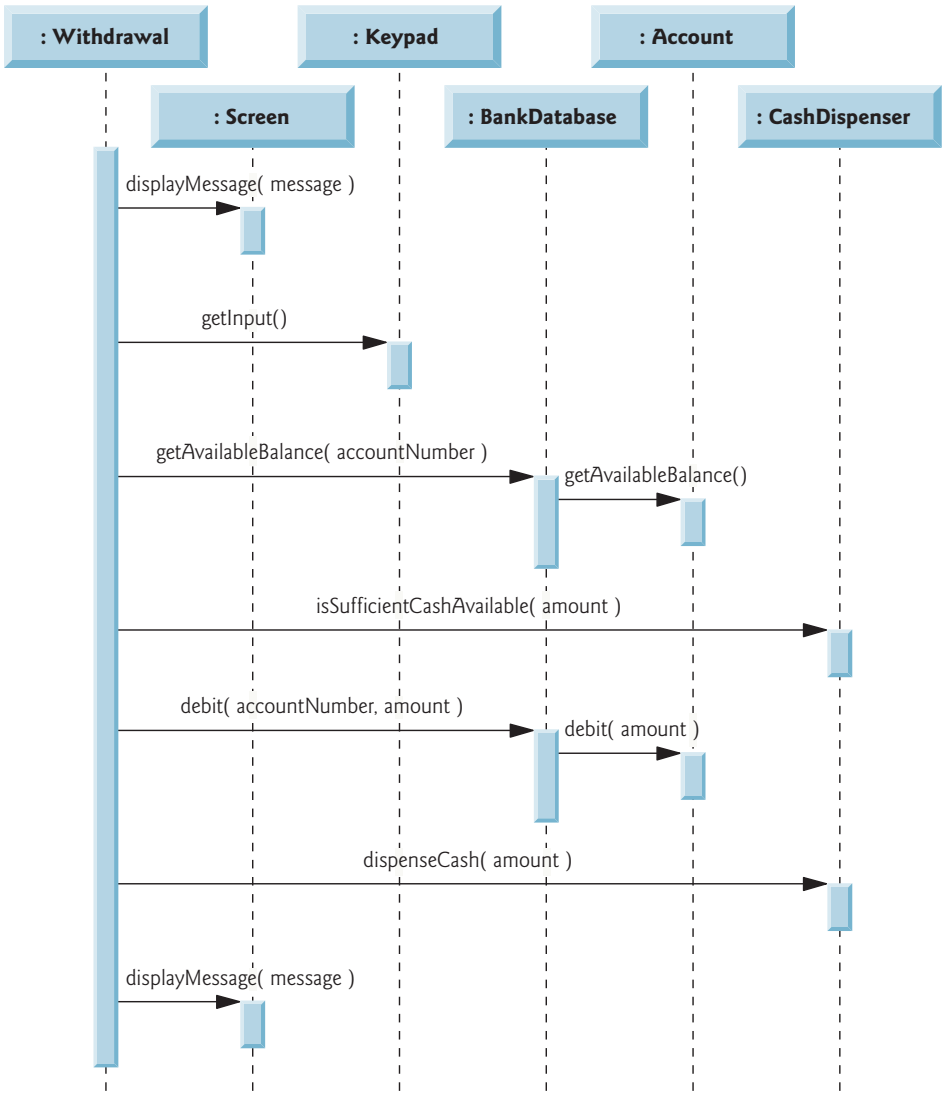


Figura 12.25 | Diagrama de sequência que modela um `Withdrawal` em execução.

Depois de obter o valor de um saque, `Withdrawal` envia uma mensagem `getAvailableBalance` para `BankDatabase`, que por sua vez envia uma mensagem `getAvailableBalance` para o `Account` do usuário. Assumindo que a conta do usuário tem dinheiro suficiente disponível para permitir a transação, `Withdrawal` em seguida envia uma mensagem `isSufficientCashAvailable` para o `CashDispenser`. Assumindo que há suficiente dinheiro disponível, `Withdrawal` diminui o saldo da conta do usuário (isto é, tanto de `totalBalance` como de `availableBalance`) enviando uma mensagem `debit` para `BankDatabase`. `BankDatabase` responde enviando uma mensagem `debit` para o `Account` do usuário. Por fim, `Withdrawal` envia uma mensagem `dispenseCash` para `CashDispenser` e uma mensagem `displayMessage` para `Screen`, que pede para o usuário retirar o dinheiro da máquina.

Identificamos as colaborações entre objetos no sistema ATM e modelamos algumas dessas colaborações utilizando diagramas de interação UML — tanto diagramas de comunicação como diagramas de sequência. Na Seção 13.2, aprimoramos a estrutura do nosso modelo para completar um projeto preliminar orientado a objetos, em seguida iniciamos a implementação do sistema ATM no Java.

Exercícios de autorrevisão da Seção 12.7

12.17 Uma _____ consiste em um objeto de uma classe que envia uma mensagem para um objeto de outra classe.

a) associação
b) agregação
c) colaboração
d) composição

12.18 Que forma de diagrama de interação enfatiza *quais* colaborações ocorrem? Que forma enfatiza *quando* as colaborações ocorrem?

12.19 Crie um diagrama de sequência que modela as interações entre os objetos no sistema ATM que ocorrem quando um `Deposit` executa com sucesso e explique a sequência de mensagens modeladas pelo diagrama.

12.8 Conclusão

Neste capítulo, você aprendeu a trabalhar a partir de um documento de requisitos detalhado para desenvolver um projeto orientado a objetos. Você trabalhou com seis tipos populares de diagramas de UML para modelar graficamente um sistema de software de caixa eletrônico orientado a objetos. No Capítulo 13, aprimoramos o projeto usando herança, então implementamos inteiramente o projeto em um aplicativo Java de 673 linhas.

Respostas dos exercícios de autorrevisão

12.1 A Figura 12.26 contém um diagrama de caso de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.

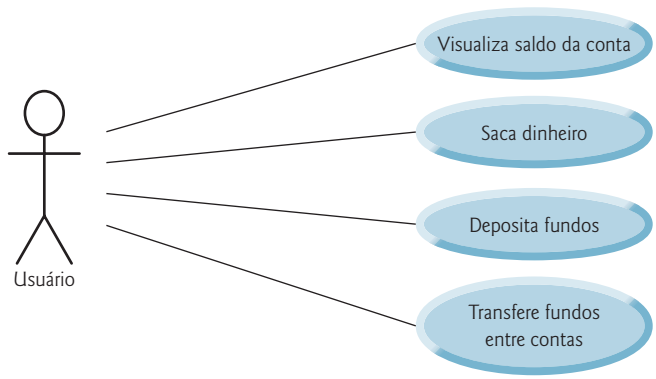


Figura 12.26 | Diagrama de caso de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.

12.2 b.

12.3 d.

12.4 [*Nota*: as respostas do aluno podem variar.] A Figura 12.27 apresenta um diagrama de classe que mostra alguns relacionamentos de composição de uma classe `Car`.

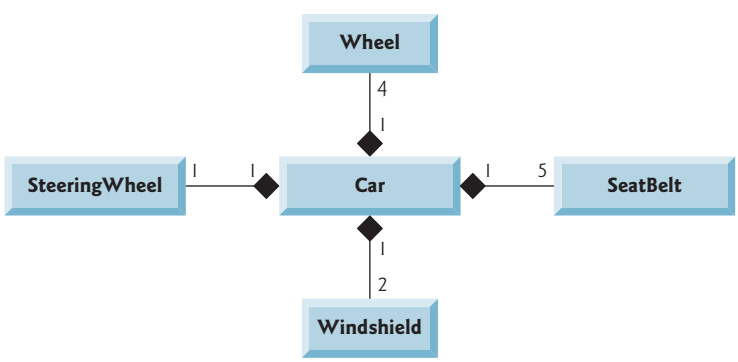


Figura 12.27 | Diagrama de classe que mostra relacionamentos de composição de uma classe `Car`.

12.5 c. [*Nota:* em uma rede de computadores, esse relacionamento poderia ser de muitos para muitos.]

12.6 Verdadeiro.

12.7 A Figura 12.28 apresenta um diagrama de classe para o ATM que inclui a classe `Deposit` em vez da classe `Withdrawal` (como na Figura 12.10). Observe que `Deposit` não acessa `CashDispenser`, mas acessa `DepositSlot`.

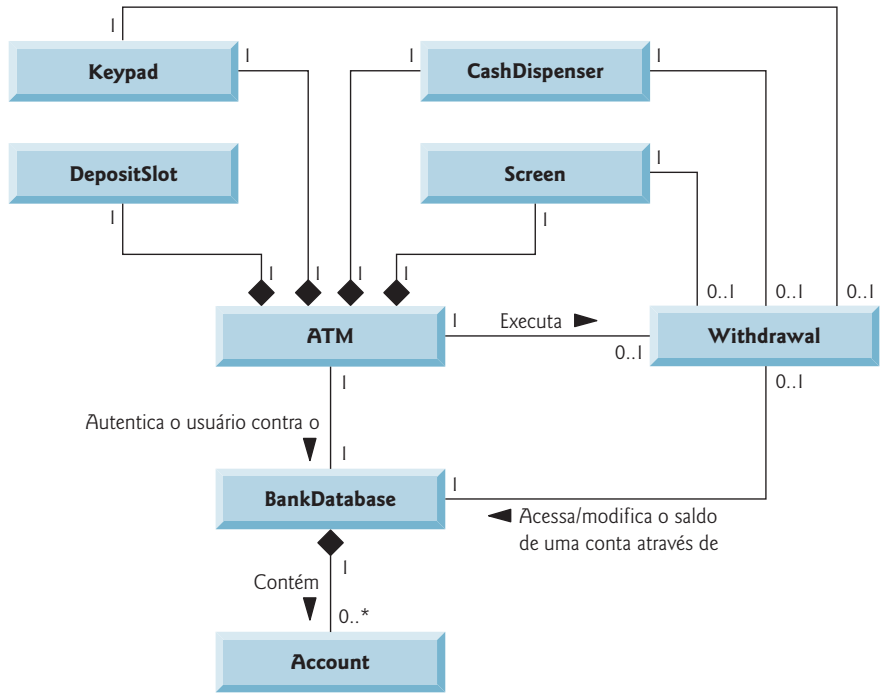


Figura 12.28 | Diagrama de classe para o modelo de sistema ATM incluindo a classe `Deposit`.

12.8 b.

12.9 c. Voo é uma operação ou comportamento de um avião, não um atributo.

12.10 Isso indica que `count` é um `Integer` com um valor inicial de `500`. Esse atributo monitora o número de contas disponível na `CashDispenser` em um determinado momento qualquer.

12.11 Falso. Diagramas de estado modelam algum comportamento de um sistema.

12.12 a.

12.13 A Figura 12.29 modela as ações que ocorrem depois de o usuário escolher a opção de depósito no menu principal e antes de o ATM retornar o usuário para o menu principal. Lembre-se de que parte do recebimento de um depósito do usuário envolve converter um número inteiro de centavos para uma quantia em dólar. Lembre-se também de que fazer um depósito em uma conta aumenta apenas o atributo `totalBalance` do objeto `Account` do usuário. O banco só atualiza o atributo `availableBalance` do objeto `Account` do usuário depois de confirmar a quantia em dinheiro no envelope de depósito e, no caso dos cheques, depois da compensação — isso ocorre independentemente do sistema ATM.

12.14 c.

12.15 Para especificar uma operação que recupera o atributo `amount` da classe `Withdrawal`, a listagem da operação a seguir seria colocada no compartimento de operações (isto é, terceiro) da classe `Withdrawal`:

```
getAmount( ) : Double
```

12.16 Essa listagem de operação indica que uma operação chamada `add` recebe inteiros `x` e `y` como parâmetros e retorna um valor inteiro.

12.17 c.

12.18 Os diagramas de comunicação enfatizam *quais* colaborações ocorrem. Os diagramas de sequência enfatizam *quando* as colaborações ocorrem.

12.19 A Figura 12.30 apresenta um diagrama de sequência que modela as interações entre objetos no sistema ATM que ocorrem quando um `Deposit` executa com sucesso. Um `Deposit` primeiro envia uma mensagem `displayMessage` à `Screen` pedindo para o usuário inserir uma quantia de depósito. Em seguida, `Deposit` envia uma mensagem `getInput` a `Keypad` para receber a entrada do usuário. `Deposit` então instrui o usuário a inserir um envelope de depósito enviando uma mensagem `displayMessage` a `Screen`. `Deposit` em seguida envia uma mensagem `isEnvelopeReceived` a `DepositSlot` para confirmar que o envelope de depósito foi recebido pela ATM. Por fim, `Deposit` aumenta o atributo `totalBalance` (mas não o atributo `availableBalance`) do `Account` do usuário enviando uma mensagem `credit` para `BankDatabase`. `BankDatabase` responde enviando a mesma mensagem para o `Account` do usuário.

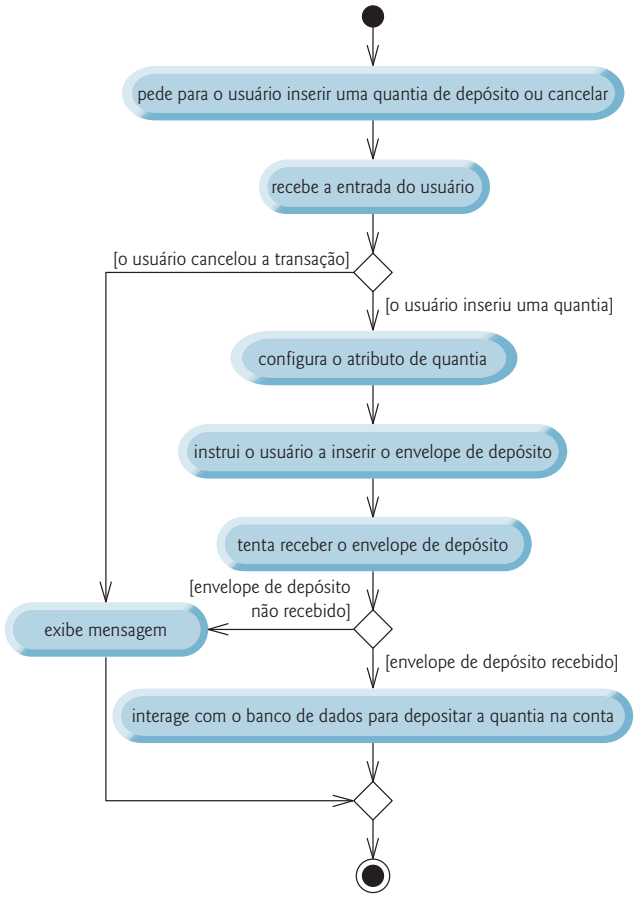


Figura 12.29 | Diagrama de atividade para uma transação de depósito.

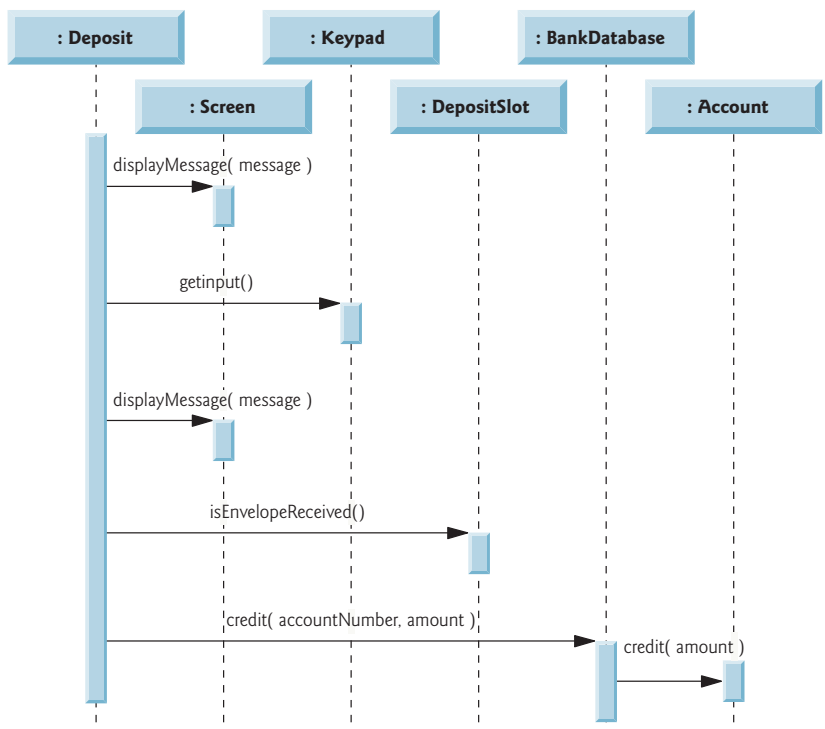


Figura 12.30 | Diagrama de sequência que modela um `Deposit` em execução.

13

Você não pode trabalhar de forma abstrata.
— **I. M. Pei**

Generalizar significa pensar.
— **Georg Wilhelm Friedrich Hegel**

Todos nós somos talentosos. É a nossa herança.
— **Ethel Waters**

Deixe-me passear pelos campos de papel
tocando com minha varinha mágica
os caules secos e as
borboletas que não podem voar …
— **Denise Levertov**

Estudo de caso de ATM, Parte 2: Implementando um projeto orientado a objetos

Objetivos

Neste capítulo, você aprenderá:

- A incorporar herança ao projeto do ATM.
- A incorporar polimorfismo ao projeto do ATM.
- A implementar totalmente no Java o projeto orientado a objetos baseado na UML do software do ATM.
- A estudar um guia passo a passo do código detalhado do sistema de software do ATM que explica as questões de implementação.

Sumário

13.1 Introdução
13.2 Começando a programar as classes do sistema ATM
13.3 Incorporando herança e polimorfismo ao sistema ATM
13.3.1 Implementando o projeto do sistema ATM (incorporando a herança)
13.4 Implementação do estudo de caso de ATM
13.4.1 Classe `ATM`
13.4.2 Classe `Screen`
13.4.3 Classe `Keypad`
13.4.4 Classe `CashDispenser`
13.4.5 Classe `DepositSlot`
13.4.6 Classe `Account`
13.4.7 Classe `BankDatabase`
13.4.8 Classe `Transaction`
13.4.9 Classe `BalanceInquiry`
13.4.10 Classe `Withdrawal`
13.4.11 Classe `Deposit`
13.4.12 Classe `ATMCaseStudy`
13.5 Conclusão

Respostas dos exercícios de autorrevisão

13.1 Introdução

No Capítulo 12, desenvolvemos um projeto orientado a objetos para nosso sistema ATM. Agora, iniciaremos a implementação do nosso projeto orientado a objetos em Java. Na Seção 13.2, mostramos como converter diagramas de classe em código Java. Na Seção 13.3, ajustamos o projeto com herança e polimorfismo. Apresentamos então uma implementação de código Java completa do software ATM na Seção 13.4. O código é cuidadosamente comentado e as discussões da implementação são completas e exatas. Estudar esse aplicativo fornece a oportunidade de ver um aplicativo mais substancial do tipo que provavelmente você encontrará no setor.

13.2 Começando a programar as classes do sistema ATM

[*Nota:* esta seção pode ser ensinada depois do Capítulo 8.]

Visibilidade

Agora, aplicamos modificadores de acesso aos membros das nossas classes. Introduzimos modificadores de acesso `public` e `private`. Os modificadores de acesso determinam a **visibilidade** ou acessibilidade dos atributos e métodos de um objeto a outros objetos. Antes de iniciarmos a implementação do nosso projeto, devemos considerar quais atributos e métodos das nossas classes devem ser `public` e quais devem ser `private`.

Observamos que atributos normalmente devem ser `private` e que os métodos invocados pelos clientes de uma dada classe devem ser `public`. Métodos que são chamados como "métodos utilitários" apenas por outros métodos da classe normalmente devem ser `private`. A UML emprega **marcadores de visibilidade** para modelar a visibilidade dos atributos e operações. Visibilidade pública é indicada colocando um sinal de adição (+) antes de uma operação ou atributo, enquanto um sinal de subtração (–) indica visibilidade privada. A Figura 13.1 mostra nosso diagrama de classe atualizado com marcadores de visibilidade incluídos. [*Nota:* não incluímos nenhum parâmetro de operação na Figura 13.1 — isso é perfeitamente normal. Adicionar marcadores de visibilidade não afeta os parâmetros já modelados nos diagramas de classe das figuras 12.17 a 12.21.]

Navegabilidade

Antes de começarmos a implementar nosso projeto em Java, apresentaremos uma notação da UML adicional. O diagrama de classe na Figura 13.2 refina ainda mais os relacionamentos entre as classes no sistema ATM adicionando setas de navegabilidade às linhas de associação. **Setas de navegabilidade** (representadas como setas com pontas (⟶) no diagrama de classe) indicam a direção em que uma associação pode ser percorrida. Ao implementar um sistema projetado para utilizar a UML, os programadores utilizam setas de navegabilidade para determinar quais objetos precisam de referências a outros objetos. Por exemplo, a seta de navegabilidade que aponta da classe `ATM` para a classe `BankDatabase` indica que podemos navegar do primeiro ao último, permitindo assim que a `ATM` invoque as operações da `BankDatabase`. Entretanto, como a Figura 13.2 não contém uma seta de navegabilidade que aponta da classe `BankDatabase` para a classe `ATM`, a `BankDatabase` não pode acessar as operações da `ATM`. Observe que as associações em um diagrama de classe que têm setas de navegabilidade nas duas extremidades ou que não têm absolutamente nenhuma indicam **navegabilidade bidirecional** — a navegação pode acontecer em qualquer direção pela associação.

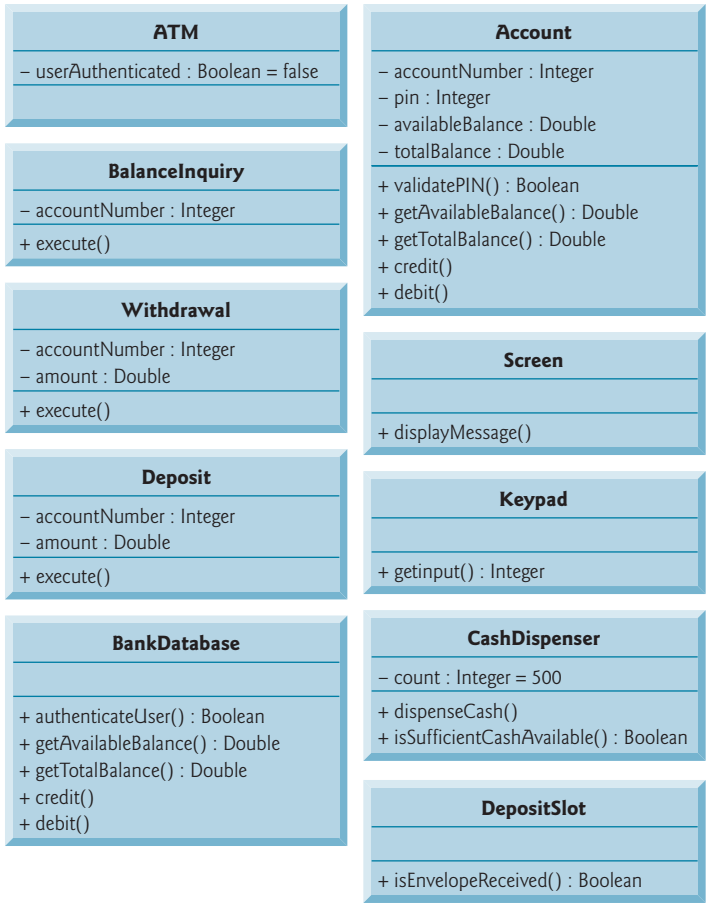


Figura 13.1 | Diagrama de classe com marcadores de visibilidade.

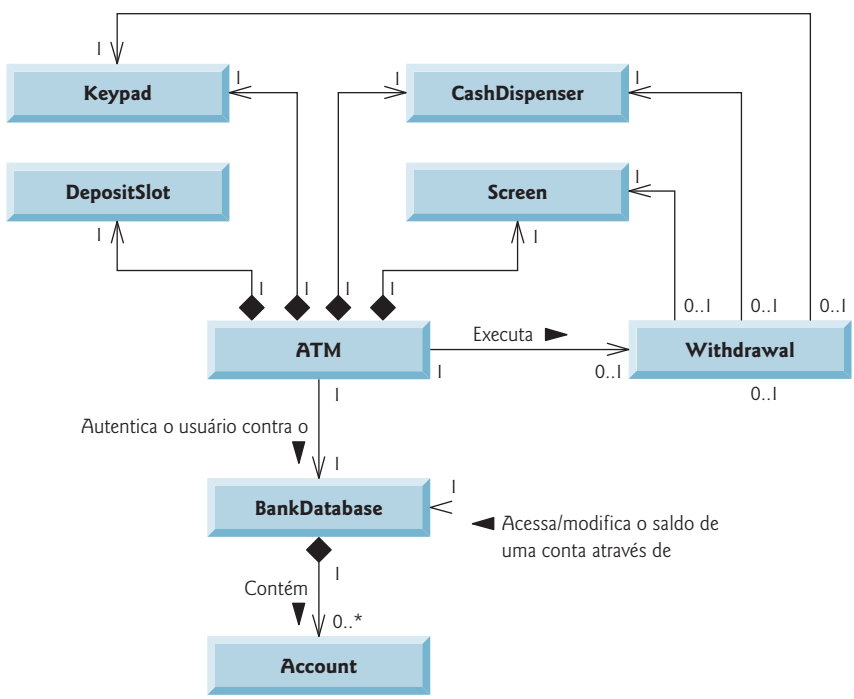


Figura 13.2 | Diagrama de classe com setas de navegabilidade.

Como o diagrama de classe da Figura 12.10, aquele da Figura 13.2 omite as classes `BalanceInquiry` e `Deposit` por simplicidade. A navegabilidade das associações em que essas classes participam é bem parecida com a da classe `Withdrawal`. A partir da Seção 12.3, lembre-se de que `BalanceInquiry` tem uma associação com a classe `Screen`. Podemos navegar da classe `BalanceInquiry` para a classe `Screen` ao longo dessa associação, mas não podemos navegar da classe `Screen` para a classe `BalanceInquiry`. Dessa forma, se quiséssemos modelar a classe `BalanceInquiry` na Figura 13.2, colocaríamos uma seta de navegabilidade no final da classe `Screen` dessa associação. Além disso, lembre-se de que a classe `Deposit` se associa com as classes `Screen`, `Keypad` e `DepositSlot`. Podemos navegar da classe `Deposit` para cada uma dessas classes, mas não vice-versa. Portanto, colocaríamos setas de navegabilidade nas extremidades `Screen`, `Keypad` e `DepositSlot` dessas associações. [*Nota*: Modelamos essas classes e associações adicionais no nosso diagrama de classe final na Seção 13.3, depois de simplificarmos a estrutura do nosso sistema incorporando o conceito de herança ao projeto orientado a objetos.]

Implementando o sistema ATM a partir de seu projeto em UML

Agora, estamos prontos para começar a implementar o sistema ATM. Primeiro, converteremos as classes dos diagramas da Figura 13.1 e Figura 13.2 em código Java. O código representará o "esqueleto" do sistema. Na Seção 13.3, modificamos o código para incorporar a herança. Na Seção 13.4, apresentamos o código Java totalmente funcional para nosso modelo.

Como um exemplo, desenvolvemos o código a partir do nosso projeto da classe `Withdrawal` na Figura 13.1. Utilizamos essa figura para determinar os atributos e as operações da classe. Utilizamos o modelo da UML na Figura 13.2 para determinar as associações entre as classes. Seguimos as quatro diretrizes a seguir para cada classe:

1. Utilize o nome localizado no primeiro compartimento para declarar a classe como uma classe `public` com um construtor vazio, sem argumentos. Incluímos esse construtor simplesmente como uma reserva de espaço para lembrar-nos de que a maioria das classes de fato precisará de construtores personalizados. Na Seção 13.4, ao completarmos uma versão funcional dessa classe, adicionaremos todos os argumentos necessários e codificaremos o corpo do construtor, conforme necessário. Por exemplo, a classe `Withdrawal` produz o código na Figura 13.3. [*Nota:* se descobrirmos que as variáveis de instância da classe requerem somente inicialização padrão, removeremos o construtor vazio sem argumentos porque ele é desnecessário.]

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // construtor sem argumentos
5     public Withdrawal()
6     {
7     } // fim do construtor sem argumentos Withdrawal
8  } // fim da classe Withdrawal
```

Figura 13.3 | Código Java para a classe `Withdrawal` baseado nas figuras 13.1 e 13.2.

2. Utilize os atributos localizados no segundo compartimento para declarar as variáveis de instância. Por exemplo, os atributos `private accountNumber` e `amount` da classe `Withdrawal` produzem o código na Figura 13.4. [*Nota:* O construtor da versão funcional completa dessa classe atribuirá os valores a esses atributos.]

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta a sacar fundos
6     private double amount; // quantia a sacar
7
8     // construtor sem argumentos
9     public Withdrawal()
10    {
11    } // fim do construtor sem argumentos Withdrawal
12 } // fim da classe Withdrawal
```

Figura 13.4 | Código Java para a classe `Withdrawal` baseado nas figuras 13.1 e13.2.

3. Utilize as associações descritas no diagrama de classe para declarar as referências a outros objetos. Por exemplo, de acordo com a Figura 13.2, `Withdrawal` pode acessar um objeto da classe `Screen`, um objeto da classe `Keypad`, um objeto da classe `CashDispenser` e um objeto da classe `BankDatabase`. Isso produz o código na Figura 13.5. [*Nota:* o construtor da versão funcional completa dessa classe inicializará essas variáveis de instância com referências a objetos reais.]

4. Utilize as operações localizadas no terceiro compartimento da Figura 13.1 para declarar o shell dos métodos. Se ainda não especificamos um tipo de retorno para uma operação, declaramos o método com um tipo de retorno `void`. Consulte os diagramas de classe das figuras 12.17 a 12.21 para declarar quaisquer parâmetros necessários. Por exemplo, adicionar a operação `public execute` na classe `Withdrawal`, que tem uma lista vazia de parâmetros, fornece o código na Figura 13.6. [*Nota:* codificamos os corpos dos métodos depois de implementarmos o sistema completo na Seção 13.4.]

Isso conclui nossa discussão dos princípios básicos de gerar classes de diagramas de UML.

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta a sacar fundos
6     private double amount; // quantia a sacar
7
8     // referências a objetos associados
9     private Screen screen; // Tela do ATM
10    private Keypad keypad; // Teclado do ATM
11    private CashDispenser cashDispenser; // dispensador de cédulas do ATM
12    private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
13
14    // construtor sem argumentos
15    public Withdrawal()
16    {
17    } // fim do construtor sem argumentos Withdrawal
18 } // fim da classe Withdrawal
```

Figura 13.5 | Código Java para a classe `Withdrawal` baseado nas figuras 13.1e 3.2.

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta a sacar fundos
6     private double amount; // quantia a sacar
7
8     // referências a objetos associados
9     private Screen screen; // Tela do ATM
10    private Keypad keypad; // Teclado do ATM
11    private CashDispenser cashDispenser; // dispensador de cédulas do ATM
12    private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
13
14    // construtor sem argumentos
15    public Withdrawal()
16    {
17    } // fim do construtor sem argumentos Withdrawal
18
19    // operações
20    public void execute()
21    {
22    } // fim do método execute
23 } // fim da classe Withdrawal
```

Figura 13.6 | Código Java para a classe `Withdrawal` baseado nas figuras 13.1 e 13.2.

Exercícios de autorrevisão da Seção 13.2

13.1 Determine se a seguinte sentença é *verdadeira* ou *falsa* e, se *falsa*, explique por quê: se um atributo de uma classe estiver marcado com um sinal de subtração (–) em um diagrama de classe, o atributo não estará diretamente acessível fora da classe.

13.2 Na Figura 13.2, a associação entre a ATM e a `Screen` indica que:

a) podemos navegar da `Screen` para a ATM

b) podemos navegar da ATM para a `Screen`

c) Tanto (a) como (b); a associação é bidirecional

d) Nenhuma acima

13.3 Escreva um código Java para começar a implementar o projeto da classe `Keypad`.

13.3 Incorporando herança e polimorfismo ao sistema ATM

[*Nota:* esta seção pode ser ensinada depois do Capítulo 10.]

Agora, iremos revisitar nosso projeto do sistema ATM para verificar como ele se beneficia da herança. Para aplicar a herança, primeiro procuramos aspectos comuns entre as classes no sistema. Criamos uma hierarquia de herança para modelar classes semelhantes (mas não idênticas) de uma maneira mais elegante e eficiente. Então modificamos nosso diagrama de classe para incorporar os novos relacionamentos de herança. Por fim, demonstramos como nosso projeto atualizado é traduzido em código Java.

Na Seção 12.3, encontramos o problema da representação de uma transação financeira no sistema. Em vez de criar uma classe para representar todos os tipos de transação, decidimos criar três classes individuais de transação — `BalanceInquiry`, `Withdrawal` e `Deposit` — para representar as transações que o sistema ATM pode realizar. A Figura 13.7 mostra os atributos e as operações das classes `BalanceInquiry`, `Withdrawal` e `Deposit`. Observe que essas classes contêm um atributo (`accountNumber`) e uma operação (`execute`) em comum. Cada classe requer o atributo `accountNumber` para especificar a conta à qual a transação se aplica. Cada classe contém uma operação `execute`, que a ATM invoca para realizar a transação. Claramente, `BalanceInquiry`, `Withdrawal` e `Deposit` representam *tipos de* transações. A Figura 13.7 revela os aspectos comuns entre as classes de transação, portanto utilizar a herança para dividir os recursos comuns parece apropriado para projetar as classes `BalanceInquiry`, `Withdrawal` e `Deposit`. Colocamos a funcionalidade comum em uma superclasse, `Transaction`, que as classes `BalanceInquiry`, `Withdrawal` e `Deposit` estendem.

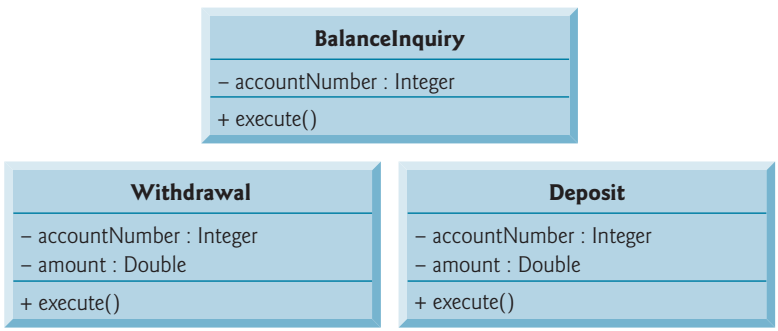


Figura 13.7 | Atributos e operações de `BalanceInquiry`, `Withdrawal` e `Deposit`.

Generalização

A UML especifica um relacionamento chamado **generalização** para modelar a herança. A Figura 13.8 é o diagrama de classe que modela a generalização da superclasse `Transaction` e das subclasses `BalanceInquiry`, `Withdrawal` e `Deposit`. As setas com pontas triangulares ocas indicam que as classes `BalanceInquiry`, `Withdrawal` e `Deposit` estendem a classe `Transaction`. Diz-se que a classe `Transaction` é uma generalização das classes `BalanceInquiry`, `Withdrawal` e `Deposit`. Diz-se que as classes `BalanceInquiry`, `Withdrawal` e `Deposit` são **especializações** da classe `Transaction`.

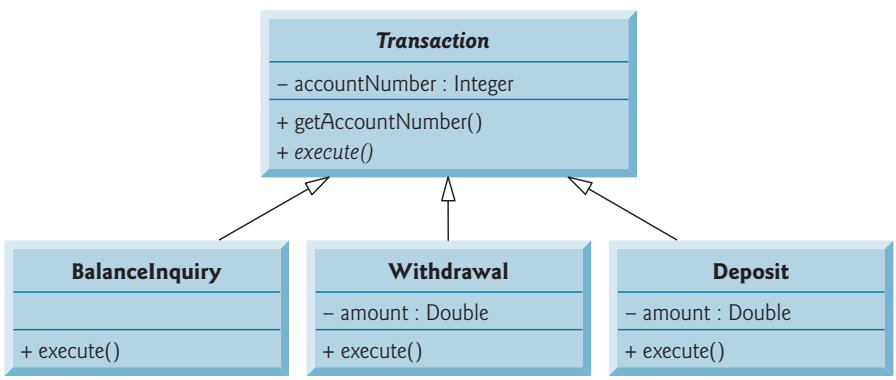


Figura 13.8 | Diagrama de classe que modela a generalização da superclasse `Transaction` e das subclasses `BalanceInquiry`, `Withdrawal` e `Deposit`. Observe que os nomes das classes abstratas (por exemplo, `Transaction`) e os nomes dos métodos (por exemplo, `execute` na classe `Transaction`) aparecem em itálico.

As classes `BalanceInquiry`, `Withdrawal` e `Deposit` compartilham o atributo do tipo inteiro `accountNumber`, assim dividimos esse atributo comum e o colocamos na superclasse `Transaction`. Não mais listamos `accountNumber` no segundo compartimento de cada subclasse, porque as três subclasses herdam esse atributo de `Transaction`. Lembre-se, porém, de que as subclasses não podem acessar diretamente atributos `private` de uma superclasse. Portanto, incluímos o método `public getAccountNumber` na classe `Transaction`. Cada subclasse herdará esse método, permitindo que a subclasse acesse seu `accountNumber` conforme necessário para executar uma transação.

De acordo com a Figura 13.7, as classes `BalanceInquiry`, `Withdrawal` e `Deposit` também compartilham a operação `execute`, portanto inserimos o método `public execute` na superclasse `Transaction`. Mas, não faz sentido implementar `execute` na classe `Transaction`, porque a funcionalidade que esse método fornece depende do tipo de transação real. Portanto, declaramos o método `execute` como `abstract` na superclasse `Transaction`. Qualquer classe que contém pelo menos um método abstrato também deve ser declarada `abstract`. Isso força quaisquer subclasses de `Transaction` a serem uma classe concreta (isto é, `BalanceInquiry`, `Withdrawal` e `Deposit`) para implementar o método `execute`. A UML exige que coloquemos nomes de classes abstratas (e métodos abstratos) em itálico, portanto `Transaction` e seu método `execute` aparecem em itálico na Figura 13.8. Observe que o método `execute` não está em itálico nas subclasses `BalanceInquiry`, `Withdrawal` e `Deposit`. Cada subclasse sobrescreve o método `execute` da superclasse `Transaction` com uma implementação concreta que realiza os procedimentos apropriados para completar esse tipo de transação. Observe que a Figura 13.8 inclui a operação `execute` no terceiro compartimento das classes `BalanceInquiry`, `Withdrawal` e `Deposit`, pois cada classe tem uma implementação concreta diferente do método sobrescrito.

Processando `Transactions` de uma maneira polimórfica

O polimorfismo fornece à classe `ATM` uma maneira elegante de executar todas as transações "no geral". Por exemplo, suponha que um usuário opte por realizar uma consulta de saldos. O `ATM` configura uma referência `Transaction` a um novo objeto `BalanceInquiry`. Quando a `ATM` utiliza sua referência `Transaction` para invocar o método `execute`, a versão de `execute` de `BalanceInquiry` é chamada.

Essa abordagem polimórfica também torna o sistema facilmente extensível. Se quiséssemos criar um novo tipo de transação (por exemplo, uma transferência de fundos ou um pagamento de conta), simplesmente criaríamos uma subclasse `Transaction` adicional que sobrescreve o método `execute` com uma versão do método apropriado para executar o novo tipo de transação. Só seriam necessárias alterações mínimas no código do sistema para permitir que os usuários escolhessem o novo tipo de transação no menu principal e para que o `ATM` instanciasse e executasse objetos da nova subclasse. O `ATM` executaria as transações do novo tipo utilizando o código atual, uma vez que ela executa todas as transações polimorficamente utilizando uma referência `Transaction` geral.

Lembre-se de que uma classe abstrata como `Transaction` é uma para a qual o programador nunca pretende instanciar objetos. Uma classe abstrata simplesmente declara atributos e comportamentos comuns das suas subclasses em uma hierarquia de herança. A classe `Transaction` define o conceito de uma transação com um número de conta e a executa. Talvez você pergunte por que nos preocupamos em incluir o método `abstract execute` na classe `Transaction` se não há uma implementação concreta. Conceitualmente, o incluímos porque ele corresponde ao comportamento definidor de todas as transações — execução. Tecnicamente, devemos incluir o método `execute` na superclasse `Transaction` de modo que a `ATM` (ou qualquer outra classe) possa invocar polimorficamente a versão sobrescrita desse método de cada subclasse por meio de uma referência `Transaction`. Além disso, da perspectiva de engenharia de software, incluir um método abstrato em uma superclasse força o implementador das subclasses a sobrescrever esse método com implementações concretas nas subclasses ou, caso contrário, as subclasses também serão abstratas, impedindo que objetos dessas subclasses sejam instanciados.

Atributo adicional das classes `Withdrawal` e `Deposit`

As subclasses `BalanceInquiry`, `Withdrawal` e `Deposit` herdam o atributo `accountNumber` da superclasse `Transaction`, mas as classes `Withdrawal` e `Deposit` contêm o atributo adicional `amount` que as distingue da classe `BalanceInquiry`. As classes `Withdrawal` e `Deposit` requerem esse atributo adicional para armazenar a quantia que o usuário deseja sacar ou depositar. A classe `BalanceInquiry` não precisa desse atributo e requer somente um número de conta para executar. Ainda que duas das três subclasses `Transaction` compartilhem esse atributo, não o colocamos na superclasse `Transaction` — colocamos somente os recursos comuns a todas as subclasses na superclasse, caso contrário as subclasses herdariam os atributos (e métodos) de que elas não necessitam e não devem ter.

Diagrama de classe com hierarquia `Transaction` incorporada

A Figura 13.9 apresenta um diagrama de classe atualizado do nosso modelo que incorpora a herança e introduz a classe `Transaction`. Modelamos uma associação entre as classes `ATM` e `Transaction` para mostrar que a `ATM`, em um dado momento, executa ou não uma transação (isto é, há zero ou um objeto do tipo `Transaction` no sistema por vez). Como uma `Withdrawal` é um tipo de `Transaction`, não mais desenha uma linha de associação diretamente entre a classe `ATM` e a classe `Withdrawal`. A subclasse `Withdrawal` herda a associação da superclasse `Transaction` com a classe `ATM`. As subclasses `BalanceInquiry` e `Deposit` também herdam essa associação; portanto, as associações anteriormente omitidas entre `ATM` e as classes `BalanceInquiry` e `Deposit` não existem mais.

Também adicionamos uma associação entre as classes `Transaction` e `BankDatabase` (Figura 13.9). Todas as classes `Transaction` exigem uma referência a `BankDatabase` para que possam acessar e modificar as informações de conta. Como cada subclasse `Transaction` herda essa referência, não mais modelamos a associação entre as classes `Withdrawal` e `BankDatabase`. De maneira semelhante, as associações anteriormente omitidas entre as classes `BankDatabase` e `BalanceInquiry` e `Deposit` não existem mais.

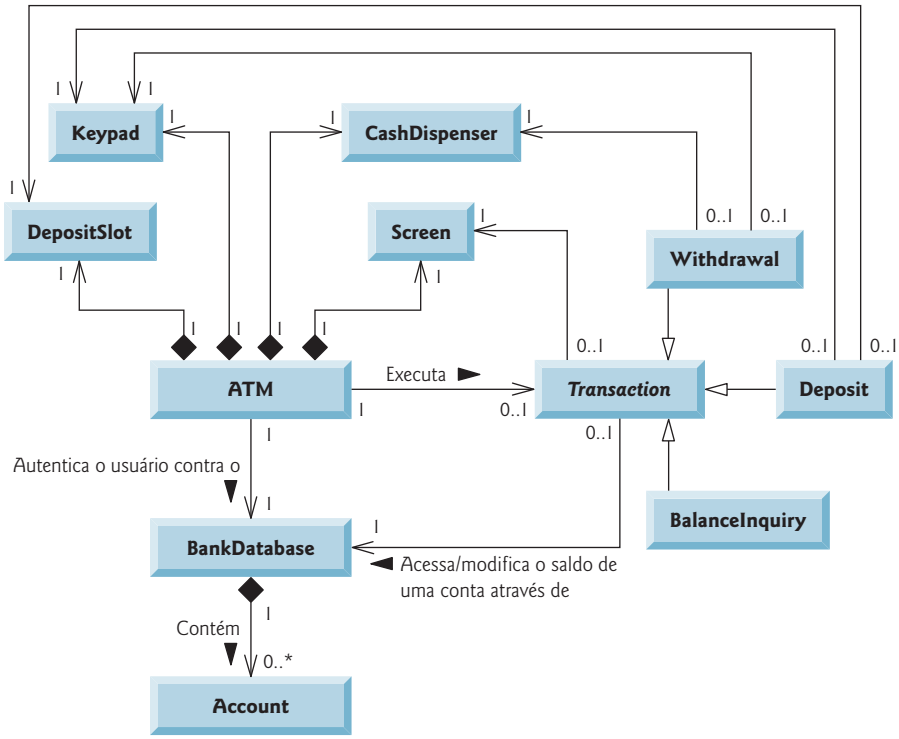


Figura 13.9 | Diagrama de classe do sistema ATM (incorporando a herança). Observe que o nome da classe abstrata `Transaction` aparece em itálico.

Mostramos uma associação entre as classes `Transaction` e `Screen`. Todas as classes `Transaction` exibem a saída para o usuário via `Screen`. Portanto, não mais incluímos a associação anteriormente modelada entre `Withdrawal` e `Screen`, embora `Withdrawal` ainda participe nas associações com a `CashDispenser` e `Keypad`. Nosso diagrama de classe que incorpora a herança também modela `Deposit` e `BalanceInquiry`. Mostramos as associações entre `Deposit`, `DepositSlot` e `Keypad`. Observe que a classe `BalanceInquiry` não faz parte de nenhuma associação além daquelas herdadas da classe `Transaction` — uma classe `BalanceInquiry` precisa interagir somente com `BankDatabase` e com `Screen`.

A Figura 13.1 mostrou os atributos e as operações com marcadores de visibilidade. Agora, na Figura 13.10, apresentamos um diagrama de classe modificado que incorpora herança. Esse diagrama abreviado não mostra os relacionamentos de herança, mas, em vez disso, mostra os atributos e métodos depois que a herança foi empregada no nosso sistema. Para economizar espaço, como fizemos na Figura 12.12, não incluímos aqueles atributos mostrados pelas associações na Figura 13.9 — mas os incluímos na implementação Java na Seção 13.4. Também omitimos todos os parâmetros de operação, como fizemos na Figura 13.1 — incorporar a herança não afeta os parâmetros já modelados nas figuras 12.17 a 12.21.

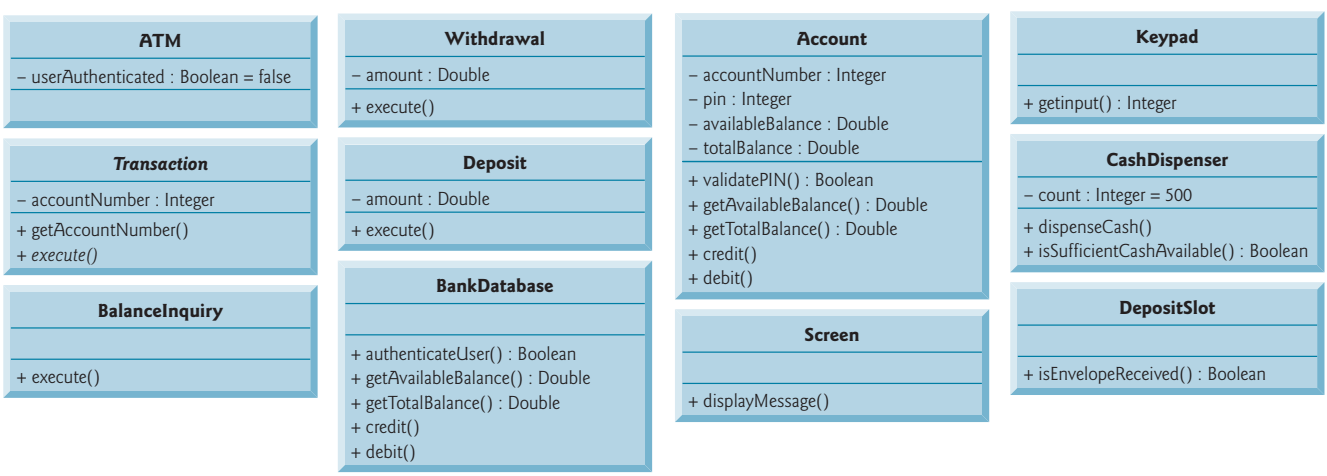


Figura 13.10 | Diagrama de classe com atributos e operações (incorporando a herança). Observe que o nome da classe abstrata `Transaction` e o nome do método abstrato `execute` na classe `Transaction` aparecem em itálico.

Observação de engenharia de software 13.1

Um diagrama de classe completo mostra todas as associações entre as classes e todos os atributos e operações para cada classe. Se o número de atributos de classe, métodos e associações for muito alto (como nas figuras 13.9 e 13.10), uma boa prática que promove a legibilidade é dividir essas informações entre dois diagramas de classe — um focalizando as associações e, o outro, os atributos e métodos.

13.3.1 Implementando o projeto do sistema ATM (incorporando a herança)

Na Seção 13.2, começamos a implementar o projeto do sistema ATM no código Java. Agora, modificaremos nossa implementação para incorporar a herança, utilizando a classe `Withdrawal` como um exemplo.

1. Se uma classe `A` for uma generalização da classe `B`, então a classe `B` estende a classe `A` na declaração de classe. Por exemplo, a superclasse abstrata `Transaction` é uma generalização da classe `Withdrawal`. A Figura 13.11 mostra a declaração da classe `Withdrawal`.

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal extends Transaction
3  {
4  } // fim da classe Withdrawal
```

Figura 13.11 | O código Java para o shell da classe `Withdrawal`.

2. Se a classe `A` for uma classe abstrata e a classe `B` for uma subclasse da classe `A`, a classe `B` deve então implementar os métodos abstratos da classe `A` se a classe `B` tiver de ser uma classe concreta. Por exemplo, a classe `Transaction` contém o método abstrato `execute`, assim a classe `Withdrawal` deve implementar esse método se quisermos instanciar um objeto `Withdrawal`. A Figura 13.12 é o código Java da classe `Withdrawal` das figuras 13.9 e 13.10. A classe `Withdrawal` herda o campo `accountNumber` da superclasse `Transaction`, portanto `Withdrawal` não precisa declarar esse campo. A classe `Withdrawal` também herda as referências a `Screen` e o `BankDatabase` da sua superclasse `Transaction`, portanto não incluímos essas referências no nosso código. A Figura 13.10 especifica o atributo `amount` e a operação `execute` para a classe `Withdrawal`. A linha 6 da Figura 13.12 declara um campo para o atributo `amount`. As linhas 16–19 declaram o shell de um método para a operação `execute`. Lembre-se de que a subclasse `Withdrawal` deve fornecer uma implementação concreta do método `abstract execute` na superclasse `Transaction`. As referências `keypad` e `cashDispenser` (linhas 7–8) são campos derivados das associações de `Withdrawal` na Figura 13.9. [*Nota:* o construtor na versão funcional completa dessa classe inicializará essas referências como objetos reais.]

```
1  // Withdrawal.java
2  // Gerada com os diagramas de classe na Figura 13.9 e na Figura 13.10
3  public class Withdrawal extends Transaction
4  {
5     // atributos
6     private double amount; // quantia a sacar
7     private Keypad keypad; // referência ao teclado numérico
8     private CashDispenser cashDispenser; // referência ao dispensador de cédulas
9
10    // construtor sem argumentos
11    public Withdrawal()
12    {
13    } // fim do construtor sem argumentos Withdrawal
14
15    // método sobrescrevendo execute
16    @Override
17    public void execute()
18    {
19    } // fim do método execute
20 } // fim da classe Withdrawal
```

Figura 13.12 | Código Java para a classe `Withdrawal` baseado nas figuras 13.9 e 13.10.

Observação de engenharia de software 13.2

Diversas ferramentas de modelagem UML podem converter designs baseados em UML para código Java, acelerando consideravelmente o processo de implementação. Para informações adicionais sobre essas ferramentas, visite nosso UML Resource Center em `www.deitel.com/UML/`.

Parabéns por completar a parte do projeto do estudo de caso! Implementamos o sistema ATM no código Java na Seção 13.4. Recomendamos que você leia cuidadosamente o código e sua descrição. O código está bem comentado e segue precisamente o design com o qual você agora está familiarizado. A descrição de acompanhamento foi cuidadosamente escrita para orientar sua implementação com base no projeto em UML. Dominar esse código é uma realização maravilhosa depois de estudar as seções 12.2 a 12.7 e 13.2 a 13.3.

Exercícios de autorrevisão da Seção 13.3

13.4 A UML utiliza uma seta com uma _____ para indicar um relacionamento de generalização.

a) ponta de seta com preenchimento sólido

b) ponta de seta triangular oca

c) ponta de seta oca na forma de losango

d) ponta de seta angular

13.5 Determine se a seguinte sentença é *verdadeira* ou *falsa* e, se *falsa*, explique por quê: a UML exige que nomes das classes abstratas e de métodos sejam sublinhados.

13.6 Escreva um código Java para começar a implementar o design da classe `Transaction` especificada nas figuras 13.9 e 13.10. Certifique-se de incluir atributos de tipo por referência `private` com base nas associações da classe `Transaction`. Também se certifique de incluir os métodos *get* `public` que fornecem acesso a qualquer desses atributos `private` que as subclasses exigem para realizar suas tarefas.

13.4 Implementação do estudo de caso de ATM

Esta seção contém a implementação das 673 linhas totalmente funcionais do sistema ATM. Consideramos as classes na ordem em que as identificamos na Seção 12.3 — `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `Transaction`, `BalanceInquiry`, `Withdrawal` e `Deposit`.

Aplicamos as diretrizes discutidas nas seções 13.2 e 13.3 para codificar essas classes com base na maneira como as modelamos nos diagramas de classe UML das figuras 13.9 e 13.10. Para desenvolver os corpos para os métodos das classes, utilizamos os diagramas de atividades da Seção 12.5 e os diagramas de sequência e comunicação apresentados na Seção 12.7. Observe que o nosso projeto do ATM não especifica toda a lógica do programa e talvez não especifique todos os atributos e operações necessárias para completar a implementação do ATM. Isso é uma parte normal do processo de um projeto orientado a objetos. À medida que implementamos o sistema, completaremos a lógica do programa e adicionaremos atributos e comportamentos conforme necessário para construir o sistema ATM especificado pelo documento de requisitos na Seção 12.2.

Concluímos a discussão apresentando um aplicativo Java (`ATMCaseStudy`) que inicia o ATM e coloca em uso as outras classes no sistema. Lembre-se de que estamos desenvolvendo uma primeira versão do sistema ATM que é executada em um computador pessoal e utilizamos o teclado e monitor do computador para simular o teclado numérico e a tela do sistema ATM. Também simulamos apenas as ações do dispensador de cédulas e da abertura para depósito. Tentamos, porém, implementar o sistema de modo que versões reais de hardware desses dispositivos possam ser integradas sem alterações significativas no código.

13.4.1 Classe ATM

A classe `ATM` (Figura 13.13) representa o ATM como um todo. As linhas 6–12 implementam os atributos dessa classe. Determinamos todos, exceto um desses atributos, a partir dos diagramas de classe UML das figuras 13.9 e 13.10. Note que implementamos o atributo `Boolean userAuthenticated` da UML na Figura 13.10 como um `boolean` em Java (linha 6). A linha 7 declara um atributo como não localizado no nosso projeto de UML — um atributo `int currentAccountNumber` que monitora o número de conta do atual usuário autenticado. Mais adiante, veremos como a classe usa esse atributo. As linhas 8–12 declaram atributos de tipo por referência correspondentes às associações da classe `ATM` modeladas no diagrama de classe da Figura 13.9. Esses atributos permitem ao ATM acessar suas partes (isto é, `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`) e interagir com o banco de dados de informações sobre contas bancárias (isto é, um objeto `BankDatabase`).

```
1  // ATM.Java
2  // Representa um caixa eletrônico
3
4  public class ATM
5  {
6     private boolean userAuthenticated; // se usuário foi autenticado
7     private int currentAccountNumber; // número atual da conta de usuário
8     private Screen screen; // tela do ATM
9     private Keypad keypad; // teclado do ATM
10    private CashDispenser cashDispenser; // dispensador de cédulas do ATM
11    private DepositSlot depositSlot; // abertura para depósito do ATM
12    private BankDatabase bankDatabase; // banco de dados com as informações sobre as contas
13
```

```
14    // constantes que correspondem às principais opções de menu
15    private static final int BALANCE_INQUIRY = 1;
16    private static final int WITHDRAWAL = 2;
17    private static final int DEPOSIT = 3;
18    private static final int EXIT = 4;
19
20    // construtor sem argumentos de ATM inicializa as variáveis de instância
21    public ATM()
22    {
23       userAuthenticated = false; // usuário não foi autenticado para iniciar
24       currentAccountNumber = 0; // nenhum número atual de conta para iniciar
25       screen = new Screen(); // cria a tela
26       keypad = new Keypad(); // cria o teclado
27       cashDispenser = new CashDispenser(); // cria o dispensador de cédulas
28       depositSlot = new DepositSlot(); // cria a abertura para depósito
29       bankDatabase = new BankDatabase(); // cria o banco de dados com informações sobre as contas
30    } // fim do construtor ATM sem argumentos
31
32    // inicia o ATM
33    public void run()
34    {
35       // dá boas-vindas e autentica o usuário; realiza transações
36       while ( true )
37       {
38          // faz um loop enquanto o usuário ainda não está autenticado
39          while ( !userAuthenticated )
40          {
41             screen.displayMessageLine( "\nWelcome!" );
42             authenticateUser(); // autentica o usuário
43          } // fim do while
44
45          performTransactions(); // o usuário agora está autenticado
46          userAuthenticated = false; // reinicializa antes da próxima sessão do ATM
47          currentAccountNumber = 0; // reinicializa antes da próxima sessão do ATM
48          screen.displayMessageLine( "\nThank you! Goodbye!" );
49       } // fim do while
50    } // fim do método run
51
52    // tenta autenticar o usuário contra o banco de dados
53    private void authenticateUser()
54    {
55       screen.displayMessage( "\nPlease enter your account number: " );
56       int accountNumber = keypad.getInput(); // insere o número de conta
57       screen.displayMessage( "\nEnter your PIN: " ); // solicita o PIN
58       int pin = keypad.getInput(); // insere o PIN
59
60       // configura userAuthenticated como um valor booleano retornado pelo banco de dados
61       userAuthenticated =
62          bankDatabase.authenticateUser( accountNumber, pin );
63
64       // verifica se a autenticação foi bem-sucedida
65       if ( userAuthenticated )
66       {
67          currentAccountNumber = accountNumber; // salva a conta do usuário #
68       } // fim do if
69       else
70          screen.displayMessageLine(
71             "Invalid account number or PIN. Please try again." );
72    } // fim do método authenticateUser
73
74    // exibe o menu principal e realiza transações
75    private void performTransactions()
76    {
77       // variável local para armazenar a transação atualmente processada
78       Transaction currentTransaction = null;
79
80       boolean userExited = false; // usuário optou por não sair
81
82       // faz um loop enquanto o usuário não escolher a opção para sair do sistema
```

```
83          while ( !userExited )
84          {
85             // mostra o menu principal e obtém a seleção de usuário
86             int mainMenuSelection = displayMainMenu();
87
88             // decide como prosseguir com base na seleção de menu feita pelo usuário
89             switch ( mainMenuSelection )
90             {
91                // o usuário optou por realizar um entre três tipos de transações
92                case BALANCE_INQUIRY:
93                case WITHDRAWAL:
94                case DEPOSIT:
95
96                   // inicializa como o novo objeto do tipo escolhido
97                   currentTransaction =
98                      createTransaction( mainMenuSelection );
99
100                  currentTransaction.execute(); // executa a transação
101                  break;
102               case EXIT: // usuário optou por terminar a sessão
103                  screen.displayMessageLine( "\nExiting the system..." );
104                  userExited = true; // essa sessão de ATM deve terminar
105                  break;
106               default: // usuário não inseriu um inteiro de 1 a 4
107                  screen.displayMessageLine(
108                     "\nYou did not enter a valid selection. Try again." );
109                  break;
110            } // fim do switch
111         } // fim do while
112      } // fim do método performTransactions
113
114      // exibe o menu principal e retorna uma seleção de entrada
115      private int displayMainMenu()
116      {
117         screen.displayMessageLine( "\nMain menu:" );
118         screen.displayMessageLine( "1 - View my balance" );
119         screen.displayMessageLine( "2 - Withdraw cash" );
120         screen.displayMessageLine( "3 - Deposit funds" );
121         screen.displayMessageLine( "4 - Exit\n" );
122         screen.displayMessage( "Enter a choice: " );
123         return keypad.getInput(); // retorna a seleção do usuário
124      } // fim do método displayMainMenu
125
126      // retorna o objeto da subclasse de Transaction especificada
127      private Transaction createTransaction( int type )
128      {
129         Transaction temp = null; // variável Transaction temporária
130
131         // determina qual tipo de Transaction criar
132         switch ( type )
133         {
134            case BALANCE_INQUIRY: // cria uma nova transação BalanceInquiry
135               temp = new BalanceInquiry(
136                  currentAccountNumber, screen, bankDatabase );
137               break;
138            case WITHDRAWAL: // cria uma nova transação Withdrawal
139               temp = new Withdrawal( currentAccountNumber, screen,
140                  bankDatabase, keypad, cashDispenser );
141               break;
142            case DEPOSIT: // cria uma nova transação Deposit
143               temp = new Deposit( currentAccountNumber, screen,
144                  bankDatabase, keypad, depositSlot );
145               break;
146         } // fim do switch
147
148         return temp; // retorna o objeto recém-criado
149      } // fim do método createTransaction
150   } // fim da classe ATM
```

Figura 13.13 | A classe ATM representa o ATM.

As linhas 15–18 declaram constantes inteiras que correspondem às quatro opções no menu principal do ATM (isto é, consulta de saldos, retirada, depósito e saída). As linhas 21–30 declaram o construtor, que inicializa os atributos da classe. Quando um objeto `ATM` é criado pela primeira vez, nenhum usuário é autenticado, assim a linha 23 inicializa `userAuthenticated` como `false`. Da mesma forma, a linha 24 inicializa `currentAccountNumber` como `0` porque ainda não há nenhum usuário atual. As linhas 25–28 instanciam novos objetos para representar as partes do ATM. Lembre-se de que a classe `ATM` tem relacionamentos de composição com as classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`, portanto a classe `ATM` é responsável pela criação dessas classes. A linha 29 cria um novo `BankDatabase`. [*Nota:* se esse sistema ATM fosse real, a classe `ATM` receberia uma referência a um objeto existente de banco de dados criado pelo banco. Mas, nessa implementação só estamos simulando o banco de dados do banco, portanto a classe `ATM` cria o objeto `BankDatabase` com o qual interage.]

Método `ATM` `run`

O diagrama de classe da Figura 13.10 não lista nenhuma operação para a classe `ATM`. Agora, implementaremos uma operação (isto é, o método `public`) na classe `ATM` que permite a um cliente externo da classe (isto é, a classe `ATMCaseStudy`) instruir a `ATM` a executar. O método `run` de `ATM` (linhas 33–50) utiliza um loop infinito (linhas 36–49) para repetidamente dar boas-vindas a um usuário, tenta autenticar o usuário e, se a autenticação for bem-sucedida, permite que o usuário realize as transações. Depois que um usuário autenticado realiza as transações desejadas e escolhe sair, o ATM redefine a si mesmo, exibe uma mensagem de adeus para o usuário e reinicia o processo. Aqui, utilizamos um loop infinito para simular o fato de que um ATM parece executar continuamente até o banco desligá-lo (uma ação além do controle do usuário). Um usuário do ATM tem a opção de sair do sistema, mas não a capacidade de desligar o ATM completamente.

Autenticando um usuário

No loop infinito do método interno `run`, as linhas 39–43 fazem com que o ATM repetidamente emita uma mensagem de boas-vindas e tente autenticar o usuário contanto que este não esteja autenticado (isto é, `!userAuthenticated` é `true`). A linha 41 invoca o método `displayMessageLine` de `screen` da classe `ATM` para exibir uma mensagem de boas vindas. Como ocorre com o método `displayMessage` de `Screen` projetado no estudo de caso, o método `displayMessageLine` (declarado nas linhas 13–16 da Figura 13.14) exibe uma mensagem para o usuário, mas esse método também gera a saída de uma nova linha depois da mensagem. Adicionamos esse método durante a implementação para fornecer aos clientes da classe `Screen` mais controle sobre o posicionamento das mensagens exibidas. A linha 42 invoca o método utilitário `private` `authenticateUser` da classe `ATM` (declarado nas linhas 53–72) para tentar autenticar o usuário.

Nós nos referimos ao documento de requisitos para determinar os passos necessários a fim de autenticar o usuário antes de permitir a ocorrência de transações. A linha 55 do método `authenticateUser` invoca o método `displayMessage` de `screen` para solicitar que o usuário insira um número de conta. A linha 56 invoca o método `getInput` de `keypad` para obter a entrada do usuário e, então, armazena o valor inteiro inserido pelo usuário em uma variável local `accountNumber`. Em seguida, o método `authenticateUser` pede para o usuário inserir um PIN (linha 57) e armazena o PIN inserido pelo usuário em uma variável local `pin` (linha 58). As linhas 61–62 tentam então autenticar o usuário passando o `accountNumber` e `pin` inserido pelo usuário para o método `authenticateUser` de `bankDatabase`. A classe `ATM` configura seu atributo `userAuthenticated` como o valor `boolean` retornado por esse método — `userAuthenticated` torna-se `true` se a autenticação for bem-sucedida (isto é, `accountNumber` e `pin` correspondem àqueles de uma `Account` existente em `bankDatabase`) e, do contrário, permanece `false`. Se `userAuthenticated` for `true`, a linha 67 salva o número de conta inserido pelo usuário (isto é, `accountNumber`) no atributo `currentAccountNumber` de ATM. Os outros métodos `ATM` utilizam essa variável sempre que uma sessão do ATM exige acesso ao número da conta do usuário. Se `userAuthenticated` é `false`, as linhas 70–71 utilizam o método `displayMessageLine` de `screen` para indicar que um número inválido de conta e/ou PIN foi inserido e o usuário deve tentar novamente. Observe que configuramos `currentAccountNumber` somente depois de autenticar o número da conta do usuário e o PIN associado — se o banco de dados não puder autenticar o usuário, `currentAccountNumber` permanece `0`.

Depois de o método `run` tentar autenticar o usuário (linha 42), se `userAuthenticated` ainda for `false`, o loop `while` nas linhas 39–43 é executado novamente. Se agora `userAuthenticated` for `true`, o loop terminará e o controle continua com a linha 45, que chama o método utilitário `performTransactions` da classe `ATM`.

Executando transações

O método `performTransactions` (linhas 75–112) executa uma sessão de ATM para um usuário autenticado. A linha 78 declara uma variável `Transaction` local à qual atribuiremos um objeto `BalanceInquiry`, `Withdrawal` ou `Deposit` que representa a transação do ATM que atualmente está sendo processada. Observe que aqui utilizamos uma variável `Transaction` para permitir que tiremos proveito do polimorfismo. Também observe que nomeamos essa variável depois do nome de papel incluído no diagrama de classe da Figura 12.7 — `currentTransaction`. A linha 80 declara uma outra variável local — uma `boolean` chamada `userExited` que monitora se o usuário optou por sair. Essa variável controla um loop `while` (linhas 83–111) que permite ao usuário executar um número ilimitado de transações antes de optar por sair. Dentro desse loop, a linha 86 exibe o menu principal e obtém a seleção de menu do usuário chamando o método utilitário `displayMainMenu` de `ATM` (declarado nas linhas 115–124). Esse método exibe o menu principal invocando os métodos de `screen` da classe `ATM` e retorna uma seleção de menu obtido do usuário pela `keypad` da `ATM`. A linha 86 armazena a seleção do usuário retornado por `displayMainMenu` na variável local `mainMenuSelection`.

Depois de obter uma seleção no menu principal, o método `performTransactions` utiliza uma instrução `switch` (linhas 89–110) para responder à seleção apropriadamente. Se `mainMenuSelection` for igual a uma das três constantes inteiras que representam tipos de transação (isto é, se o usuário optou por realizar uma transação), as linhas 97–98 chamam o método utilitário `createTransaction` (declarado nas linhas 127–149) para retornar um objeto instanciado recentemente do tipo que corresponde à transação selecionada. A variável

`currentTransaction` recebe a referência retornada por `createTransaction` e então a linha 100 invoca o método `execute` dessa transação para executá-la. Discutiremos o método `execute` de `Transaction` e as três subclasses `Transaction` mais adiante. Atribuímos à variável `Transaction` de `currentTransaction` um objeto de uma das três subclasses `Transaction` para que seja possível executar as transações polimorficamente. Por exemplo, se o usuário escolher realizar uma consulta de saldos, `mainMenuSelection` será igual a `BALANCE_INQUIRY`, levando `createTransaction` a retornar um objeto `BalanceInquiry`. Portanto, `currentTransaction` refere-se a uma `BalanceInquiry`, e invocar `currentTransaction.execute()` resulta no fato de que a versão de `BalanceInquiry` para `execute` é chamada.

Criando uma transação

O método `createTransaction` (linhas 127–149) usa uma instrução `switch` (linhas 132–146) para instanciar um novo objeto da subclasse `Transaction` do tipo indicado pelo parâmetro `type`. Lembre-se de que o método `performTransactions` passa `mainMenuSelection` para esse método somente se `mainMenuSelection` contiver um valor correspondente a um dos três tipos de transação. Portanto, `type` é `BALANCE_INQUIRY`, `WITHDRAWAL` ou `DEPOSIT`. Cada `case` na instrução `switch` instancia um novo objeto chamando o construtor da subclasse `Transaction` apropriado. Cada construtor contém uma lista única de parâmetros, baseada nos dados específicos necessários para inicializar o objeto da subclasse. Uma `BalanceInquiry` exige apenas o número de conta do usuário atual e referências a `bankDatabase` e `screen` da classe `ATM`. Além desses parâmetros, um `Withdrawal` requer referências a `keypad` e `cashDispenser` da classe `ATM` e um `Deposit` requer referências a `keypad` e `depositSlot` da classe `ATM`. Discutimos as classes de transação em mais detalhes nas seções 13.4.8 a 13.4.11.

Fechando o menu principal e processando seleções inválidas

Depois de executar uma transação (linha 100 em `performTransactions`), `userExited` permanece `false` e as linhas 83–111 se repetem, retornando o usuário ao menu principal. Mas, se um usuário não realizar uma transação e, em vez disso, selecionar uma opção de saída no menu principal, a linha 104 configura `userExited` como `true`, fazendo com que a condição do loop `while` (`!userExited`) torne-se `false`. Esse `while` é a instrução final do método `performTransactions`, portanto o controle retorna ao método chamador `run`. Se o usuário inserir uma seleção inválida no menu principal (isto é, não um inteiro entre 1 e 4), as linhas 107–108 exibem uma mensagem de erro apropriada, `userExited` permanece `false` e o usuário retorna ao menu principal para tentar novamente.

Esperando o próximo usuário do ATM

Quando o controle de `performTransactions` retorna ao método `run`, o usuário optou por sair do sistema, então as linhas 46–47 redefinem os atributos `userAuthenticated` e `currentAccountNumber` de `ATM` para preparar-se para o próximo usuário do ATM. A linha 48 exibe uma mensagem de adeus antes de o ATM recomeçar e dar boas-vindas ao próximo usuário.

13.4.2 Classe Screen

A classe `Screen` (Figura 13.14) representa a tela do ATM e encapsula todos os aspectos da exibição de saída para o usuário. A classe `Screen` lembra a tela de um ATM real de um monitor de computador e gera saída de mensagens de texto utilizando os métodos padrão de saída de console `System.out.print`, `System.out.println` e `System.out.printf`. Nesse estudo de caso, projetamos a classe `Screen` com uma operação — `displayMessage`. Para melhor flexibilidade ao exibir mensagens em `Screen`, agora declaramos três métodos `Screen` — `displayMessage`, `displayMessageLine` e `displayDollarAmount`.

```
1  // Screen.java
2  // Representa a tela do ATM
3
4  public class Screen
5  {
6     // exibe uma mensagem sem retorno de carro
7     public void displayMessage( String message )
8     {
9        System.out.print( message );
10    } // fim do método displayMessage
11
12    // exibe uma mensagem com um retorno de carro
13    public void displayMessageLine( String message )
14    {
15       System.out.println( message );
16    } // fim do método displayMessageLine
17
18    // exibe um valor em dólares
19    public void displayDollarAmount( double amount )
20    {
21       System.out.printf( "$%,.2f", amount );
22    } // fim do método displayDollarAmount
23 } // fim da classe Screen
```

Figura 13.14 | A classe `Screen` representa a tela do ATM.

O método `displayMessage` (linhas 7–10) recebe um argumento `String` e imprime-o no console. O cursor permanece na mesma linha, tornando esse método apropriado para exibir solicitações para o usuário. O método `displayMessageLine` (linhas 13–16) faz a mesma coisa usando `System.out.println`, que gera a saída de uma nova linha para mover o cursor para a próxima linha. Por fim, o método `displayDollarAmount` (linhas 19–22) gera a saída de uma quantia em dólares adequadamente formatada (por exemplo, `$1,234.56`). A linha 21 utiliza `System.out.printf` para gerar a saída de um valor `double` formatado com vírgulas a fim de aumentar a legibilidade e duas casas decimais.

13.4.3 Classe Keypad

A classe `Keypad` (Figura 13.15) representa o teclado numérico do ATM e é responsável por receber todas as entradas de usuário. Lembre-se de que estamos simulando esse hardware, portanto utilizamos o teclado do computador para simular o teclado numérico. Utilizamos a classe `Scanner` para obter a entrada de console do usuário. O teclado de um computador contém muitas teclas não encontradas no teclado numérico do ATM. Entretanto, supomos que o usuário pressione somente as teclas presentes no teclado de um computador que também aparecem no teclado numérico — as teclas numeradas de 0–9 e a tecla *Enter*.

```
1  // Keypad.java
2  // Representa o teclado do ATM
3  import java.util.Scanner; // o programa utiliza Scanner para obter a entrada de usuário
4
5  public class Keypad
6  {
7     private Scanner input; // lê os dados na linha de comando
8
9     // o construtor sem argumentos inicializa a classe Scanner
10    public Keypad()
11    {
12       input = new Scanner( System.in );
13    } // fim do construtor Keypad sem argumentos
14
15    // retorna um valor inteiro inserido pelo usuário
16    public int getInput()
17    {
18       return input.nextInt(); // supomos que o usuário insira um inteiro
19    } // fim do método getInput
20 } // fim da classe Keypad
```

Figura 13.15 | A classe `Keypad` representa o teclado do ATM.

A linha 3 da classe `Keypad` utiliza `import` para importar a classe `Scanner` para uso na classe `Keypad`. A linha 7 declara a variável `input` de `Scanner` como uma variável de instância. A linha 12 no construtor cria um novo objeto `Scanner` que lê a entrada a partir do fluxo de entrada padrão (`System.in`) e atribui a referência do objeto à variável `input`. O método `getInput` (declarado nas linhas 16–19) invoca o método `nextInt` de `Scanner` (linha 18) para retornar o próximo inteiro inserido pelo usuário. [*Nota:* o método `nextInt` pode lançar uma `InputMismatchException` se o usuário inserir uma entrada de número não inteiro. Uma vez que o teclado numérico de um ATM real só permite entrada de inteiros, supomos que nenhuma exceção ocorrerá e não tentaremos corrigir esse problema. Consulte o Capítulo 11 para informações sobre como capturar exceções.] Lembre-se de que `nextInt` obtém todas as entradas utilizadas pelo ATM. O método `getInput` de `Keypad` simplesmente retorna o inteiro inserido pelo usuário. Se um cliente da classe `Keypad` exigir uma entrada que satisfaz alguns critérios (isto é, um número que corresponde a uma opção válida no menu), o cliente deverá realizar a verificação de erros.

13.4.4 Classe CashDispenser

A classe `CashDispenser` (Figura 13.16) representa o dispensador de cédulas do ATM. A linha 7 declara a constante `INITIAL_COUNT`, que indica a contagem inicial de cédulas no dispensador de cédulas quando o ATM é inicializado (isto é, `500`). A linha 8 implementa o atributo `count` (modelado na Figura 13.10), que monitora o número de cédulas que permanecem no `CashDispenser` em um dado momento. O construtor (linhas 11–14) configura `count` como a conta inicial. `CashDispenser` tem dois métodos `public` — `dispenseCash` (linhas 17–21) e `isSufficientCashAvailable` (linhas 24–32). A classe confia no fato de que um cliente (isto é, `Withdrawal`) chama `dispenseCash` somente depois de estabelecer que há cédulas suficientes disponíveis chamando `isSufficientCashAvailable`. Portanto, a classe `dispenseCash` simplesmente simula o ato de entregar a quantia solicitada sem verificar se há cédulas suficientes disponíveis.

```
1  // CashDispenser.java
2  // Representa o dispensador de cédulas do ATM
3
4  public class CashDispenser
```

```
5  {
6     // o número inicial padrão de cédulas no dispensador de cédulas
7     private final static int INITIAL_COUNT = 500;
8     private int count; // número de cédulas de US$ 20 remanescente
9
10    // construtor sem argumentos CashDispenser inicializa a count para o padrão
11    public CashDispenser()
12    {
13       count = INITIAL_COUNT; // configura atributo count como o padrão
14    } // fim do construtor CashDispenser
15
16    // simula a entrega da quantia especificada de cédulas
17    public void dispenseCash( int amount )
18    {
19       int billsRequired = amount / 20; // número de cédulas de US$ 20 requerido
20       count -= billsRequired; // atualiza a contagem das cédulas
21    } // fim do método dispenseCash
22
23    // indica se o dispensador de cédulas pode entregar a quantia desejada
24    public boolean isSufficientCashAvailable( int amount )
25    {
26       int billsRequired = amount / 20; // número de cédulas de US$ 20 requerido
27
28       if ( count >= billsRequired  )
29          return true; // há cédulas suficientes disponíveis
30       else
31          return false; // não há cédulas suficientes disponíveis
32    } // fim do método isSufficientCashAvailable
33 } // fim da classe CashDispenser
```

Figura 13.16 | A classe `CashDispenser` representa o dispensador de cédulas do ATM.

O método `isSufficientCashAvailable` (linhas 24–32) contém um parâmetro `amount` que especifica a quantia de cédulas em questão. A linha 26 calcula o número de cédulas de US$ 20 requeridas para entregar o `amount` especificado. O ATM permite que o usuário escolha somente quantias de retirada que são múltiplos de US$ 20, assim dividimos `amount` por `20` para obter o número de `billsRequired`. As linhas 28–31 retornam `true` se o `count` de `CashDispenser` for maior ou igual a `billsRequired` (isto é, há cédulas suficientes disponíveis) e `false`, caso contrário (isto é, sem cédulas suficientes). Por exemplo, se um usuário deseja sacar US$ 80 (isto é, `billsRequired` é 4), mas só há três cédulas (isto é, `count` é 3), o método retorna `false`.

O método `dispenseCash` (linhas 17–21) simula a entrega de cédulas. Se nosso sistema estivesse acoplado a um dispensador de cédulas de um hardware real, esse método interagiria com o dispositivo para fisicamente entregar cédulas. Nossa versão do método simplesmente diminui a `count` das cédulas remanescentes de acordo com o número requerido para entregar o `amount` especificado (linha 20). Observe que é responsabilidade do cliente da classe (isto é, `Withdrawal`) informar o usuário de que cédulas foram entregues — `CashDispenser` não pode interagir diretamente com `Screen`.

13.4.5 Classe `DepositSlot`

A classe `DepositSlot` (Figura 13.17) representa a abertura para depósito do ATM. Como a classe `CashDispenser`, `DepositSlot` simplesmente simula a funcionalidade de um mecanismo real de entrada de depósito. `DepositSlot` não tem nenhum atributo e somente um método — `isEnvelopeReceived` (linhas 8–11) — que indica se um envelope de depósito foi recebido.

```
1  // DepositSlot.java
2  // Representa a abertura para depósito do ATM
3
4  public class DepositSlot
5  {
6     // indica se o envelope foi recebido (sempre retorna true,
7     // porque isso só é uma simulação do software de uma abertura para depósito real)
8     public boolean isEnvelopeReceived()
9     {
10       return true; // o envelope de depósito foi recebido
11    } // fim do método isEnvelopeReceived
12 } // fim da classe DepositSlot
```

Figura 13.17 | A classe `DepositSlot` representa a abertura para depósito do ATM.

Lembre-se de que no documento de requisitos o ATM permite que o usuário insira um envelope dentro de no máximo dois minutos. A versão atual do método `isEnvelopeReceived` simplesmente retorna `true` imediatamente (linha 10), pois isso só é uma simulação de software e supomos que o usuário inseriu um envelope dentro do prazo exigido. Se um hardware real de abertura para depósito estivesse conectado ao nosso sistema, o método `isEnvelopeReceived` poderia ser implementado para esperar no máximo dois minutos a fim de receber um sinal do hardware de abertura para depósito indicando que o usuário de fato inseriu um envelope de depósito. Se `isEnvelopeReceived` fosse receber esse sinal dentro de dois minutos, o método retornaria `true`. Se dois minutos tivessem passado e o método ainda não tivesse recebido um sinal, o método então retornaria `false`.

13.4.6 Classe Account

A classe `Account` (Figura 13.18) representa uma conta bancária. Cada `Account` tem quatro atributos (modelados na Figura 13.10) — `accountNumber`, `pin`, `availableBalance` e `totalBalance`. As linhas 6–9 implementam esses atributos como campos `private`. A variável `availableBalance` representa a quantia de fundos disponível para saque. A variável `totalBalance` representa a quantia de fundos disponível, mais a quantia de fundos depositados ainda aguardando confirmação ou compensação.

```
1  // Account.java
2  // Representa uma conta bancária
3
4  public class Account
5  {
6     private int accountNumber; // número da conta
7     private int pin; // PIN para autenticação
8     private double availableBalance; // fundos disponíveis para saque
9     private double totalBalance; // fundos disponíveis + depósitos pendentes
10
11    // O construtor Account inicializa os atributos
12    public Account( int theAccountNumber, int thePIN,
13       double theAvailableBalance, double theTotalBalance )
14    {
15       accountNumber = theAccountNumber;
16       pin = thePIN;
17       availableBalance = theAvailableBalance;
18       totalBalance = theTotalBalance;
19    } // fim do construtor Account
20
21    // determina se um PIN especificado pelo usuário corresponde ao PIN em Account
22    public boolean validatePIN( int userPIN )
23    {
24       if ( userPIN == pin )
25          return true;
26       else
27          return false;
28    } // fim do método validatePIN
29
30    // retorna o saldo disponível
31    public double getAvailableBalance()
32    {
33       return availableBalance;
34    } // fim de getAvailableBalance
35
36    // retorna o saldo total
37    public double getTotalBalance()
38    {
39       return totalBalance;
40    } // fim do método getTotalBalance
41
42    // credita uma quantia à conta
43    public void credit( double amount )
44    {
45       totalBalance += amount; // adiciona ao saldo total
46    } // fim do método credit
47
48    // debita uma quantia da conta
49    public void debit( double amount )
50    {
51       availableBalance -= amount; // subtrai do saldo disponível
```

```
52          totalBalance -= amount; // subtrai do saldo total
53       } // fim do método debit
54
55       // retorna o número da conta
56       public int getAccountNumber()
57       {
58          return accountNumber;
59       } // fim do método getAccountNumber
60    } // fim da classe Account
```

Figura 13.18 | A classe `Account` representa uma conta bancária.

A classe `Account` contém um construtor (linhas 12–19) que recebe um número de conta, o PIN estabelecido para a conta, o saldo inicial disponível da conta e o saldo inicial total da conta como argumentos. As linhas 15–18 atribuem esses valores aos atributos da classe (isto é, campos).

O método `validatePIN` (linhas 22–28) determina se um PIN especificado pelo usuário (isto é, o parâmetro `userPIN`) corresponde ao PIN associado com a conta (isto é, o atributo `pin`). Lembre-se de que modelamos o parâmetro desse método `userPIN` na Figura 12.19. Se os dois PINs corresponderam, o método retornará `true` (linha 25); do contrário, ele retornará `false` (linha 27).

Os métodos `getAvailableBalance` (linhas 31–34) e `getTotalBalance` (linhas 37–40) são métodos *get* que retornam os valores dos atributos `double` `availableBalance` e `totalBalance`, respectivamente.

O método `credit` (linhas 43–46) adiciona uma quantia de dinheiro (isto é, o parâmetro `amount`) a uma `Account` como parte de uma transação de depósito. Note que esse método adiciona o `amount` apenas ao atributo `totalBalance` (linha 45). O dinheiro creditado em uma conta durante um depósito não se torna disponível imediatamente, portanto só modificamos o saldo total. Supomos que o banco atualiza o saldo disponível apropriadamente em um momento posterior. Nossa implementação da classe `Account` inclui somente os métodos necessários para executar as transações no ATM. Portanto, omitimos os métodos que alguns outros sistemas bancários invocariam para adicionar o atributo `availableBalance` (para confirmar um depósito) ou subtrair do atributo `totalBalance` (para rejeitar um depósito).

O método `debit` (linhas 49–53) subtrai uma quantia de dinheiro (isto é, o parâmetro `amount`) de uma `Account` como parte de uma transação de saque. Esse método subtrai o `amount` dos dois atributos `availableBalance` (linha 51) e `totalBalance` (linha 52), porque um saque afeta ambas as medidas do saldo de uma conta.

O método `getAccountNumber` (linhas 56–59) fornece acesso a `accountNumber` de uma `Account`. Incluímos esse método na nossa implementação de modo que um cliente da classe (isto é, `BankDatabase`) possa identificar uma `Account` particular. Por exemplo, `BankDatabase` contém muitos objetos `Account` e pode invocar esse método em cada um dos seus objetos `Account` para localizar aquele com um número de conta específico.

13.4.7 Classe BankDatabase

A classe `BankDatabase` (Figura 13.19) modela o banco de dados do banco com o qual o ATM interage para acessar e modificar informações da conta de um usuário. [*Nota:* estudamos o acesso de banco de dados no Capítulo 28. Por enquanto, modelamos o banco de dados como um array. Um exercício no Capítulo 28 solicita que você reimplemente essa parte do ATM utilizando um banco de dados real.]

```
1  // BankDatabase.java
2  // Representa o banco de dados com as informações sobre as contas bancárias
3
4  public class BankDatabase
5  {
6     private Account[] accounts; // array de Accounts
7
8     // construtor BankDatabase sem argumentos inicializa as contas
9     public BankDatabase()
10    {
11       accounts = new Account[ 2 ]; // apenas 2 contar para teste
12       accounts[ 0 ] = new Account( 12345, 54321, 1000.0, 1200.0 );
13       accounts[ 1 ] = new Account( 98765, 56789, 200.0, 200.0 );
14    } // fim do construtor BankDatabase sem argumentos
15
16    // recupera o objeto Account que contém o número de conta especificado
17    private Account getAccount( int accountNumber )
18    {
19       // faz um loop pelas contas procurando uma correspondência com o número de conta
20       for ( Account currentAccount : accounts )
21       {
22          // retorna a conta atual se uma correspondência for localizada
23          if ( currentAccount.getAccountNumber() == accountNumber )
24             return currentAccount;
```

```
25          } // for final
26
27          return null; // se nenhuma correspondência com uma conta foi localizada, retorna null
28       } // fim do método getAccount
29
30       // determina se número da conta e PIN especificados pelo usuário correspondem
31       // àqueles de uma conta no banco de dados
32       public boolean authenticateUser( int userAccountNumber, int userPIN )
33       {
34          // tenta recuperar a conta com o número da conta
35          Account userAccount = getAccount( userAccountNumber );
36
37          // se a conta existir, retorna o resultado do método validatePIN de Account
38          if ( userAccount != null )
39             return userAccount.validatePIN( userPIN );
40          else
41             return false; // número de conta não foi localizado, portanto retorna false
42       } // fim do método authenticateUser
43
44       // retorna o saldo disponível de Account com o número da conta especificado
45       public double getAvailableBalance( int userAccountNumber )
46       {
47          return getAccount( userAccountNumber ).getAvailableBalance();
48       } // fim do método getAvailableBalance
49
50       // retorna o saldo total de Account com o número da conta especificado
51       public double getTotalBalance( int userAccountNumber )
52       {
53          return getAccount( userAccountNumber ).getTotalBalance();
54       } // fim do método getTotalBalance
55
56       // credita uma quantia a Account com o número da conta especificado
57       public void credit( int userAccountNumber, double amount )
58       {
59          getAccount( userAccountNumber ).credit( amount );
60       } // fim do método credit
61
62       // debita uma quantidade da Account com o número da conta especificado
63       public void debit( int userAccountNumber, double amount )
64       {
65          getAccount( userAccountNumber ).debit( amount );
66       } // fim do método debit
67    } // fim da classe BankDatabase
```

Figura 13.19 | A classe `BankDatabase` representa o banco de dados com as informações sobre as contas do banco.

Determinamos um atributo do tipo por referência para a classe `BankDatabase` com base no seu relacionamento de composição com a classe `Account`. Como vimos na Figura 13.9, um `BankDatabase` é composto de zero ou mais objetos da classe `Account`. A linha 6 implementa o atributo `accounts` — um array de objetos `Account` — para implementar esse relacionamento de composição. A classe `BankDatabase` tem um construtor sem argumentos (linhas 9–14) que inicializa `accounts` para que elas contenham um conjunto de novos objetos `Account`. Em consideração ao teste do sistema, declaramos `accounts` para conter somente dois elementos no array (linha 11), que instanciamos como os novos objetos `Account` com os dados do teste (linhas 12–13). Observe que o construtor `Account` tem quatro parâmetros — o número de conta, PIN atribuído à conta, o saldo inicial disponível e o saldo inicial total. Lembre-se de que a classe `BankDatabase` serve como um intermediário entre a classe `ATM` e os objetos `Account` reais que contêm informações sobre a conta de um usuário. Portanto, os métodos da classe `BankDatabase` não fazem nada além de invocar os métodos correspondentes do objeto `Account` pertencente ao atual usuário do ATM.

Incluímos o método utilitário `private` de `getAccount` (linhas 17–28) para permitir que a `BankDatabase` obtenha uma referência a um `Account` particular dentro do array de `accounts`. Para localizar a `Account` do usuário, a `BankDatabase` compara o valor retornado pelo método `getAccountNumber` de cada elemento de `accounts` com um número especificado de conta até encontrar uma correspondência. As linhas 20–25 percorrem o array de `accounts`. Se o número de conta da `currentAccount` for igual ao valor do parâmetro `accountNumber`, o método retornará imediatamente a `currentAccount`. Se nenhuma conta tiver o número de conta dado, a linha 27 então retornará `null`.

O método `authenticateUser` (linhas 32–42) aprova ou desaprova a identidade de um usuário do ATM. Esse método recebe um número e PIN da conta especificados pelo usuário como os argumentos e indica se eles correspondem ao número da conta e PIN de uma `Account` no banco de dados. A linha 35 chama o método `getAccount`, que retorna uma `Account` com `userAccountNumber` como seu número de conta ou `null` para indicar que `userAccountNumber` é inválido. Se `getAccount` retornar um objeto `Account`, a linha 39

retornará o valor boolean retornado pelo método validatePIN desse objeto. O método authenticateUser de BankDatabase não executa a comparação com o PIN — em vez disso ele encaminha userPIN ao método validatePIN do objeto Account para fazer isso. O valor retornado pelo método validatePIN de Account indica se o PIN especificado pelo usuário corresponde ao PIN da Account do usuário, portanto o método authenticateUser simplesmente retorna esse valor ao cliente da classe (isto é, ATM).

O método BankDatabase confia na classe ATM para invocar o método authenticateUser e receber um valor de retorno de true antes de permitir que o usuário realize as transações. BankDatabase também confia no fato de que cada objeto Transaction criado pela ATM contém o número de conta válida do atual usuário autenticado e que isso é o número de conta passado para os métodos BankDatabase remanescentes como o argumento userAccountNumber. Os métodos getAvailableBalance (linhas 45–48), getTotalBalance (linhas 51–54), credit (linhas 57–60) e debit (linhas 63–66), portanto, simplesmente recuperam o objeto Account do usuário com o método utilitário getAccount, então invocam o método Account apropriado nesse objeto. Sabemos que as chamadas a getAccount a partir desses métodos nunca retornarão null, porque userAccountNumber deve referenciar um Account existente. Os métodos getAvailableBalance e getTotalBalance retornam os valores retornados pelos métodos Account correspondentes. Também observe que credit e debit simplesmente redirecionam o parâmetro amount aos métodos Account que eles invocam.

13.4.8 Classe Transaction

A classe Transaction (Figura 13.20) é uma superclasse abstrata que representa a noção de uma transação no ATM. Ela contém os recursos comuns das subclasses BalanceInquiry, Withdrawal e Deposit. Essa classe expande o código "esqueleto" desenvolvido inicialmente na Seção 13.3. A linha 4 declara essa classe como abstract. As linhas 6–8 declaram os atributos private da classe. A partir do diagrama de classe da Figura 13.10, lembre-se de que a classe Transaction tem um atributo accountNumber (linha 6) que indica a conta envolvida na Transaction. Derivamos os atributos screen (linha 7) e bankDatabase (linha 8) das associações da classe Transaction modeladas na Figura 13.9 — todas as transações requerem acesso à tela do ATM e ao banco de dados do banco.

```
1  // Transaction.java
2  // A superclasse abstrata Transaction representa uma transação no ATM
3
4  public abstract class Transaction
5  {
6     private int accountNumber; // indica conta envolvida
7     private Screen screen; // Tela do ATM
8     private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
9
10    // Construtor de Transaction invocado pelas subclasses utilizando super()
11    public Transaction( int userAccountNumber, Screen atmScreen,
12       BankDatabase atmBankDatabase )
13    {
14       accountNumber = userAccountNumber;
15       screen = atmScreen;
16       bankDatabase = atmBankDatabase;
17    } // fim do construtor de Transaction
18
19    // retorna o número de conta
20    public int getAccountNumber()
21    {
22       return accountNumber;
23    } // fim do método getAccountNumber
24
25    // retorna a referência à tela
26    public Screen getScreen()
27    {
28       return screen;
29    } // fim do método getScreen
30
31    // retorna a referência ao banco de dados da instituição financeira
32    public BankDatabase getBankDatabase()
33    {
34       return bankDatabase;
35    } // fim do método getBankDatabase
36
37    // realiza a transação (sobrescrita por cada subclasse)
38    abstract public void execute();
39 } // fim da classe Transaction
```

Figura 13.20 | A superclasse abstrata Transaction representa uma transação no ATM.

A classe `Transaction` tem um construtor (linhas 11–17) que recebe como argumentos o número atual da conta do usuário e se refere à tela do ATM e ao banco de dados do banco. Como `Transaction` é uma classe abstrata, esse construtor só será chamado pelos construtores das subclasses `Transaction`.

A classe tem três métodos *get* `public` — `getAccountNumber` (linhas 20–23), `getScreen` (linhas 26–29) e `getBankDatabase` (linhas 32–35). Estes são herdados pelas subclasses `Transaction` e utilizados para ganhar acesso aos atributos `private` da classe `Transaction`.

A classe `Transaction` também declara o método `abstract execute` (linha 38). Não faz sentido fornecer uma implementação desse método, pois uma transação genérica não pode ser executada. Assim, declaramos esse método como `abstract` e forçamos cada subclasse `Transaction` a fornecer uma implementação concreta que executa esse tipo de transação em particular.

13.4.9 Classe `BalanceInquiry`

A classe `BalanceInquiry` (Figura 13.21) estende a classe `Transaction` e representa uma transação no ATM de consulta de saldos. `BalanceInquiry` não contém nenhum atributo próprio, mas ela herda os atributos `accountNumber`, `screen` e `bankDatabase` de `Transaction`, acessíveis por meio dos métodos *get* `public` da classe `Transaction`. O construtor `BalanceInquiry` recebe os argumentos correspondentes a esses atributos e simplesmente os encaminha ao construtor da `Transaction` utilizando `super` (linha 10).

A classe `BalanceInquiry` sobrescreve o método abstrato `execute` da classe `Transaction` para fornecer uma implementação concreta (linhas 14–36) que realiza os procedimentos envolvidos em uma consulta de saldos. As linhas 18–19 obtêm as referências ao banco de dados do banco e à tela do ATM invocando os métodos herdados da superclasse `Transaction`. As linhas 22–23 recuperam o saldo disponível da conta envolvida invocando o método `getAvailableBalance` de `bankDatabase`. Observe que a linha 23 utiliza o método herdado `getAccountNumber` para obter o número da conta do usuário atual, que ela então passa para `getAvailableBalance`. As linhas 26–27 recuperam o saldo total da conta do usuário atual. As linhas 30–35 exibem as informações sobre o saldo na tela do ATM. Lembre-se de que `displayDollarAmount` recebe um argumento `double` e gera a saída dele na tela formatado como uma quantia em dólares. Por exemplo, se `availableBalance` de um usuário for `1000.5`, a linha 32 irá gerar a saída de `$1,000.50`. Observe que a linha 35 insere uma linha em branco da saída para separar as informações do saldo da saída subsequente (isto é, o menu principal repetido pela classe `ATM` depois de executar a `BalanceInquiry`).

```
1  // BalanceInquiry.java
2  // Representa uma transação de consulta de saldos no ATM
3
4  public class BalanceInquiry extends Transaction
5  {
6     // Construtor de BalanceInquiry
7     public BalanceInquiry( int userAccountNumber, Screen atmScreen,
8        BankDatabase atmBankDatabase )
9     {
10       super( userAccountNumber, atmScreen, atmBankDatabase );
11    } // fim do construtor de BalanceInquiry
12
13    // realiza a transação
14    @Override
15    public void execute()
16    {
17       // obtém as referências ao banco de dados e tela do banco
18       BankDatabase bankDatabase = getBankDatabase();
19       Screen screen = getScreen();
20
21       // obtém o saldo disponível da conta envolvida
22       double availableBalance =
23          bankDatabase.getAvailableBalance( getAccountNumber() );
24
25       // obtém o saldo total da conta envolvida
26       double totalBalance =
27          bankDatabase.getTotalBalance( getAccountNumber() );
28
29       // exibe as informações sobre o saldo na tela
30       screen.displayMessageLine( "\nBalance Information:" );
31       screen.displayMessage( " - Available balance: " );
32       screen.displayDollarAmount( availableBalance );
33       screen.displayMessage( "\n - Total balance:     " );
34       screen.displayDollarAmount( totalBalance );
35       screen.displayMessageLine( "" );
36    } // fim do método execute
37 } // fim da classe BalanceInquiry
```

Figura 13.21 | A classe `BalanceInquiry` representa uma transação de consulta de saldo no ATM.

13.4.10 Classe `Withdrawal`

A classe `Withdrawal` (Figura 13.22) estende `Transaction` e representa uma transação de saque no ATM. Essa classe é expandida sobre o "esqueleto" de código para essa classe desenvolvida na Figura 13.12. Como vimos no diagrama de classe da Figura 13.10, a classe `Withdrawal` tem um atributo, `amount`, que a linha 6 implementa como um campo `int`. A Figura 13.9 modela as associações entre a classe `Withdrawal` e as classes `Keypad` e `CashDispenser`, com as quais as linhas 7–8 implementam os atributos `keypad` e `cashDispenser`, respectivamente, do tipo por referência. A linha 11 declara uma constante que corresponde à opção de cancelamento no menu. Discutiremos mais adiante como a classe utiliza essa constante.

```
1  // Withdrawal.java
2  // Representa uma transação de saque no ATM
3
4  public class Withdrawal extends Transaction
5  {
6     private int amount; // quantia a sacar
7     private Keypad keypad; // referência ao teclado numérico
8     private CashDispenser cashDispenser; // referência ao dispensador de cédulas
9
10    // constante que corresponde à opção cancelar no menu
11    private final static int CANCELED = 6;
12
13    // Construtor de Withdrawal
14    public Withdrawal( int userAccountNumber, Screen atmScreen,
15       BankDatabase atmBankDatabase, Keypad atmKeypad,
16       CashDispenser atmCashDispenser )
17    {
18       // inicializa as variáveis da superclasse
19       super( userAccountNumber, atmScreen, atmBankDatabase );
20
21       // inicializa as referências ao teclado numérico e ao dispensador de cédulas
22       keypad = atmKeypad;
23       cashDispenser = atmCashDispenser;
24    } // fim do construtor de Withdrawal
25
26    // realiza a transação
27    @Override
28    public void execute()
29    {
30       boolean cashDispensed = false; // cédulas ainda não foram entregues
31       double availableBalance; // quantia disponível para saque
32
33       // obtém as referências ao banco de dados e tela do banco
34       BankDatabase bankDatabase = getBankDatabase();
35       Screen screen = getScreen();
36
37       // faz um loop até as cédulas serem entregues ou o usuário cancelar
38       do
39       {
40          // obtém a quantia de um saque escolhida pelo usuário
41          amount = displayMenuOfAmounts();
42
43          // verifica se o usuário escolheu uma quantia de saque ou cancelou
44          if ( amount != CANCELED )
45          {
46             // obtém o saldo disponível na conta envolvida
47             availableBalance =
48                bankDatabase.getAvailableBalance( getAccountNumber() );
49
50             // verifica se o usuário tem dinheiro suficiente na conta
51             if ( amount <= availableBalance )
52             {
53                // verifica se o dispensador de cédulas tem cédulas suficientes
54                if ( cashDispenser.isSufficientCashAvailable( amount ) )
55                {
56                   // atualiza a conta envolvida para refletir a retirada/saque
57                   bankDatabase.debit( getAccountNumber(), amount );
58
```

```
59                   cashDispenser.dispenseCash( amount ); // entrega cédulas
60                   cashDispensed = true; // cédulas foram entregues
61
62                   // instrui o usuário a pegar as cédulas
63                   screen.displayMessageLine( "\nYour cash has been" +
64                      " dispensed. Please take your cash now." );
65                } // fim do if
66                else // o dispensador de cédulas não tem cédulas suficientes
67                   screen.displayMessageLine(
68                      "\nInsufficient cash available in the ATM." +
69                      "\n\nPlease choose a smaller amount." );
70             } // fim do if
71             else // não há dinheiro suficiente disponível na conta do usuário
72             {
73                screen.displayMessageLine(
74                   "\nInsufficient funds in your account." +
75                   "\n\nPlease choose a smaller amount." );
76             } // fim de else
77          } // fim do if
78          else // o usuário escolheu a opção cancelar no menu
79          {
80             screen.displayMessageLine( "\nCanceling transaction..." );
81             return; // retorna ao menu principal porque o usuário cancelou
82          } // fim de else
83       } while ( !cashDispensed );
84
85    } // fim do método execute
86
87    // exibe um menu de quantias de saques e a opção para cancelar;
88    // retorna a quantia escolhida ou 0 se o usuário escolher cancelar
89    private int displayMenuOfAmounts()
90    {
91       int userChoice = 0; // variável local para armazenar o valor de retorno
92
93       Screen screen = getScreen(); // obtém referência de tela
94
95       // array de quantias que correspondem aos números no menu
96       int[] amounts = { 0, 20, 40, 60, 100, 200 };
97
98       // faz um loop enquanto nenhuma escolha válida for feita
99       while ( userChoice == 0 )
100      {
101         // exibe o menu
102         screen.displayMessageLine( "\nWithdrawal Menu:" );
103         screen.displayMessageLine( "1 - $20" );
104         screen.displayMessageLine( "2 - $40" );
105         screen.displayMessageLine( "3 - $60" );
106         screen.displayMessageLine( "4 - $100" );
107         screen.displayMessageLine( "5 - $200" );
108         screen.displayMessageLine( "6 - Cancel transaction" );
109         screen.displayMessage( "\nChoose a withdrawal amount: " );
110
111         int input = keypad.getInput(); // obtém a entrada de usuário pelo teclado
112
113         // determina como prosseguir com base no valor de entrada
114         switch ( input )
115         {
116            case 1: // se o usuário escolheu uma quantia de saque
117            case 2: // (isto é, escolheu a opção 1, 2, 3, 4 ou 5), retorna a
118            case 3: // quantia correspondente do array de quantias
119            case 4:
120            case 5:
121               userChoice = amounts[ input ]; // salva a escolha do usuário
122               break;
123            case CANCELED: // o usuário escolheu cancelar
124               userChoice = CANCELED; // salva a escolha do usuário
125               break;
126            default: // o usuário não inseriu um valor ente 1 e 6
127               screen.displayMessageLine(
```

```
128                    "\nInvalid selection. Try again." );
129             } // fim do switch
130          } // fim do while
131
132          return userChoice; // retorna a quantia de saque ou CANCELADA
133       } // fim do método displayMenuOfAmounts
134    } // fim da classe Withdrawal
```

Figura 13.22 | A classe `Withdrawal` representa uma transação de saque no ATM.

O construtor da classe `Withdrawal` (linhas 14–24) tem cinco parâmetros. Ele utiliza `super` para passar os parâmetros `userAccountNumber`, `atmScreen` e `atmBankDatabase` para o construtor da superclasse `Transaction` a fim de configurar os atributos que `Withdrawal` herda de `Transaction`. O construtor também recebe as referências `atmKeypad` e `atmCashDispenser` como parâmetros e as especifica para os atributos `keypad` e `cashDispenser` do tipo por referência.

A classe `Withdrawal` sobrescreve o método `Transaction` `execute` com uma implementação concreta (linhas 27–85) que executa os passos de uma retirada. A linha 30 declara e inicializa uma variável `boolean` `cashDispensed` local, que indica se o dinheiro foi entregue (isto é, se a transação completou com sucesso) e inicialmente é `false`. A linha 31 declara a variável `double` `availableBalance` local, que armazenará o saldo disponível do usuário durante uma transação de saque. As linhas 34–35 obtêm as referências ao banco de dados do banco e à tela do ATM invocando os métodos herdados da superclasse `Transaction`.

As linhas 38–83 contêm uma instrução `do...while` que executa seu corpo até que cédulas sejam entregues (isto é, até `cashDispensed` tornar-se `true`) ou até o usuário escolher cancelar (nesse caso, o loop termina). Utilizamos esse loop continuamente para retornar o usuário ao início da transação se ocorrer um erro (isto é, a quantia de saque solicitada é maior do que o saldo disponível do usuário ou maior do que a quantia de cédulas no dispensador de cédulas). A linha 41 exibe um menu das quantias de saques e obtém uma seleção de usuário chamando o método utilitário `private` `displayMenuOfAmounts` (declarado nas linhas 89–133). Esse método exibe o menu das quantias e retorna uma quantia de saque `int` ou uma constante `int` `CANCELED` para indicar que o usuário optou por cancelar a transação.

O método `displayMenuOfAmounts` (linhas 89–133) primeiro declara a variável local `userChoice` (inicialmente `0`) para armazenar o valor que o método retornará (linha 91). A linha 93 obtém uma referência à tela chamando o método `getScreen` herdado da superclasse `Transaction`. A linha 96 declara um array de inteiros das quantias de saque que correspondem às quantias exibidas no menu de saque. Ignoramos o primeiro elemento no array (índice `0`) porque o menu não tem nenhuma opção 0. A instrução `while` nas linhas 99–130 é repetida até `userChoice` assumir um valor além de `0`. Veremos mais adiante que isso ocorre quando o usuário faz uma seleção válida no menu. As linhas 102–109 exibem o menu de saque na tela e solicitam ao usuário para inserir uma escolha. A linha 111 obtém o inteiro de `input` pelo teclado. A instrução `switch` nas linhas 114–129 determina como prosseguir com base na entrada do usuário. Se o usuário selecionar um número entre `1` e `5`, a linha 121 configura `userChoice` como o valor do elemento em `amounts` no índice `input`. Por exemplo, se o usuário inserir `3` para sacar US$ 60, a linha 121 configura `userChoice` como o valor de `amounts[ 3 ]` (isto é, `60`). A linha 122 termina a instrução `switch`. A variável `userChoice` não mais é igual a `0`, assim o `while` nas linhas 99–130 termina e a linha 132 retorna `userChoice`. Se o usuário selecionar a opção cancelar no menu, as linhas 124–125 executam, configurando `userChoice` como `CANCELED` e fazendo com que o método retorne esse valor. Se o usuário não inserir uma seleção válida de menu, as linhas 127–128 exibem uma mensagem de erro e o usuário é retornado ao menu de saque.

A linha 44 no método `execute` determina se o usuário selecionou um valor de retirada ou optou por cancelar. Se o usuário cancelar, as linhas 80–81 executam e exibem uma mensagem apropriada para o usuário antes de retornar o controle ao método de chamada (isto é, o método `performTransactions` da classe `ATM`). Se o usuário escolheu uma quantia de saque, as linhas 47–48 recuperam o saldo disponível da `Account` do usuário atual e o armazenam na variável `availableBalance`. Em seguida, a instrução `if` na linha 51 determina se a quantia selecionada é menor ou igual ao saldo disponível do usuário. Se não for, as linhas 73–75 exibem uma mensagem de erro apropriada. O controle então continua até o final da `do...while` e o loop se repetirá, pois `cashDispensed` ainda é `false`. Se o saldo do usuário for suficientemente alto, a instrução `if` na linha 54 determina se o dispensador de cédulas tem cédulas suficientes para satisfazer a solicitação de saque invocando o método `isSufficientCashAvailable` de `cashDispenser`. Se esse método retornar `false`, as linhas 67–69 exibirão uma mensagem de erro apropriada e a `do...while` é repetida. Se houver cédulas suficientes disponíveis, então os requisitos para o saque serão atendidos e a linha 57 debita `amount` da conta do usuário no banco de dados. As linhas 59–60 então instruem o dispensador de cédulas a entregar as cédulas ao usuário e configuram `cashDispensed` como `true`. Por fim, as linhas 63–64 exibem uma mensagem para o usuário de que as cédulas foram entregues. Como `cashDispensed` agora é `true`, o controle continua depois de `do...while`. Nenhuma instrução adicional aparece abaixo do loop, portanto o método retorna o controle à classe `ATM`.

13.4.11 Classe `Deposit`

A classe `Deposit` (Figura 13.23) estende `Transaction` e representa uma transação de depósito. Lembre-se da Figura 13.10 de que a classe `Deposit` tem um atributo `amount`, que a linha 6 implementa como um campo `int`. As linhas 7–8 criam atributos de tipo por referência `keypad` e `depositSlot` que implementam as associações entre a classe `Deposit` e as classes `Keypad` e `DepositSlot` modeladas na Figura 13.9. A linha 9 declara uma constante `CANCELED` que corresponde ao valor que um usuário insere para cancelar. Discutiremos mais adiante como a classe utiliza essa constante.

```
1  // Deposit.java
2  // Representa uma transação de depósito no ATM
3
4  public class Deposit extends Transaction
5  {
6     private double amount; // quantia a depositar
7     private Keypad keypad; // referência ao teclado numérico
8     private DepositSlot depositSlot; // referência à abertura para depósito
9     private final static int CANCELED = 0; // constante para a opção de cancelamento
10
11    // Construtor de Deposit
12    public Deposit( int userAccountNumber, Screen atmScreen,
13       BankDatabase atmBankDatabase, Keypad atmKeypad,
14       DepositSlot atmDepositSlot )
15    {
16       // inicializa as variáveis da superclasse
17       super( userAccountNumber, atmScreen, atmBankDatabase );
18
19       // inicializa as referências a teclado e abertura para depósito
20       keypad = atmKeypad;
21       depositSlot = atmDepositSlot;
22    } // fim do construtor de Deposit
23
24    // realiza a transação
25    @Override
26    public void execute()
27    {
28       BankDatabase bankDatabase = getBankDatabase(); // obtém a referência
29       Screen screen = getScreen(); // obtém a referência
30
31       amount = promptForDepositAmount(); // obtém a quantia de depósito do usuário
32
33       // verifica se usuário inseriu uma quantia de depósito ou cancelou
34       if ( amount != CANCELED )
35       {
36          // solicita o envelope de depósito contendo a quantia especificada
37          screen.displayMessage(
38             "\nPlease insert a deposit envelope containing " );
39          screen.displayDollarAmount( amount );
40          screen.displayMessageLine( "." );
41
42          // recebe o envelope de depósito
43          boolean envelopeReceived = depositSlot.isEnvelopeReceived();
44
45          // verifica se envelope de depósito foi recebido
46          if ( envelopeReceived )
47          {
48             screen.displayMessageLine( "\nYour envelope has been " +
49                "received.\nNOTE: The money just deposited will not " +
50                "be available until we verify the amount of any " +
51                "enclosed cash and your checks clear." );
52
53             // credita na conta para refletir o depósito
54             bankDatabase.credit( getAccountNumber(), amount );
55          } // fim do if
56          else // envelope de depósito não foi recebido
57          {
58             screen.displayMessageLine( "\nYou did not insert an " +
59                "envelope, so the ATM has canceled your transaction." );
60          } // fim de else
61       } // fim do if
62       else // o usuário cancelou em vez de inserir uma quantia
63       {
64          screen.displayMessageLine( "\nCanceling transaction..." );
65       } // fim de else
66    } // fim do método execute
```

```
67
68       // solicita que o usuário insira uma quantia de depósito em centavos
69       private double promptForDepositAmount()
70       {
71          Screen screen = getScreen(); // obtém a referência à tela
72
73          // exibe a solicitação
74          screen.displayMessage( "\nPlease enter a deposit amount in " +
75             "CENTS (or 0 to cancel): " );
76          int input = keypad.getInput(); // recebe a entrada da quantia do depósito
77
78          // verifica se o usuário cancelou ou inseriu uma quantia válida
79          if ( input == CANCELED )
80             return CANCELED;
81          else
82          {
83             return ( double ) input / 100; // retorna a quantia em dólares
84          } // fim de else
85       } // fim do método promptForDepositAmount
86    } // fim da classe Deposit
```

Figura 13.23 | A classe `Deposit` representa uma transação de depósito no ATM.

Como ocorre com a classe `Withdrawal`, a classe `Deposit` contém um construtor (linhas 12–22) que passa três parâmetros para o construtor da superclasse `Transaction`. O construtor também tem parâmetros `atmKeypad` e `atmDepositSlot`, que ele atribui aos atributos correspondentes (linhas 20–21).

O método `execute` (linhas 25–66) sobrescreve a versão `abstract` na superclasse `Transaction` com uma implementação concreta que realiza os procedimentos necessários em uma transação de depósito. As linhas 28–29 obtêm as referências ao banco de dados e à tela. A linha 31 solicita que o usuário insira uma quantia de depósito invocando o método utilitário `private promptForDepositAmount` (declarado nas linhas 69–85) e configura o atributo `amount` como o valor retornado. O método `promptForDepositAmount` solicita que o usuário insira uma quantia de depósito como um número inteiro dos centavos (visto que o teclado numérico do ATM não contém um ponto de fração decimal; isso é compatível com muitos ATMS reais) e retorna o valor de `double` que representa a quantia em dólares a ser depositada.

A linha 71 no método `promptForDepositAmount` obtém uma referência à tela do ATM. As linhas 74–75 exibem uma mensagem na tela que solicita ao usuário para inserir uma quantia de depósito como um número de centavos ou "0" para cancelar a transação. A linha 76 recebe a entrada do usuário do teclado. A instrução `if` nas linhas 79–84 determina se o usuário inseriu uma quantia de depósito real ou optou por cancelar. Se o último, a linha 80 retorna a constante `CANCELED`. Caso contrário, a linha 83 retorna a quantia de depósito depois de converter do número de centavos para uma quantia em dólares fazendo uma coerção de `input` para um `double` e então dividindo por `100`. Por exemplo, se o usuário inserir `125` como o número de centavos, a linha 83 retornará `125.0` dividido por `100` ou `1.25` — 125 centavos é US$ 1,25.

As linhas 34–65 no método `execute` determinam se o usuário optou por cancelar a transação em vez de inserir um valor de depósito. Se o usuário cancelar, a linha 64 exibirá uma mensagem apropriada e o método retorna. Se o usuário inserir uma quantia de depósito, as linhas 37–40 irão instruir o usuário a inserir um envelope de depósito com a quantia correta. Lembre-se de que o método `displayDollarAmount` de `Screen` gera a saída de um `double` formatado como uma quantia em dólares.

A linha 43 configura uma variável `boolean` local como o valor retornado pelo método `isEnvelopeReceived` de `depositSlot`, indicando se um envelope de depósito foi recebido. Lembre-se de que codificamos o método `isEnvelopeReceived` (linhas 8–11 da Figura 13.17) para sempre retornar `true`, pois estamos simulando a funcionalidade da abertura para depósito e supomos que o usuário sempre insere um envelope. Entretanto, codificamos o método `execute` da classe `Deposit` para testar a possibilidade de que o usuário não insira um envelope — boa engenharia de software demanda que os programas são responsáveis por todos os possíveis valores de retorno. Portanto, a classe `Deposit` está preparada para futuras versões do `isEnvelopeReceived` que poderiam retornar `false`. As linhas 48–54 executam se a abertura de depósito receber um envelope. As linhas 48–51 exibem uma mensagem apropriada para o usuário. A linha 54 então credita a quantia de depósito na conta do usuário no banco de dados. As linhas 58–59 executarão se a abertura de depósito não receber um envelope de depósito. Nesse caso, exibimos uma mensagem ao usuário indicando que o ATM cancelou a transação. O método então retorna sem modificar a conta do usuário.

13.4.12 Classe ATMCaseStudy

A classe `ATMCaseStudy` (Figura 13.24) é uma classe simples que permite iniciar ou "ativar" o ATM e testar a implementação de nosso modelo do sistema ATM. O método `main` da classe `ATMCaseStudy` (linhas 7–11) não faz nada além de instanciar um novo objeto `ATM` chamado `theATM` (linha 9) e invoca seu método `run` (linha 10) para iniciar o ATM.

```
1  // ATMCaseStudy.java
2  // Programa de driver para o estudo de caso do ATM
3
4  public class ATMCaseStudy
5  {
6     // método main cria e executa o ATM
7     public static void main( String[] args )
8     {
9        ATM theATM = new ATM();
10       theATM.run();
11    } // fim de main
12 } // fim da classe ATMCaseStudy
```

Figura 13.24 | `ATMCaseStudy.java` inicia o ATM.

13.5 Conclusão

Neste capítulo, você utilizou a herança para ajustar o projeto do sistema de software de um ATM e também implementou completamente o ATM no Java. Parabéns por completar todo o estudo de caso do ATM! Esperamos que você tenha achado essa experiência valiosa e que ela tenha reforçado muitos dos conceitos da programação orientada a objetos que você aprendeu. No próximo capítulo, faremos um exame mais detalhado das interfaces gráficas com o usuário (GUIs).

Respostas dos exercícios de autorrevisão

13.1 Verdadeiro. O sinal de subtração (–) indica visibilidade privada.

13.2 b.

13.3 O design da classe `Keypad` produz o código na Figura 13.25. Lembre-se de que a classe `Keypad` não tem nenhum atributo por enquanto, mas os atributos podem tornar-se aparentes à medida que continuamos a implementação. Também observe que se fôssemos projetar um ATM real, o método `getInput` precisaria interagir com o hardware do teclado do ATM. Na verdade, nossa entrada virá do teclado de um computador pessoal ao escrevermos o código Java completo na Seção 13.4.

13.4 b.

13.5 Falso. A UML requer que os nomes das classes abstratas e de métodos sejam escritos em itálico.

13.6 O design da classe `Transaction` produz o código na Figura 13.26. Os corpos do construtor de classe e métodos são completados na Seção 13.4. Quando totalmente implementados, os métodos `getScreen` e `getBankDatabase` retornarão os atributos de referência `screen` e `bankDatabase` `private`, respectivamente, da superclasse `Transaction`. Esses métodos permitem que as subclasses `Transaction` acessem a tela do ATM e interajam com o banco de dados do banco.

```
1  // A classe Keypad representa o teclado de um ATM
2  public class Keypad
3  {
4     // nenhum atributo foi especificado ainda
5
6     // construtor sem argumentos
7     public Keypad()
8     {
9     } // fim do construtor Keypad sem argumentos
10
11    // operações
12    public int getInput()
13    {
14    } // fim do método getInput
15 } // fim da classe Keypad
```

Figura 13.25 | Código Java para a classe `Keypad` baseado nas figuras 13.1 e 13.2.

```
1  // Classe abstrata Transaction representa uma transação do ATM
2  public abstract class Transaction
3  {
4     // atributos
5     private int accountNumber; // indica conta envolvida
6     private Screen screen; // Tela do ATM
```

```
7       private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
8
9       // construtor sem argumentos invocado pelas subclasses utilizando super()
10      public Transaction()
11      {
12      } // fim do construtor Transaction sem argumentos
13
14      // retorna o número de conta
15      public int getAccountNumber()
16      {
17      } // fim do método getAccountNumber
18
19      // retorna a referência à tela
20      public Screen getScreen()
21      {
22      } // fim do método getScreen
23
24      // retorna a referência ao banco de dados da instituição financeira
25      public BankDatabase getBankDatabase()
26      {
27      } // fim do método getBankDatabase
28
29      // método abstrato sobrescrito pelas subclasses
30      public abstract void execute();
31  } // fim da classe Transaction
```

Figura 13.26 | Código Java para a classe `Transaction` baseado nas figuras 13.9 e 13.10.

14

Você acha que posso ouvir essas coisas o dia todo?
— Lewis Carroll

Mesmo um pequeno evento na vida de uma criança é um evento no mundo da criança e, portanto, um evento do mundo.
— Gaston Bachelard

Você paga e faz a sua escolha.
— Punch

Componentes GUI: Parte 1

Objetivos

Neste capítulo, você aprenderá:

- Os princípios do projeto de interfaces gráficas com o usuário (Graphical User Interfaces — GUIs).
- Como utilizar a nova e elegante interface Nimbus do Java compatível com várias plataformas.
- A construir GUIs e tratar eventos gerados por interações de usuário com GUIs.
- A entender os pacotes que contêm componentes GUI, interfaces e classes de tratamento de evento.
- A criar e manipular botões, rótulos, listas, campos de texto e painéis.
- A tratar eventos de mouse e eventos de teclado.
- A utilizar gerenciadores de layout para organizar componentes GUI.

Sumário

14.1 Introdução
14.2 A nova interfacedo Nimbus do Java
14.3 Entrada/saída baseada em GUI simples com `JOptionPane`
14.4 Visão geral de componentes Swing
14.5 Exibição de texto e imagens em uma janela
14.6 Campos de texto e uma introdução ao tratamento de evento com classes aninhadas
14.7 Tipos comuns de eventos GUI e interfaces ouvintes
14.8 Como o tratamento de evento funciona
14.9 `JButton`
14.10 Botões que mantêm o estado
14.10.1 `JCheckBox`
14.10.2 `JRadioButton`
14.11 `JComboBox` e uso de uma classe interna anônima para tratamento de evento
14.12 `JList`
14.13 Listas de seleção múltipla
14.14 Tratamento de evento de mouse
14.15 Classes adaptadoras
14.16 Subclasse `JPanel` para desenhar com o mouse
14.17 Tratamento de evento-chave
14.18 Introdução a gerenciadores de layout
14.18.1 `FlowLayout`
14.18.2 `BorderLayout`
14.18.3 `GridLayout`
14.19 Utilizando painéis para gerenciar layouts mais complexos
14.20 `JTextArea`
14.21 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

14.1 Introdução

Uma **interface gráfica com usuário (*Graphical User Interface — GUI*)** apresenta um mecanismo amigável ao usuário para interagir com um aplicativo. Uma GUI (pronuncia-se "Gui") dá ao aplicativo uma "aparência" e "comportamento" distintos. Fornecer aos diferentes aplicativos componentes de interface com o usuário consistentes e intuitivos permite que os usuários se familiarizem com um novo aplicativo, para que possam aprendê-lo mais rapidamente e utilizá-lo mais produtivamente.

Observações sobre a aparência e comportamento 14.1

Interfaces com o usuário consistentes permitem que ele aprenda mais rápido novos aplicativos.

As GUIs são construídas a partir de **componentes GUI**. Esses são às vezes chamados **controles** ou **widgets** — abreviação de **window gadgets**. Um componente GUI é um objeto com que o usuário interage via mouse, teclado ou outro formulário de entrada, como reconhecimento de voz. Neste capítulo e no Capítulo 25, "Componentes GUI: Parte 2", você aprenderá sobre muitos componentes GUI do Java. Também abordamos outros componentes GUI uma vez que são necessários nas demais partes do livro.

Muitos IDEs fornecem ferramentas de projeto GUI nas quais é possível especificar o tamanho e a localização exatos de um componente de maneira visual utilizando o mouse. O IDE gera o código GUI para você. Embora isso simplifique muito a criação de GUIs, cada IDE tem diferentes capacidades e gera o código distinto. Por essa razão, escrevemos o código GUI à mão.

Como exemplo de uma GUI, considere a Figura 14.1, que mostra o aplicativo SwingSet3 que está disponível em `download.java.net/javadesktop/swingset3/SwingSet3.jnlp`. Esse aplicativo é uma ótima maneira de navegar por vários componentes GUI fornecidos pelas Swing GUI APIs do Java. Basta clicar em um nome de componente (por exemplo, `JFrame`, `JTabbedPane` etc.) na área GUI Components à esquerda da janela para ver uma demonstração do componente GUI à direita da janela. O código-fonte de cada demo é mostrado na área de texto na parte inferior da janela. Rotulamos alguns dos componentes GUI no aplicativo. Na parte superior da janela há uma **barra de título** que contém o título da janela. Abaixo disso há uma **barra de menus** que contém **menus** (File e View). Na região superior direita da janela está um conjunto de **botões** — em geral, usuários pressionam botões para realizar tarefas. Na área GUI Components da janela, há uma **caixa de combinação**; o usuário pode clicar na seta para baixo no lado direito da caixa para selecionar uma lista de itens. Os menus, botões e caixa de combinação fazem parte da GUI do aplicativo. Eles permitem a interação com o aplicativo.

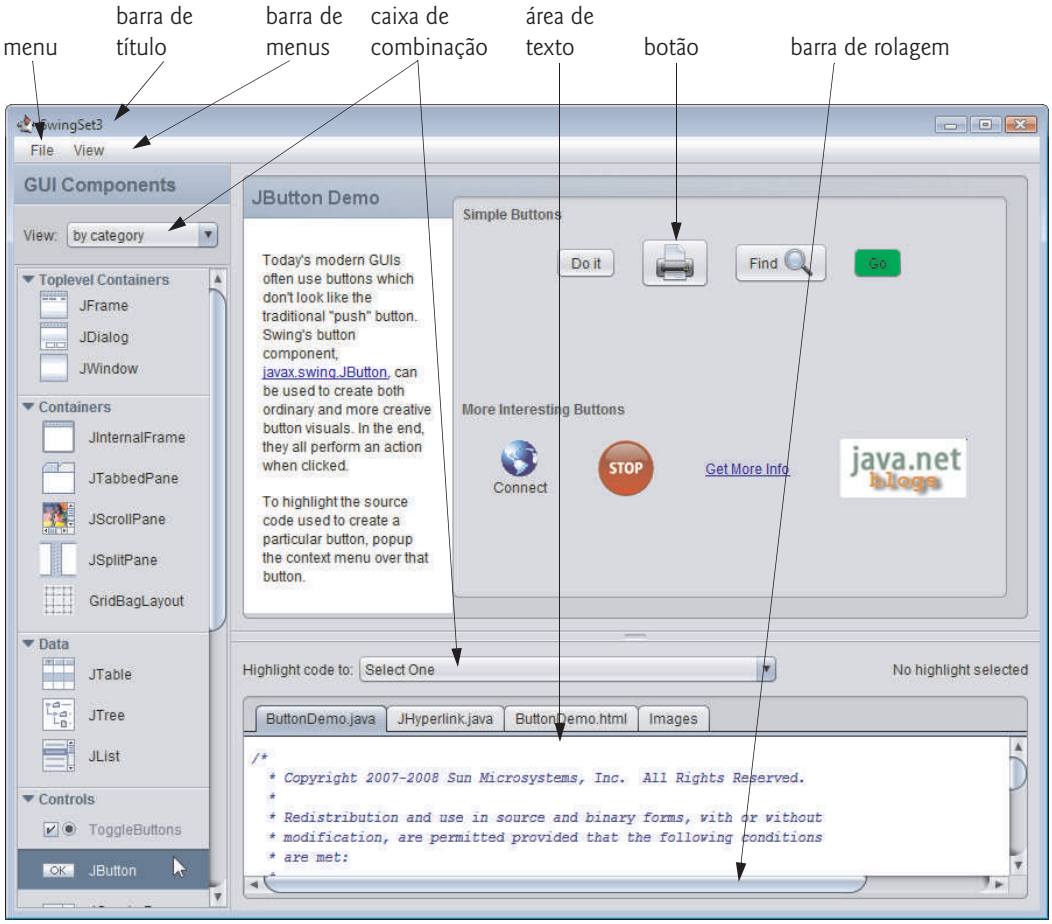


Figura 14.1 | Aplicativo SwingSet3 demonstra muitos dos componentes Swing GUI do Java.

14.2 A nova interface Nimbus do Java

Desde a atualização 10 do Java SE 6, o Java vem com uma interface nova, elegante e compatível com várias plataformas conhecida como **Nimbus**. Para capturas de tela GUI como a mostrada na Figura 14.1, configuramos os nossos sistemas para utilizar o Nimbus como a interface padrão. Há três modos de utilizar o Nimbus:

1. configurá-lo como o padrão de todos os aplicativos Java que executam no computador.
2. configurá-lo como a interface no momento em que se carrega um aplicativo passando um argumento de linha de comando para o comando `java`.
3. configurá-lo como a interface programaticamente no aplicativo (ver Seção 25.6).

Para configurar o Nimbus como o padrão para todos os aplicativos Java, você precisa criar um arquivo de texto chamado `swing.properties` na pasta `lib` tanto da sua pasta de instalação do JDK como da sua pasta de instalação do JRE. Insira a seguinte linha do código no arquivo:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

Para informações adicionais sobre a localização dessas pastas de instalação, visite `java.sun.com/javase/6/webnotes/install/index.html`. [*Nota:* além do JRE autônomo, há um JRE aninhado na pasta de instalação do seu JDK. Se estiver utilizando um IDE que depende do JDK (por exemplo, NetBeans), talvez você também precise inserir o arquivo `swing.properties` na pasta `lib` aninhada na pasta `jre`.]

Se você preferir selecionar o Nimbus individualmente por aplicativo, coloque o seguinte argumento de linha de comando depois do comando `java` e antes do nome do aplicativo quando você executar o aplicativo:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

14.3 Entrada/saída baseada em GUI simples com JOptionPane

Os aplicativos nos capítulos 2 a 10 exibem o texto na janela de comando e obtêm a entrada a partir da janela de comando. A maioria dos aplicativos que você utiliza diariamente utiliza janelas ou **caixas de diálogo** (também chamadas de **diálogos**) para interagir com o usuário. Por exemplo, um programa de email permite digitar e ler mensagens em uma janela fornecida pelo programa. Em geral, as caixas de diálogo são janelas em que programas exibem mensagens importantes para o usuário ou obtêm informações do usuário. A classe **JOptionPane** do Java (pacote `javax.swing`) fornece caixas de diálogo pré-construídas tanto para entrada como para saída. Esses diálogos são exibidos invocando métodos `JOptionPane static`. A Figura 14.2 apresenta um aplicativo de adição simples que utiliza dois **diálogos de entrada** para obter inteiros do usuário e um **diálogo de mensagem** para exibir a soma dos inteiros que o usuário insere.

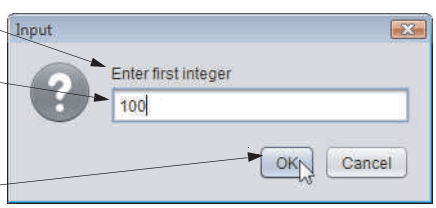
```
1  // Figura 14.2: Addition.java
2  // Programa de adição que utiliza JOptionPane para entrada e saída.
3  import javax.swing.JOptionPane; // programa utiliza JOptionPane
4
5  public class Addition
6  {
7     public static void main( String[] args )
8     {
9        // obtém a entrada de usuário a partir dos diálogos de entrada JOptionPane
10       String firstNumber =
11          JOptionPane.showInputDialog( "Enter first integer" );
12       String secondNumber =
13          JOptionPane.showInputDialog( "Enter second integer" );
14
15       // converte String em valores int para utilização em um cálculo
16       int number1 = Integer.parseInt( firstNumber );
17       int number2 = Integer.parseInt( secondNumber );
18
19       int sum = number1 + number2; // soma os números
20
21       // exibe o resultado em um diálogo de mensagem JOptionPane
22       JOptionPane.showMessageDialog( null, "The sum is " + sum,
23          "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24    } // fim do método main
25 } // fim da classe Addition
```

(a) Diálogo de entrada exibido pelas linhas 10–11

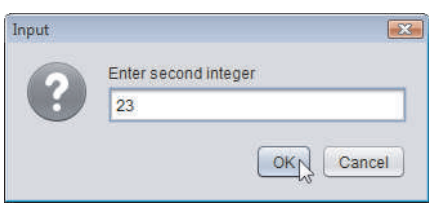
Prompt para o usuário

Campos de texto em que o usuário digita um valor

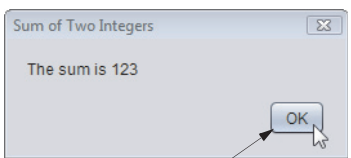
Quando o usuário clica em **OK**, `showInputDialog` retorna para o programa o **100** digitado pelo usuário como `String`. O programa deve converter a `String` em um `int`



(b) Diálogo de entrada exibido pelas linhas 12–13



(c) Diálogo de entrada exibido pelas linhas 22–23



Quando o usuário clica em **OK**, a caixa de diálogo da mensagem se fecha (desaparece da tela).

Figura 14.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída.

Diálogos de entrada

A linha 3 importa a classe `JOptionPane`. As linhas 10–11 declaram a variável `String` local `firstNumber` e atribuem a ela o resultado da chamada ao método `JOptionPane` `static` **`showInputDialog`**. Esse método exibe um diálogo de entrada (ver a primeira captura de tela na Figura 14.2), utilizando o argumento `String` do método (`"Enter first integer"`) como um prompt.

Observações sobre a aparência e comportamento 14.2

Em geral, o prompt em um diálogo de entrada emprega ***maiúsculas e minúsculas no estilo de frases*** *— um estilo que emprega a maiúscula inicial apenas na primeira palavra da frase a menos que a palavra seja um nome próprio (por exemplo, Jones).*

O usuário digita caracteres no campo de texto, depois clica em OK ou pressiona a tecla *Enter* para enviar a `String` para o programa. Clicar em OK também **fecha (oculta) o diálogo**. [*Nota:* se você digitar no campo de texto e não aparecer nada, ative o campo de texto clicando nele com o mouse.] Ao contrário de `Scanner`, que pode ser utilizado para inserir valores de vários tipos do usuário no teclado, um diálogo de entrada pode inserir somente `String`s. Isso é comum na maioria dos componentes GUI. O usuário pode digitar qualquer caractere no campo de texto do diálogo de entrada. Nosso programa assume que o usuário insere um inteiro válido. Se o usuário clica em Cancel, `showInputDialog` retorna `null`. Se o usuário digitar um valor não inteiro ou clicar no botão Cancel no diálogo de entrada, ocorrerá uma exceção e o programa não irá operar corretamente. O Capítulo 11, "Tratamento de exceções", discutiu como tratar esses erros. As linhas 12–13 exibem outro diálogo de entrada que pede ao usuário para inserir o segundo inteiro. Note que cada diálogo `JOptionPane` que você exibe é a chamada **caixa de diálogo modal** — enquanto o diálogo está na tela, o usuário não pode interagir com o restante do aplicativo.

Observações sobre a aparência e comportamento 14.3

Não use excessivamente caixas de diálogo modais uma vez que elas podem reduzir a usabilidade dos aplicativos. Utilize uma caixa de diálogo modal somente quando for necessário impedir usuários de interagir com o restante de um aplicativo até que eles descartem o diálogo.

Convertendo `Strings` em valores `int`

Para realizar o cálculo, convertemos as `String`s que o usuário inseriu em valores `int`. Lembre-se de que o método `static` `parseInt` da classe `Integer` converte seu argumento `String` em um valor `int`. As linhas 16–17 atribuem os valores convertidos às variáveis locais `number1` e `number2`. Em seguida, a linha 19 soma esses valores e atribui o resultado à variável local `sum`.

Diálogos de mensagem

As linhas 22–23 utilizam método `static` `JOptionPane` **`showMessageDialog`** para exibir um diálogo de mensagem (a última captura de tela da Figura 14.2) que contém a soma. O primeiro argumento ajuda o aplicativo Java a determinar onde posicionar a caixa de diálogo. Um diálogo é tipicamente exibido a partir de um aplicativo GUI em uma janela própria. O primeiro argumento refere-se a essa janela (conhecida como a janela pai) e faz com que o diálogo pareça centrado sobre a janela pai (como faremos na Seção 14.9). Se o primeiro argumento for `null`, a caixa de diálogo será exibida no centro da tela. O segundo argumento é a mensagem a exibir — nesse caso, o resultado da concatenação de `String` `"The sum is "` e do valor de `sum`. O terceiro argumento — `"Sum of Two Integers"` — é a `String` que deve aparecer na barra de título em cima do diálogo. O quarto argumento — **`JOptionPane.PLAIN_MESSAGE`** — é o tipo do diálogo de mensagem a exibir. O diálogo `PLAIN_MESSAGE` não exibe um ícone à esquerda da mensagem. A classe `JOptionPane` fornece várias versões sobrecarregadas dos métodos `showInputDialog` e `showMessageDialog`, bem como os métodos que exibem outros tipos de diálogo. Para informações completas sobre a classe `JOptionPane`, visite `java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html`.

Observações sobre a aparência e comportamento 14.4

Em geral, a barra de título de uma janela adota o uso de letras ***maiúsculas e minúsculas de título de livro*** *— um estilo que emprega a inicial maiúscula em cada palavra significativa no texto e não termina com pontuação (por exemplo, Uso de Letras Maiúsculas e Minúsculas no Título de um Livro).*

Constantes de diálogo de mensagem `JOptionPane`

As constantes que representam os tipos de diálogo de mensagem são mostradas na Figura 14.3. Todos os tipos de diálogo de mensagem exceto `PLAIN_MESSAGE` exibem um ícone à esquerda da mensagem. Esses ícones fornecem uma indicação visual da importância da mensagem para o usuário. Observe que um ícone `QUESTION_MESSAGE` é o ícone padrão de uma caixa de diálogo de entrada (ver Figura 14.2).

Tipo de diálogo de mensagem	Ícone	Descrição
`ERROR_MESSAGE`		Indica um erro ao usuário.
`INFORMATION_MESSAGE`		Indica uma mensagem informativa ao usuário.
`WARNING_MESSAGE`		Alerta o usuário de um potencial problema.
`QUESTION_MESSAGE`		Propõe uma questão ao usuário. Normalmente, esse diálogo exige uma resposta, como clicar em um botão Yes ou No.
`PLAIN_MESSAGE`	Nenhum ícone	Um diálogo que contém uma mensagem, mas nenhum ícone.

Figura 14.3 | Constantes `JOptionPane` `static` para diálogo de mensagem.

14.4 Visão geral de componentes Swing

Embora seja possível realizar entrada e saída utilizando os diálogos `JOptionPane` apresentados na Seção 14.3, a maioria dos aplicativos GUI exige interfaces com o usuário mais elaboradas e personalizadas. O restante deste capítulo discute muitos componentes GUI que permitem aos desenvolvedores de aplicações criar GUIs robustas. A Figura 14.4 lista vários **componentes GUI Swing** do pacote **`javax.swing`** que são utilizados para construir GUIs Java. A maioria dos componentes Swing são componentes Java puros — eles são completamente escritos, manipulados e exibidos em Java. Eles fazem parte das **Java Foundation Classes (JFC)** — bibliotecas do Java para desenvolvimento de GUI para múltiplas plataformas. Visite `java.sun.com/javase/technologies/desktop/` para obter mais informações sobre tecnologias desktop JFC e Java.

Componente	Descrição
`JLabel`	Exibe texto não editável ou ícones.
`JTextField`	Permite ao usuário inserir dados do teclado. Também pode ser utilizado para exibir texto editável ou não editável.
`JButton`	Desencadeia um evento quando o usuário clicar nele com o mouse.
`JCheckBox`	Especifica uma opção que pode ser ou não selecionada.
`JComboBox`	Fornece uma lista drop-down de itens a partir da qual o usuário pode fazer uma seleção clicando em um item ou possivelmente digitando na caixa.
`JList`	Fornece uma lista de itens a partir da qual o usuário pode fazer uma seleção clicando em qualquer item na lista. Múltiplos elementos podem ser selecionados.
`JPanel`	Fornece uma área em que os componentes podem ser colocados e organizados. Também pode ser utilizado como uma área de desenho para imagens gráficas.

Figura 14.4 | Alguns componentes GUI básicos.

Swing versus AWT

Há realmente dois conjuntos de componentes GUI no Java. Antes de o Swing ter sido introduzido no J2SE 1.2, as Java GUIs eram construídas com componentes do **Abstract Window Toolkit (AWT)** no pacote **`java.awt`**. Quando um aplicativo Java com uma AWT GUI executa em diferentes plataformas Java, os componentes GUI do aplicativo exibem diferentemente em cada plataforma. Considere um aplicativo que exibe um objeto do tipo `Button` (pacote `java.awt`). Em um computador que executa o sistema operacional do Microsoft Windows, `Button` terá a mesma aparência dos botões de outros aplicativos Windows. De maneira semelhante, em um computador que executa o sistema operacional Mac OS X da Apple, `Button` terá a mesma aparência e comportamento dos botões em outros aplicativos Macintosh. Às vezes, a maneira como um usuário pode interagir com um componente AWT particular difere entre plataformas.

Juntas, a aparência e a maneira como o usuário interage com o aplicativo são conhecidas como **a aparência e o comportamento** desse aplicativo. Os componentes Swing GUI permitem especificar uniformemente a aparência e comportamento para o aplicativo em todas as plataformas ou utilizar a aparência e comportamento personalizados de cada plataforma. Um aplicativo pode até mesmo alterar a aparência e comportamento durante a execução para permitir aos usuários escolher a aparência e comportamento preferidos.

Dica de portabilidade 14.1

Os componentes Swing são implementados no Java, desse modo eles são mais portáveis e flexíveis do que os componentes Java GUI originais de pacotes `java.awt`*, que foram baseados nos componentes GUI da plataforma subjacente. Por essa razão, os componentes Swing GUI geralmente são preferidos.*

Componentes GUI leves versus pesados

A maioria dos componentes Swing não está amarrada a componentes GUI reais da plataforma subjacente. Esses componentes GUI são conhecidos como **componentes leves**. Os componentes AWT (muitos dos quais são equivalentes dos componentes Swing) são vinculados à plataforma local e são chamados **componentes de peso pesado**, porque contam com o **sistema de janela** da plataforma local para determinar sua funcionalidade e sua aparência e comportamento.

Vários componentes Swing são componentes pesados. Como os componentes AWT, os componentes Swing GUI pesados requerem interação direta com o sistema de janela local, que pode restringir sua aparência e funcionalidade, tornando-os menos flexíveis do que os componentes leves.

Dica de portabilidade 14.2

A aparência e comportamento de uma GUI definida com componentes de GUI peso pesado do pacote `java.awt` *podem variar de uma plataforma para outra. Como os componentes pesados estão vinculados à GUI da plataforma local, a aparência e comportamento variam de plataforma para plataforma.*

Superclasses de componentes GUI leves do Swing

O diagrama de classe UML da Figura 14.5 mostra uma hierarquia de herança que contém classes a partir da quais os componentes Swing leves herdam seus atributos e comportamento comuns.

Observação de engenharia de software 14.1

Estude os atributos e comportamentos das classes na hierarquia de classe da Figura 14.5. Essas classes declaram os recursos comuns à maioria dos componentes Swing.

A classe **`Component`** (pacote `java.awt`) é uma subclasse de `Object` que declara muitos dos atributos e comportamentos comuns aos componentes GUI em pacotes `java.awt` e `javax.swing`. A maioria dos componentes GUI estende a classe `Component` direta ou indiretamente. Visite `java.sun.com/javase/6/docs/api/java/awt/Component.html` para obter uma lista completa dessas características comuns.

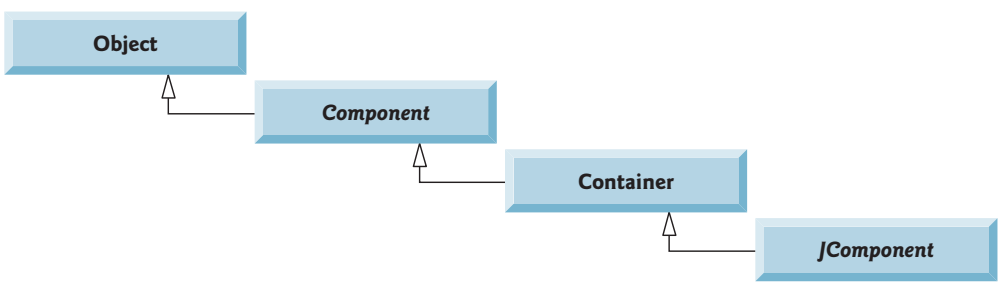


Figura 14.5 | Superclasses comuns de muitos dos componentes do Swing.

A classe **`Container`** (pacote `java.awt`) é uma subclasse de `Component`. Como você logo verá, `Components` são anexados a `Containers` (como janelas) de modo que os `Components` possam ser organizados e exibidos na tela. Qualquer objeto que *é um* `Container` pode ser usado para organizar outros `Components` em uma GUI. Como um `Container` *é um* `Component`, você pode colocar `Containers` em outros `Containers` para ajudar a organizar uma GUI. Visite `java.sun.com/javase/6/docs/api/java/awt/Container.html` para obter uma lista completa dos recursos do `Container` que são comuns aos componentes leves do Swing.

A classe **JComponent** (pacote `javax.swing`) é uma subclasse de `Container`. `JComponent` é a superclasse de todos os componentes leves Swing e declara seus atributos e comportamentos comuns. Como `JComponent` é uma subclasse de `Container`, todos os componentes Swing leves também são `Containers`. Alguns recursos de componentes leves comuns suportados por `JComponent` incluem:

1. **aparência e comportamento plugáveis** que podem ser utilizados para personalizar a aparência de componentes (por exemplo, para o uso em plataformas particulares). Você verá um exemplo disso na Seção 25.6.
2. teclas de atalho (chamadas **mnemônicas**) para acesso direto a componentes GUI pelo teclado. Você verá um exemplo disso na Seção 25.4.
3. capacidades de tratamento de evento comuns para casos em que vários componentes GUI iniciam as mesmas ações em um aplicativo.
4. breves descrições do propósito de um componente GUI (chamadas **dicas de ferramenta**). Essas são exibidas quando o cursor de mouse é posicionado sobre o componente por um breve instante. Você verá um exemplo disso na próxima seção.
5. suporte para a acessibilidade, como leitores de tela em braile para deficientes visuais.
6. suporte para **localização** de interface com o usuário — isto é, personalizar a interface com o usuário para exibir em diferentes linguagens e utilização de convenções culturais locais.

Visite `java.sun.com/javase/6/docs/api/javax/swing/JComponent.html` para obter mais detalhes sobre os recursos componentes leves comuns.

14.5 Exibição de texto e imagens em uma janela

Nosso próximo exemplo introduz um framework para construir aplicativos GUIs. Esse framework utiliza vários conceitos que você verá em muitos de nossos aplicativos GUI. Esse é nosso primeiro exemplo em que o aplicativo aparece na própria janela. A maioria das janelas que você criará que podem conter componentes Swing GUI são instâncias da classe `JFrame` ou uma subclasse de `JFrame`. `JFrame` é uma subclasse indireta da classe `java.awt.Window` que fornece os atributos e comportamentos básicos de uma janela — uma barra de títulos na parte superior e botões para minimizar, maximizar e fechar a janela. Visto que, em geral, a GUI de um aplicativo é específica ao aplicativo, a maioria dos nossos exemplos consistirá em duas classes — uma subclasse de `JFrame` que ajuda a demonstrar novos conceitos das GUIs e uma classe de aplicativo em que `main` cria e exibe a principal janela do aplicativo.

Rotulando componentes GUI

Uma GUI típica consiste em muitos componentes. Em uma grande GUI, pode ser difícil identificar o propósito de cada componente. Portanto, os projetistas de GUIs costumam fornecer um texto que declara a finalidade de cada componente. Tal texto é conhecido como **rótulo** e é criado com a classe **JLabel** — uma subclasse de `JComponent`. Um `JLabel` exibe texto somente de leitura, uma imagem ou tanto texto como imagem. Os aplicativos raramente alteram o conteúdo de um rótulo depois de criá-lo.

Observações sobre a aparência e comportamento 14.5

Normalmente, o texto em um `JLabel` emprega maiúsculas e minúsculas no estilo de frases.

O aplicativo das figuras 14.6 e 14.7 demonstra vários recursos `JLabel` e apresenta o framework que utilizamos na maioria de nossos exemplos de GUIs. Não destacamos o código nesse exemplo, visto que a maior parte dele é nova. [*Nota:* há muito mais recursos para cada componente GUI que podemos abranger em nossos exemplos. Para aprender os detalhes completos de cada componente GUI, visite sua página na documentação on-line. Para a classe `JLabel`, visite `java.sun.com/javase/6/docs/api/javax/swing/JLabel.html`.]

```
1  // Figura 14.6: LabelFrame.java
2  // Demonstrando a classe JLabel.
3  import java.awt.FlowLayout; // especifica como os componentes são organizados
4  import javax.swing.JFrame; // fornece recursos básicos de janela
5  import javax.swing.JLabel; // exibe texto e imagens
6  import javax.swing.SwingConstants; // constantes comuns utilizadas com Swing
7  import javax.swing.Icon; // interface utilizada para manipular imagens
8  import javax.swing.ImageIcon; // carrega imagens
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel label1; // JLabel apenas com texto
13    private JLabel label2; // JLabel construído com texto e ícone
14    private JLabel label3; // JLabel com texto e ícone adicionados
15
16    // construtor LabelFrame adiciona JLabels a JFrame
17    public LabelFrame()
18    {
19       super( "Testing JLabel" );
```

```
20          setLayout( new FlowLayout() ); // configura o layout de frame
21
22          // Construtor JLabel com um argumento de string
23          label1 = new JLabel( "Label with text" );
24          label1.setToolTipText( "This is label1" );
25          add( label1 ); // adiciona o label1 ao JFrame
26
27          // construtor JLabel com string, Icon e argumentos de alinhamento
28          Icon bug = new ImageIcon( getClass().getResource(  "bug1.png" ) );
29          label2 = new JLabel( "Label with text and icon", bug,
30             SwingConstants.LEFT );
31          label2.setToolTipText( "This is label2" );
32          add( label2 ); // adiciona label2 ao JFrame
33
34          label3 = new JLabel(); // Construtor JLabel sem argumentos
35          label3.setText( "Label with icon and text at bottom" );
36          label3.setIcon( bug ); // adiciona o ícone ao JLabel
37          label3.setHorizontalTextPosition( SwingConstants.CENTER );
38          label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39          label3.setToolTipText( "This is label3" );
40          add( label3 ); // adiciona label3 ao JFrame
41       } // fim do construtor LabelFrame
42    } // fim da classe LabelFrame
```

Figura 14.6 | `JLabel`s com texto e ícones.

```
1  // Figura 14.7: LabelTest.java
2  // Testando LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest
6  {
7     public static void main( String[] args )
8     {
9        LabelFrame labelFrame = new LabelFrame(); // cria LabelFrame
10       labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       labelFrame.setSize( 260, 180 ); // configura o tamanho do frame
12       labelFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe LabelTest
```

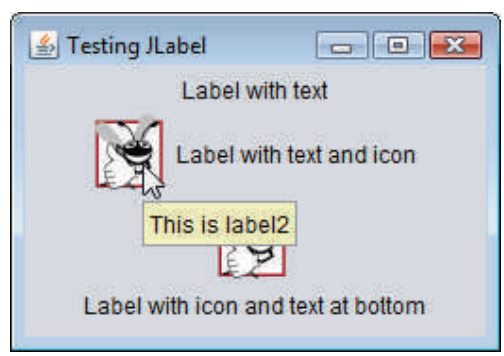


Figura 14.7 | Classe de teste para `LabelFrame`.

A classe `LabelFrame` (Figura 14.6) é uma subclasse de `JFrame`. Utilizaremos uma instância de classe `LabelFrame` para exibir uma janela que contém três `JLabel`s. As linhas 3–8 importam as classes utilizadas na classe `LabelFrame`. A classe estende `JFrame` para herdar os recursos de uma janela. As linhas 12–14 declaram as três variáveis de instância `JLabel`, das quais cada uma é instanciada no construtor `LabelFrame` (linhas 17–41). Em geral, o construtor `JFrame` da subclasse constrói a GUI que é exibida na janela quando o aplicativo executa. A linha 19 invoca o construtor da superclasse `JFrame` com o argumento `"Testing JLabel"`. O construtor de `JFrame` utiliza essa `String` como o texto na barra de título da janela.

Especificando o layout

Ao construir uma GUI, você deve anexar cada componente GUI a um contêiner, como uma janela criada com um `JFrame`. Além disso, em geral você deve decidir onde posicionar cada componente GUI. Isso é conhecido como especificar o layout dos componentes GUI. Como você aprenderá no final deste capítulo e no Capítulo 25, o Java fornece vários **gerenciadores de layout** que podem ajudá-lo a posicionar componentes.

Muitos ambientes de desenvolvimento integrados fornecem ferramentas de desenho da GUI em que você pode especificar o tamanho e local exatos de um componente em uma maneira visual utilizando o mouse; então o IDE gerará o código GUI para você. Embora esses IDEs possam simplificar significativamente a criação GUI, todos são diferentes em capacidade.

Para assegurar que os exemplos deste livro podem ser utilizados com qualquer IDE, não usamos um IDE para criar o código GUI. Por essa razão, utilizamos gerenciadores de layout do Java nos nossos exemplos de GUI. Um desses gerenciadores de layout é o **`FlowLayout`**, no qual os componentes GUI são colocados em um contêiner da esquerda para a direita na ordem em que o programa os anexa ao contêiner. Quando não houver mais espaço para ajustar componentes da esquerda para a direita, os componentes continuam a aparecer da esquerda para a direita na próxima linha. Se o contêiner for redimensionado, um `FlowLayout` recorre (isto é, reorganiza) os componentes para acomodar a nova largura do contêiner, possivelmente com menos ou mais linhas de componentes GUI. Cada contêiner tem um layout padrão, que estamos mudando de `LabelFrame` para um `FlowLayout` (linha 20). O método **`setLayout`** é herdado na classe `LabelFrame` indiretamente da classe `Container`. O argumento para o método deve ser um objeto de uma classe que implementa a interface `LayoutManager` (por exemplo, `FlowLayout`). A linha 20 cria um novo objeto `FlowLayout` e passa a sua referência como o argumento para `setLayout`.

Criando e anexando `label1`

Agora que especificamos o layout da janela, podemos começar a criar e anexar componentes GUI à janela. A linha 23 cria um objeto `JLabel` e passa `"Label with text"` para o construtor. `JLabel` exibe esse texto na tela como parte da GUI do aplicativo. A linha 24 utiliza o método **`setToolTipText`** (herdado por `JLabel` de `JComponent`) para especificar a dica de ferramenta que é exibida quando o usuário posiciona o cursor de mouse sobre o `JLabel` na GUI. Você pode ver uma dica de ferramenta de exemplo na segunda captura de tela da Figura 14.7. Ao executar esse aplicativo, tente posicionar o mouse por cima de cada `JLabel` para ver a dica de ferramenta. A linha 25 anexa `label1` ao `LabelFrame` passando `label1` para o método **`add`**, que é herdado indiretamente da classe `Container`.

Erro comum de programação 14.1

Se você não adicionar explicitamente um componente GUI a um contêiner, o componente GUI não será exibido quando o contêiner aparecer na tela.

Observações sobre a aparência e comportamento 14.6

Utilize as dicas de ferramenta para adicionar texto descritivo aos componentes GUI. Esse texto ajuda o usuário a determinar o propósito do componente GUI na interface com o usuário.

Criando e anexando `label2`

Os ícones são uma maneira popular para aprimorar a aparência e comportamento de um aplicativo e também são comumente utilizados para indicar funcionalidade. Por exemplo, a maioria dos players de VCR e DVD utiliza o mesmo ícone para reproduzir uma fita ou DVD. Vários componentes Swing podem exibir imagens. Normalmente, um ícone é especificado com um argumento **`Icon`** para um construtor ou para o método **`setIcon`** do componente. Um `Icon` é um objeto de qualquer classe que implementa a interface `Icon` (pacote `javax.swing`). Um tipo de classe assim é **`ImageIcon`** (pacote `javax.swing`), que suporta vários formatos de imagem, inclusive **Graphics Interchange Format (GIF)**, **Portable Network Graphics (PNG)** e **Joint Photographic Experts Group (JPEG)**. Os nomes de arquivo para cada um desses tipos terminam em `.gif`, `.png` ou `.jpg` (ou `.jpeg`), respectivamente. Discutimos imagens em mais detalhes no Capítulo 24, "Multimídia: applets e aplicativos".

A linha 28 declara um `ImageIcon`. O arquivo `bug1.png` contém a imagem para carregar e armazenar no objeto `ImageIcon`. Essa imagem está no diretório desse exemplo. O objeto `ImageIcon` é atribuído à referência `Icon` `bug`. Lembre-se, a classe `ImageIcon` implementa a interface `Icon`; um `ImageIcon` *é um* `Icon`.

Na linha 28, a expressão `getClass().getResource("bug1.png")` invoca o método **`getClass`** (herdado indiretamente da classe `Object`) para recuperar uma referência ao objeto `Class` que representa a declaração da classe `LabelFrame`. Essa referência é então utilizada para invocar o método `Class` **`getResource`**, que retorna a localização da imagem como um URL. O construtor `ImageIcon` utiliza o URL para localizar a imagem e, em seguida, carrega essa imagem na memória. Como discutimos no Capítulo 1, a JVM carrega as declarações de classe na memória, utilizando um carregador de classe. O carregador de classe sabe onde está cada classe que ele carrega no disco. O método `getResource` utiliza o carregador de classe do objeto `Class` para determinar a localização de um recurso, como um arquivo de imagem. Nesse exemplo, o arquivo de imagem é armazenado na mesma localização que o arquivo `LabelFrame.class`. As técnicas descritas aqui permitem a um aplicativo carregar arquivos de imagem a partir de localizações que são relativas ao arquivo `.class` do `LabelFrame` no disco.

Um `JLabel` pode exibir um `Icon`. As linhas 29–30 utilizam outro construtor `JLabel` para criar um `JLabel` que exibe o texto `"Label with text and icon"` e o `Icon bug` criado na linha 28. O último argumento do construtor indica que o conteúdo do rótulo está justificado à esquerda ou alinhado à esquerda (isto é, o ícone e texto estão à esquerda da área do rótulo na tela). A interface **`SwingConstants`** (pacote `javax.swing`) declara um conjunto de constantes de inteiro comuns (como `SwingConstants.LEFT`) que são usadas com muitos componentes Swing. Por padrão, o texto aparece à direita da imagem quando um rótulo contém tanto texto como imagem. Note que os alinhamentos horizontal e vertical de um `JLabel` podem ser configurados com os métodos **`setHorizontalAlignment`** e **`setVerticalAlignment`**, respectivamente. A linha 31 especifica o tool tip text para `label2` e a linha 32 adiciona `label2` ao `JFrame`.

Criando e anexando `label3`

A classe `JLabel` fornece métodos para alterar a aparência de um rótulo depois de ele ter sido instanciado. A linha 34 cria um `JLabel` e invoca seu construtor sem argumentos. Esse rótulo inicialmente não tem texto ou `Icon`. A linha 35 utiliza o método `JLabel` **`setText`** para configurar o texto exibido no rótulo. O método correspondente **`getText`** recupera o texto atual exibido em um rótulo. A linha 36 utiliza o método `JLabel` **`setIcon`** para especificar o `Icon` a ser exibido no rótulo. O método correspondente **`getIcon`** recupera o `Icon` atual exibido em um rótulo. As linhas 37–38 utilizam os métodos `JLabel` **`setHorizontalTextPosition`** e **`setVerticalTextPosition`** para especificar a posição de texto no rótulo. Nesse caso, o texto será centralizado horizontalmente e aparecerá na parte inferior do rótulo. Portanto, o `Icon` aparecerá acima do texto. As constantes de posição horizontal em `SwingConstants` são `LEFT`, `CENTER` e `RIGHT` (Figura 14.8). As constantes de posição vertical em `SwingConstants` são `TOP`, `CENTER` e `BOTTOM` (Figura 14.8). A linha 39 configura o tool tipo text de `label3`. A linha 40 adiciona `label3` a `JFrame`.

Constante	Descrição
Constantes de posição horizontal	
`SwingConstants.LEFT`	Coloca o texto à esquerda.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.RIGHT`	Coloca o texto à direita.
Constantes de posição vertical	
`SwingConstants.TOP`	Coloca o texto na parte superior.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.BOTTOM`	Coloca o texto na parte inferior.

Figura 14.8 | Posicionando constantes.

Criando e exibindo uma janela `LabelFrame`

A classe `LabelTest` (Figura 14.7) cria um objeto de classe `LabelFrame` (linha 9) e, em seguida, especifica a operação de fechamento padrão da janela. Por padrão, fechar uma janela simplesmente a oculta. Entretanto, quando o usuário fechar a janela `LabelFrame`, queremos que o aplicativo termine. A linha 10 invoca o método **`setDefaultCloseOperation`** de `LabelFrame` (herdado de classe `JFrame`) com a constante **`JFrame.EXIT_ON_CLOSE`** como o argumento para indicar que o programa deve terminar quando a janela for fechada pelo usuário. Essa linha é importante. Sem essa linha o aplicativo não terminará quando o usuário fechar a janela. Em seguida, a linha 11 invoca o método **`setSize`** de `LabelFrame` para especificar a largura e altura em pixels. Por fim, a linha 12 invoca o método **`setVisible`** de `LabelFrame` com o argumento `true` para exibir a janela na tela. Tente redimensionar a janela para ver como o `FlowLayout` altera as posições `JLabel` à medida que a largura de janela muda.

14.6 Campos de texto e uma introdução ao tratamento de evento com classes aninhadas

Normalmente, um usuário interage com uma GUI do aplicativo para indicar as tarefas que o aplicativo deve realizar. Por exemplo, ao escrever um email em um aplicativo de email, clicar no botão Send diz ao aplicativo para enviar o email para os endereços de email especificados. As GUIs são **baseadas em evento**. Quando o usuário interagir com um componente GUI, a interação — conhecida como **evento** — guia o programa para realizar uma tarefa. Algumas interações de usuário comuns que fazem com que um aplicativo realize uma tarefa incluem clicar em um botão, digitar em um campo de texto, selecionar o item de um menu, fechar uma janela e mover o mouse. O código que realiza uma tarefa em resposta a um evento é chamado de **handler de evento** e o processo total de responder a eventos é conhecido como **tratamento de evento**.

Consideremos dois outros componentes GUI que podem gerar eventos — **`JTextField`**s e **`JPasswordField`**s (pacote `javax.swing`). A classe `JTextField` estende a classe **`JTextComponent`** (pacote `javax.swing.text`), que fornece muitos recursos comuns aos componentes baseados em texto do Swing. A classe `JPasswordField` estende `JTextField` e adiciona métodos que são específicos ao processamento de senhas. Cada um desses componentes é uma área de uma única linha em que o usuário pode inserir texto pelo teclado. Os aplicativos também podem exibir texto em um `JTextField` (ver a saída da Figura 14.10). Um `JPasswordField` mostra que os caracteres estão sendo digitados à medida que o usuário os insere, mas oculta os caracteres reais com um **caractere eco**, assumindo que eles representam uma senha que só deve ser conhecida pelo usuário.

Quando o usuário digitar dados em um `JTextField` ou `JPasswordField`, e depois pressionar *Enter*, um evento ocorre. Nosso próximo exemplo demonstra como um programa pode realizar uma tarefa em resposta àquele evento. Todas as técnicas mostradas aqui são aplicáveis a componentes GUI que geram eventos.

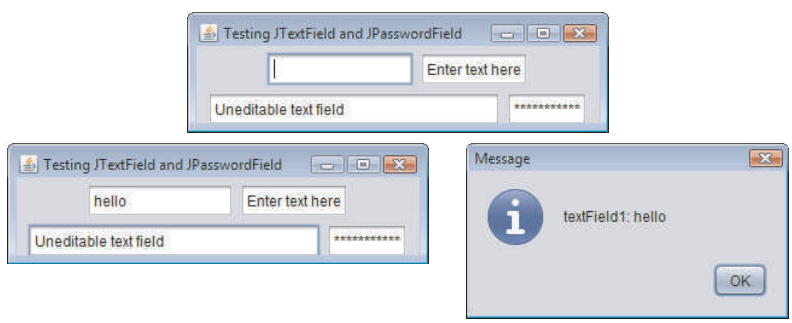
A aplicação das figuras 14.9 e 14.10 utiliza as classes `JTextField` e `JPasswordField` para criar e manipular quatro campos de texto. Quando o usuário digitar em um dos campos de texto, e depois pressionar *Enter*, o aplicativo exibirá uma caixa de diálogo de mensagem que contém o texto que ele digitou. Você pode digitar somente no campo de texto que estiver "em **foco**". Um componente recebe o foco quando o usuário clica no componente. Isso é importante, pois o campo de texto com o foco é o campo que gera um evento quando o usuário pressiona *Enter*. Neste exemplo, quando o usuário pressionar *Enter* no `JPasswordField`, a senha é revelada. Começamos discutindo a configuração da GUI, e depois discutimos o código de tratamento de evento.

```
1  // Figura 14.9: TextFieldFrame java
2  // Demonstrando a classe JTextField.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13    private JTextField textField1; // campo de texto com tamanho configurado
14    private JTextField textField2; // campo de texto construído com texto
15    private JTextField textField3; // campo de texto com texto e tamanho
16    private JPasswordField passwordField; // campo de senha com texto
17
18    // construtor TextFieldFrame adiciona JTextFields a JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // configura o layout de frame
23
24       // constrói textfield com 10 colunas
25       textField1 = new JTextField( 10 );
26       add( textField1 ); // adiciona textField1 ao JFrame
27
28       // constrói campo de texto com texto padrão
29       textField2 = new JTextField( "Enter text here" );
30       add( textField2 ); // adiciona textField2 ao JFrame
31
32       // constrói textfield com o texto padrão e 21 colunas
33       textField3 = new JTextField( "Uneditable text field", 21 );
34       textField3.setEditable( false ); // desativa a edição
35       add( textField3 ); // adiciona textField3 ao JFrame
36
37       // constrói passwordfield com o texto padrão
38       passwordField = new JPasswordField( "Hidden text" );
39       add( passwordField ); // adiciona passwordField ao JFrame
40
41       // handlers de evento registradores
42       TextFieldHandler handler = new TextFieldHandler();
43       textField1.addActionListener( handler );
44       textField2.addActionListener( handler );
45       textField3.addActionListener( handler );
46       passwordField.addActionListener( handler );
```

```
47      } // fim do construtor TextFieldFrame
48
49      // classe interna private para tratamento de evento
50      private class TextFieldHandler implements ActionListener
51      {
52         // processa eventos de campo de texto
53         public void actionPerformed( ActionEvent event )
54         {
55            String string = ""; // declara string a ser exibida
56
57            // usuário pressionou Enter no JTextField textField1
58            if (event.getSource() == textField1 )
59               string = String.format( "textField1: %s",
60                  event.getActionCommand() );
61
62            // usuário pressionou Enter no JTextField textField2
63            else if (event.getSource() == textField2 )
64               string = String.format( "textField2: %s",
65                  event.getActionCommand() );
66
67            // usuário pressionou Enter no JTextField textField3
68            else if (event.getSource() == textField3 )
69               string = String.format( "textField3: %s",
70                  event.getActionCommand() );
71
72            // usuário pressionou Enter no JTextField passwordField
73            else if (event.getSource() == passwordField )
74               string = String.format( "passwordField: %s",
75                  event.getActionCommand() );
76
77            // exibe o conteúdo de JTextField
78            JOptionPane.showMessageDialog( null, string );
79         } // fim do método actionPerformed
80      } // fim da classe TextFieldHandler interna private
81   } // fim da classe TextFieldFrame
```

Figura 14.9 | JTextFields e JPasswordFields.

```
1   // Figura 14.10: TextFieldTest.java
2   // Testando TextFieldFrame.
3   import javax.swing.JFrame;
4
5   public class TextFieldTest
6   {
7      public static void main( String[] args )
8      {
9         TextFieldFrame textFieldFrame = new TextFieldFrame();
10        textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        textFieldFrame.setSize( 350, 100 ); // configura o tamanho do frame
12        textFieldFrame.setVisible( true ); // exibe o frame
13     } // fim de main
14  } // fim da classe TextFieldTest
```



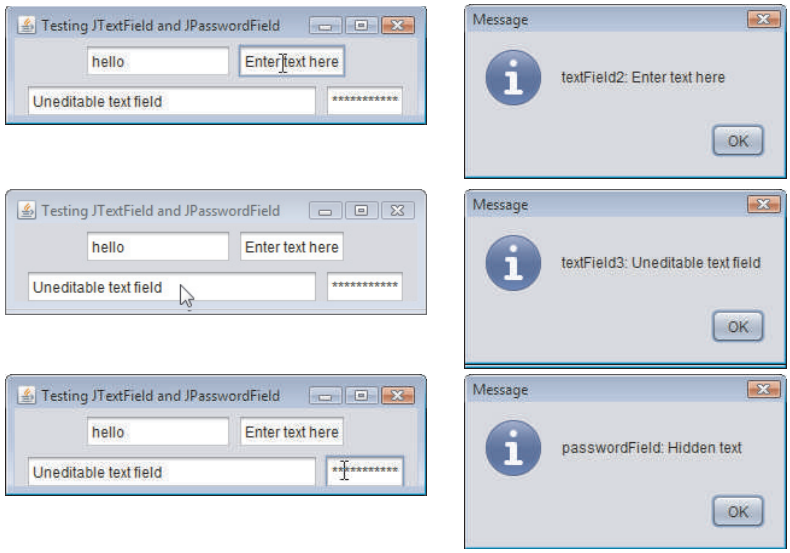


Figura 14.10 | Teste a classe `TextFieldFrame`.

As linhas 3–9 importam as classes e interfaces utilizadas nesse exemplo. A classe `TextFieldFrame` estende `JFrame` e declara três variáveis `JTextField` e uma variável `JPasswordField` (linhas 13–16). Cada um dos campos de texto correspondentes é instanciado e anexado ao `TextFieldFrame` no construtor (linhas 19–47).

Criando a GUI

A linha 22 configura o layout do `TextFieldFrame` como `FlowLayout`. A linha 25 cria `textField1` com `10` colunas de texto. A largura em pixels de uma coluna de texto é determinada pela largura média de um caractere na fonte atual do campo de texto. Quando o texto é exibido em um campo de texto e o texto for mais largo que o próprio campo de texto, uma parte do texto à direita não é visível. Se você estiver digitando em um campo de texto e o cursor alcançar o canto direito, o texto no canto esquerdo é incluído no lado esquerdo do campo e não é mais visível. Os usuários podem utilizar as teclas de seta esquerda e direita para mover-se por todo o texto, mesmo que o texto inteiro não seja visível de uma vez. A linha 26 adiciona `label3` a `JFrame`.

A linha 29 cria `textField2` com o texto inicial `"Enter text here"` para exibir no campo de texto. A largura do campo é determinada pela largura do texto padrão especificado no construtor. A linha 30 adiciona `label3` a `JFrame`.

A linha 33 cria `textField3` e chama o construtor `JTextField` com dois argumentos — o texto padrão `"Uneditable text field"` a ser exibido e o número de colunas (`21`). A largura do campo de texto é determinada pelo número de colunas especificadas. A linha 34 utiliza o método **`setEditable`** (herdado por `JTextField` da classe `JTextComponent`) para tornar o campo de texto não editável — isto é, o usuário não pode modificar o texto no campo. A linha 35 adiciona `label3` a `JFrame`.

A linha 38 cria `passwordField` com o texto `"Hidden text"` a ser exibido no campo de texto. A largura do campo é determinada pela largura do texto padrão. Ao executar o aplicativo, note que o texto é exibido como uma string de asteriscos. A linha 39 adiciona `label3` a `JFrame`.

Passos necessários para configurar o tratamento de evento de um componente GUI

Esse exemplo deve exibir um diálogo de mensagem que contém o texto de um campo de texto quando o usuário pressionar *Enter* nesse campo de texto. Antes que um aplicativo possa responder a um evento de um componente GUI particular, você deve realizar vários passos de codificação:

1. crie uma classe que represente o handler de evento.
2. implemente uma interface apropriada, conhecida como **interface ouvinte de eventos**, na classe do *Passo 1*.
3. indique que um objeto da classe dos *Passos 1* e *2* deve ser notificado quando o evento ocorrer. Isso é conhecido como **registrar handler de evento**.

Utilizando uma classe aninhada para implementar um handler de evento

Todas as classes discutidas até agora foram chamadas de **classes de primeiro nível** — isto é, elas não foram declaradas dentro de outra classe. O Java permite declarar classes dentro de outras classes — essas são chamadas de **classes aninhadas**. As classes aninhadas podem ser `static` ou não-`static`. As classes não `static` aninhadas são chamadas **classes internas** e são frequentemente utilizadas para implementar handlers de evento.

Antes que um objeto de uma classe interna possa ser criado, deve ser primeiro um objeto da classe de primeiro nível que contém a classe interna. Isso é necessário porque um objeto de classe interna tem implicitamente uma referência a um objeto de sua classe de primeiro nível. Há também um relacionamento especial entre esses objetos — o objeto da classe interna tem permissão de acessar diretamente todas as variáveis e métodos da classe externa. Uma classe aninhada que é `static` não exige um objeto de sua classe de primeiro nível e não tem implicitamente uma referência a um objeto da classe de primeiro nível. Como você verá no Capítulo 15, "Imagens gráficas e Java 2D™", a API de imagens gráficas 2D do Java utiliza extensivamente as classes `static` aninhadas.

Observação de engenharia de software 14.2

Uma classe interna tem permissão de acessar diretamente todas as variáveis e métodos de sua classe superior.

O tratamento de evento nesse exemplo é realizado por um objeto da classe interna `private` `TextFieldHandler` (linhas 50–80). Essa classe é `private` porque será utilizada apenas para criar handlers de evento para os campos de texto na classe de primeiro nível `TextFieldFrame`. Como com outros membros de classe, as classes internas podem ser declaradas `public`, `protected` ou `private`. Visto que as rotinas de tratamento de evento tendem a ser específicas do aplicativo em que são definidas, muitas vezes elas são implementadas como classes internas `private` ou, como você aprenderá na Seção 14.11, como classes internas anônimas.

Os componentes GUI podem gerar muitos eventos em resposta a interações de usuário. Cada evento é representado por uma classe e pode ser processado apenas pelo tipo de handler de evento apropriado. Normalmente, os eventos suportados de um componente são descritos na documentação da Java API para a classe desse componente e suas superclasses. Quando o usuário pressiona *Enter* em um `JTextField` ou `JPasswordField`, ocorre um **`ActionEvent`** (pacote `java.awt.event`). Um evento assim é processado por um objeto que implementa a interface **`ActionListener`** (pacote `java.awt.event`). As informações discutidas aqui são disponíveis na documentação de Java API das classes `JTextField` e `ActionEvent`. Visto que `JPasswordField` é uma subclasse de `JTextField`, `JPasswordField` suporta os mesmos eventos.

A fim de se preparar para tratar os eventos nesse exemplo, a classe interna `TextFieldHandler` implementa a interface `ActionListener` e declara o único método nessa interface — `actionPerformed` (linhas 53–79). Esse método especifica as tarefas a serem realizadas quando ocorrer um `ActionEvent`. Então, a classe interna `TextFieldHandler` satisfaz os *Passos 1* e *2* listados anteriormente nesta seção. Discutiremos os detalhes do método `actionPerformed` em breve.

Registrando o handler de evento para cada campo de texto

No construtor `TextFieldFrame`, a linha 42 cria um objeto `TextFieldHandler` e o atribui à variável `handler`. O método `actionPerformed` desse objeto será chamado automaticamente quando o usuário pressionar *Enter* em qualquer um dos campos de texto da GUI. Entretanto, antes que isso possa ocorrer, o programa deve registrar esse objeto como o handler de evento de cada campo de texto. As linhas 43–46 são as instruções de registro de evento que especificam `handler` como o handler de evento para os três `JTextFields` e o `JPasswordField`. O aplicativo chama o método `JTextField` **`addActionListener`** para registrar o handler de evento para cada componente. Esse método recebe como seu argumento um objeto `ActionListener`, que pode ser um objeto de qualquer classe que implemente `ActionListener`. O objeto `handler` *é um* `ActionListener`, porque a classe `TextFieldHandler` implementa `ActionListener`. Depois que as linhas 43–46 são executadas, o objeto `handler` **ouve eventos**. Agora, quando o usuário pressiona *Enter* em qualquer desses quatro campos de texto, o método `actionPerformed` (linhas 53–79) na classe `TextFieldHandler` é chamado para tratar o evento. Se um handler de evento não é registrado para um campo de texto particular, o evento que ocorre quando o usuário pressiona *Enter* nesse campo de texto é **consumido** — isto é, é simplesmente ignorado pelo aplicativo.

Observação de engenharia de software 14.3

O ouvinte de evento para um evento deve implementar a interface apropriada de ouvinte de evento.

Erro comum de programação 14.2

Esquecer de registrar um objeto tratador de evento para um tipo de evento de um componente GUI particular faz com que o tipo seja ignorado.

Detalhes do método `actionPerformed` da classe `TextFieldHandler`

Nesse exemplo, utilizamos o método `actionPerformed` de um objeto de tratamento de evento (linhas 53–79) para tratar os eventos gerados por quatro campos de texto. Visto que gostaríamos de gerar saída do nome de variável de instância de cada campo de texto para propósitos de demonstração, devemos determinar qual campo de texto gerou o evento toda vez que `actionPerformed` for chamado. O componente GUI com o qual o usuário interage é a **origem de evento**. Nesse exemplo, a origem de evento é um dos campos de texto ou o campo de senha. Quando o usuário pressionar *Enter* enquanto um desses componentes GUI tiver o foco, o sistema cria um objeto `ActionEvent` único que contém informações sobre o evento que acabou de ocorrer, como a origem de evento e o texto no campo de texto. O sistema então passa esse objeto `ActionEvent` em uma chamada de método para o método `actionPerformed` do ouvinte de evento. Nesse exemplo,

exibimos algumas dessas informações em um diálogo de mensagem. A linha 55 declara a `String` que será exibida. A variável é inicializada com a **string vazia** — uma `String` que não contém nenhum caractere. O compilador requer isso em caso de nenhum dos desvios do `if` aninhado nas linhas 58–75 executar.

O método `ActionEvent` `getSource` (que é herdado em `ActionEvent` a partir da classe `EventObject` e chamado nas linhas 58, 63, 68 e 73) retorna uma referência à origem de evento. A condição na linha 58 pergunta "`textField1` é a origem de evento?" Essa condição compara as referências nos dois lados do operador == para determinar se elas referenciam o mesmo objeto. Se ambas referenciarem `textField1`, o programa sabe que o usuário pressionou *Enter* em `textField1`. Nesse caso, as linhas 59–60 criam uma `String` que contém a mensagem que a linha 78 exibe em um diálogo de mensagem. A linha 60 utiliza o método `ActionEvent` **`getActionCommand`** para obter o texto que o usuário digitou no campo de texto que gerou o evento.

Nesse exemplo, exibimos o texto da senha no `JPasswordField` quando o usuário pressiona *Enter* nesse campo. Às vezes é necessário processar programaticamente os caracteres em uma senha. O método **`getPassword`** da classe `JPasswordField` retorna os caracteres da senha como um array de tipo `char`.

Classe `TextFieldTest`

A classe `TextFieldTest` (Figura 14.10) contém o método `main` que executa esse aplicativo e exibe um objeto da classe `TextFieldFrame`. Ao executar o aplicativo, observe que até o `JTextField` não editável (`textField3`) pode gerar um `ActionEvent`. Para testar isso, clique no campo de texto para colocá-lo em foco, depois pressione *Enter*. Também observe que o texto real da senha é exibido quando você pressionar *Enter* no `JPasswordField`. Naturalmente, você em geral não exibiria a senha!

Esse aplicativo utilizou um único objeto de classe `TextFieldHandler` como o ouvinte, ou listener, de evento para quatro campos de texto. Iniciando na Seção 14.10, você verá que é possível declarar vários objetos ouvintes de evento do mesmo tipo e registrar cada objeto para o evento de um componente GUI separado. Essa técnica permite eliminar a lógica `if...else` utilizada no handler de evento desse exemplo fornecendo handlers de evento separados para os eventos de cada componente.

14.7 Tipos comuns de eventos GUI e interfaces ouvintes

Na Seção 14.6, você aprendeu que as informações sobre o evento que ocorre quando o usuário pressiona *Enter* em um campo de texto são armazenadas em um objeto `ActionEvent`. Muitos tipos diferentes de eventos podem ocorrer quando o usuário interage com uma GUI. As informações de evento são armazenadas em um objeto de uma classe que estende `AWTEvent` (do pacote `java.awt`). A Figura 14.11 ilustra uma hierarquia que contém muitas classes de evento do pacote **`java.awt.event`**. Alguns desses são discutidos neste capítulo e no Capítulo 25. Esses tipos de evento são utilizados tanto com componentes AWT como com Swing. Tipos adicionais de evento que são específicos dos componentes Swing GUI são declarados no pacote **`javax.swing.event`**.

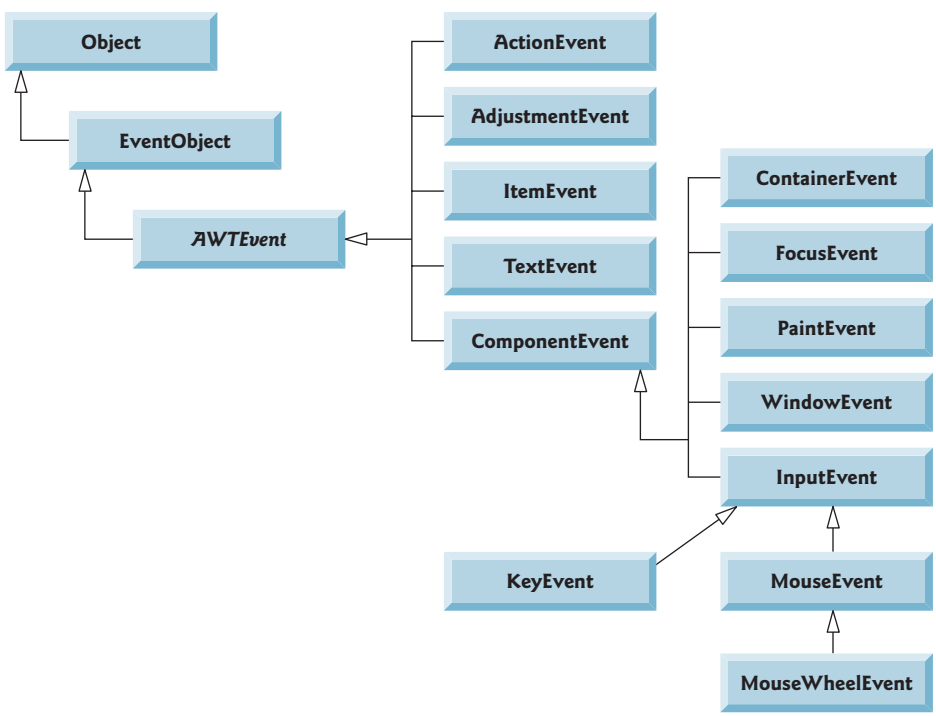


Figura 14.11 | Algumas classes de evento do pacote `java.awt.event`.

Vamos resumir as três partes para o mecanismo de tratamento de evento que você viu na Seção 14.6 — a origem do evento, o objeto do evento e o ouvinte de evento. A origem do evento é o componente GUI com o qual o usuário interage. O objeto do evento encapsula informações sobre o evento que ocorreu, como uma referência à origem do evento e quaisquer informações específicas do evento que podem ser exigidas pelo ouvinte de evento para tratar o evento. O ouvinte de evento é um objeto que é notificado pela origem de evento quando um evento ocorre; de fato, ele "ouve" um evento e um de seus métodos executa em resposta ao evento. Um método do ouvinte de evento recebe um objeto do evento quando o ouvinte de evento é notificado do evento. O ouvinte de evento então utiliza o objeto do evento para responder ao evento. O modelo de tratamento de evento descrito aqui é conhecido como **modelo de delegação de evento** — o processamento de um evento é delegado a um objeto particular (o ouvinte de evento) no aplicativo.

Para cada tipo de objeto de evento, há em geral uma interface listener de eventos correspondentes. Um ouvinte de evento para um evento GUI é um objeto de uma classe que implementa uma ou mais das interfaces ouvintes de evento dos pacotes `java.awt.event` e `javax.swing.event`. Muitos dos tipos de ouvinte de evento são comuns aos componentes Swing e AWT. Esses tipos são declarados no pacote `java.awt.event`, e alguns deles são mostrados na Figura 14.12. Os tipos de ouvinte de evento adicionais que são específicos dos componentes Swing são declarados no pacote `javax.swing.event`.

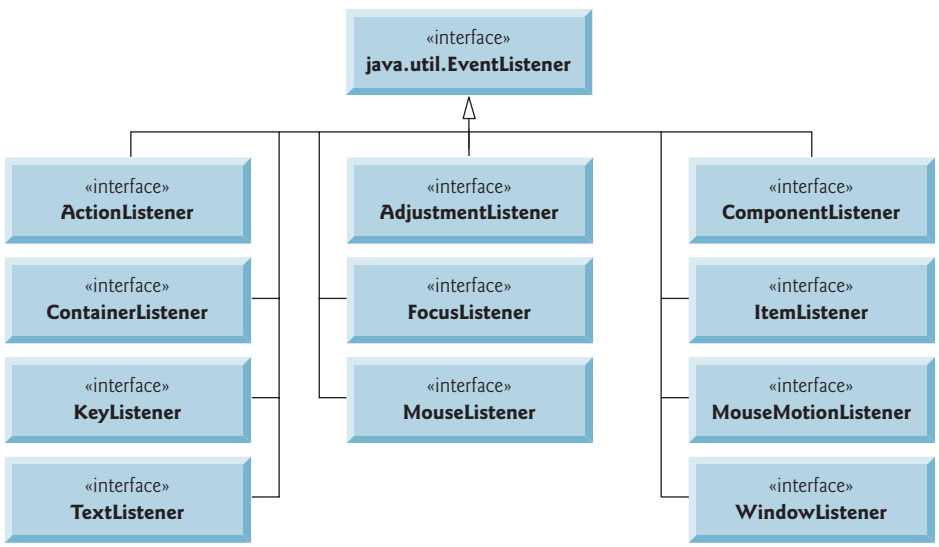


Figura 14.12 | Algumas interfaces listener de eventos comuns do pacote `java.awt.event`.

Cada interface listener de eventos especifica um ou mais métodos de tratamento de evento que devem ser declarados na classe que implementa a interface. A partir da Seção 10.7, lembre-se de que qualquer classe que implementa uma interface deve declarar todos os métodos `abstract` dessa interface; caso contrário, a classe é uma classe `abstract` e não pode ser utilizada para criar objetos.

Quando um evento ocorre, o componente GUI com o qual o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte. Por exemplo, quando o usuário pressiona a tecla *Enter* em um `JTextField`, o método `actionPerformed` do ouvinte registrado é chamado. Como o handler de evento é registrado? Como o componente GUI sabe chamar `actionPerformed` em vez de outro método de tratamento de evento? Respondemos a essas perguntas e diagramamos a interação na próxima seção.

14.8 Como o tratamento de evento funciona

Vamos ilustrar como o mecanismo de tratamento de evento funciona, utilizando `textField1` do exemplo da Figura 14.9. Temos duas perguntas abertas restantes da Seção 14.7:

1. como o handler de evento é registrado?
2. como o componente GUI sabe chamar `actionPerformed` em vez de algum outro método de tratamento de evento?

A primeira pergunta é respondida pelo registro de evento realizado nas linhas 43–46 da Figura 14.9. A Figura 14.13 diagrama a variável `JTextField textField1`, variável `TextFieldHandler handler` e os objetos aos quais elas se referem.

Registrando eventos

Cada `Jcomponent` tem uma variável de instância chamada `listenerList` que referencia um objeto de classe **`EventListenerList`** (pacote `javax.swing.event`). Cada objeto de uma subclasse `JComponent` mantém referências a seus ouvintes (listeners) registrados na `listenerList`. Por uma questão de simplicidade, diagramamos `listenerList` como um array abaixo do objeto `JTextField` na Figura 14.13.

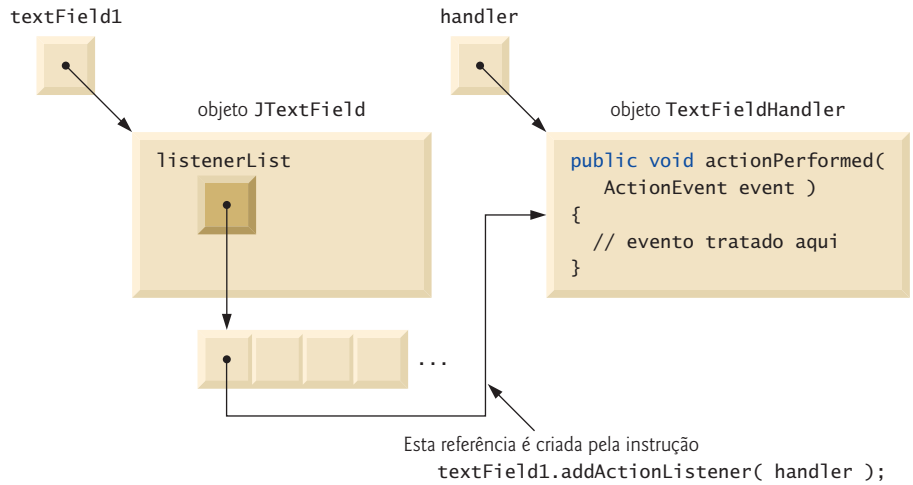


Figura 14.13 | Registro de evento para o `JTextField textField1`.

Quando a linha 43 da Figura 14.9

```
textField1.addActionListener( handler );
```

for executada, uma nova entrada que contém uma referência ao objeto `TextFieldHandler` é colocada em `listenerList` de `textField1`. Embora não mostrado no diagrama, essa nova entrada também inclui o tipo do listener (nesse caso, `ActionListener`). Utilizando esse mecanismo, cada componente Swing GUI leve mantém sua própria lista de ouvintes que foram registrados para tratar os eventos do componente.

Invocação de handler de evento

O tipo ouvinte de evento é importante ao responder à segunda pergunta: Como o componente GUI sabe chamar `actionPerformed` em vez de outro método? Cada componente GUI suporta vários tipos de evento, inclusive **eventos de mouse**, **eventos de teclado** e outros. Quando um evento ocorre, ele é despachado apenas para os ouvintes de evento do tipo apropriado. O despacho (dispatching) é simplesmente o processo pelo qual o componente GUI chama um método de tratamento de evento em cada um de seus ouvintes que são registrados para o tipo de evento que ocorreu.

Cada tipo de evento tem uma ou mais interfaces listener de eventos correspondentes. Por exemplo, `ActionEvent`s são tratados por `ActionListener`s, **`MouseEvent`**s são tratados por **`MouseListener`**s e **`MouseMotionListener`**s e **`KeyEvent`**s são tratados por **`KeyListener`**s. Quando ocorre um evento, o componente GUI recebe (do JVM) um único **ID de evento** para especificar o tipo de evento. O componente GUI utiliza o ID de evento para decidir o tipo de ouvinte para o evento que deve ser despachado e decidir qual método chamar em cada objeto listener. Para um `ActionEvent`, o evento é **despachado** para o método `actionPerformed` de cada `ActionListener` registrado (o único método na interface `ActionListener`). Para um `MouseEvent`, o evento é despachado para cada `MouseListener` ou `MouseMotionListener` registrado, dependendo do evento de mouse que ocorre. O ID de evento do `MouseEvent` determina quais dos vários métodos de tratamento de evento de mouse são chamados. Todas essas decisões são tratadas para você pelos componentes GUI. Tudo que você precisa fazer é registrar um handler de evento para o tipo particular de evento que seu aplicativo exige e o componente GUI assegurará que o método apropriado do handler de evento é chamado quando o evento ocorre. [*Nota:* discutimos outros tipos de evento e interfaces ouvintes de evento à medida que eles se tornarem necessários com cada novo componente que apresentarmos.]

14.9 JButton

Um **botão** é um componente em que o usuário clica para acionar uma ação específica. Um aplicativo Java pode usar vários tipos de botões, incluindo **botões de comando**, **caixas de seleção**, **botões de alternação** e **botões de opção**. A Figura 14.14 mostra a hierarquia de herança dos botões do Swing que abordaremos neste capítulo. Como você pode ver, todos os tipos de botão são subclasses de **`AbstractButton`** (pacote `javax.swing`), que declara os recursos comuns de botões Swing. Nesta seção, concentramos-nos nos botões que são em geral utilizados para iniciar um comando.

Observações sobre a aparência e comportamento 14.7

Em geral, os botões utilizam letras maiúsculas e minúsculas no estilo de título de livro.

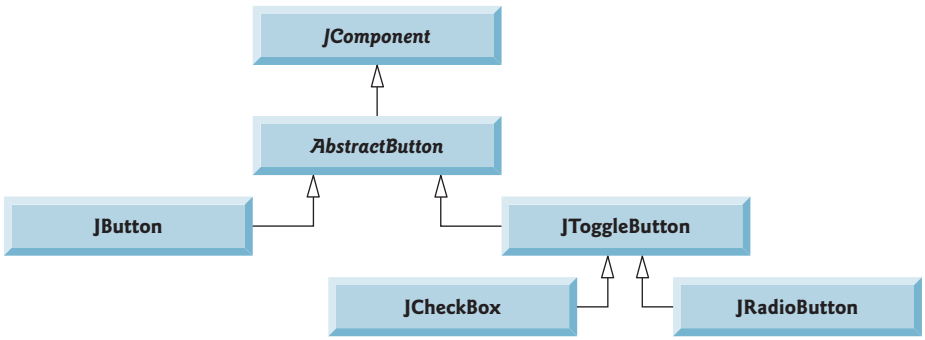


Figura 14.14 | Hierarquia do botão Swing.

Um botão de comando (ver a saída da Figura 14.16) gera um `ActionEvent` quando o usuário o clica. Os botões de comando são criados com a classe **`JButton`**. O texto na face de um `JButton` é chamado **rótulo de botão**. Uma GUI pode ter muitos `JButtons`, mas, em geral, cada rótulo de botão deve ser único na parte da GUI que é atualmente exibida.

Observações sobre a aparência e comportamento 14.8

Ter mais de um `JButton` com o mesmo rótulo torna os `JButtons` ambíguos para o usuário. Forneça um rótulo único para cada botão.

O aplicativo das figuras 14.15 e 14.16 cria dois `JButtons` e demonstra que `JButtons` suportam a exibição de `Icons`. O tratamento de evento para os botões é realizado por uma única instância da classe interna `ButtonHandler` (linhas 39–47).

```
1  // Figura 14.15: ButtonFrame.java
2  // Criando JButtons.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14    private JButton plainJButton; // botão apenas com texto
15    private JButton fancyJButton; // botão com ícones
16
17    // ButtonFrame adiciona JButtons ao JFrame
18    public ButtonFrame()
19    {
20       super( "Testing Buttons" );
21       setLayout( new FlowLayout() ); // configura o layout de frame
22
23       plainJButton = new JButton( "Plain Button" ); // botão com texto
24       add( plainJButton ); // adiciona plainJButton ao JFrame
25
26       Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27       Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28       fancyJButton = new JButton( "Fancy Button", bug1 ); // configura imagem
29       fancyJButton.setRolloverIcon( bug2 ); // configura imagem de rollover
30       add( fancyJButton ); // adiciona fancyJButton ao JFrame
31
32       // cria novo ButtonHandler para tratamento de evento de botão
33       ButtonHandler handler = new ButtonHandler();
34       fancyJButton.addActionListener( handler );
35       plainJButton.addActionListener( handler );
36    } // fim do construtor ButtonFrame
37
38    // classe interna para tratamento de evento de botão
```

```
39      private class ButtonHandler implements ActionListener
40      {
41         // trata evento de botão
42         public void actionPerformed( ActionEvent event )
43         {
44            JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
45               "You pressed: %s", event.getActionCommand() ) );
46         } // fim do método actionPerformed
47      } // fim da classe ButtonHandler private interna
48   } // fim da classe ButtonFrame
```

Figura 14.15 | Botões de comando e eventos de ação.

As linhas 14–15 declaram variáveis `JButton` `plainJButton` e `fancyJButton`. Os objetos correspondentes são instanciados no construtor. A linha 23 cria `plainJButton` com o rótulo de botão `"Plain Button"`. A linha 24 adiciona o `JButton` ao `JFrame`.

```
1   // Figura 14.16: ButtonTest.java
2   // Testando ButtonFrame.
3   import javax.swing.JFrame;
4
5   public class ButtonTest
6   {
7      public static void main( String[] args )
8      {
9         ButtonFrame buttonFrame = new ButtonFrame(); // cria ButtonFrame
10        buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        buttonFrame.setSize( 275, 110 ); // configura o tamanho do frame
12        buttonFrame.setVisible( true ); // exibe o frame
13     } // fim de main
14  } // fim da classe ButtonTest
```

Figura 14.16 | Teste a classe `ButtonFrame`.

Um `JButton` pode exibir um `Icon`. Para fornecer ao usuário um nível extra de interação visual com a GUI, um `JButton` também pode ter um **Icon rollover** — um `Icon` que é exibido quando o usuário posiciona o mouse sobre `JButton`. O ícone em `JButton` altera-se quando o mouse move-se para dentro e para fora da área de `Jbutton` na tela. As linhas 26–27 (Figura 14.15) criam dois objetos `ImageIcon` que representam o `Icon` padrão e `Icon` rollover para o `JButton` criado na linha 28. Ambas as instruções supõem que os arquivos de imagem sejam armazenados no mesmo diretório que o aplicativo. As imagens são comumente colocadas no mesmo diretório que o aplicativo ou em um subdiretório como `images`. Esses arquivos de imagem foram fornecidos para você com o exemplo.

A linha 28 cria `fancyButton` com o texto `"Fancy Button"` e o ícone `bug1`. Por padrão, o texto é exibido à direita do ícone. A linha 29 utiliza **`setRolloverIcon`** (herdado da classe `AbstractButton`) para especificar a imagem exibida em `JButton` quando o usuário posiciona o mouse sobre ele. A linha 30 adiciona o `JButton` ao `JFrame`.

Observações sobre a aparência e comportamento 14.9

Como a classe `AbstractButton` suporta exibição de texto e imagens em um botão, todas as subclasses de `AbstractButton` também suportam exibição de texto e imagens.

Observações sobre a aparência e comportamento 14.10

Utilizar ícones rollover para `JButton`s fornece aos usuários um feedback visual que indica que, quando eles clicam no mouse enquanto o cursor está posicionado sobre `JButton`, uma ação ocorrerá.

`JButton`s, como `JTextField`s, geram `ActionEvent`s que podem ser processados por qualquer objeto `ActionListener`. As linhas 33–35 criam um objeto de classe interna `private` `ButtonHandler` e utilizam `addActionListener` para registrá-lo como o handler de evento para cada `JButton`. A classe `ButtonHandler` (linhas 39–47) declara `actionPerformed` para exibir uma caixa de diálogo de mensagem que contém o rótulo do botão que o usuário pressionou. Para um evento `JButton`, o método `ActionEvent` `getActionCommand` retorna o rótulo no `JButton`.

Acessando a referência `this` em um objeto de uma classe de primeiro nível a partir de uma classe interna

Ao executar esse aplicativo e clicar em um de seus botões, note que o diálogo de mensagem que aparece é centralizado na janela do aplicativo. Isso ocorre porque a chamada para o método `JOptionPane` `showMessageDialog` (linhas 44–45 da Figura 14.15) utiliza `ButtonFrame.this` em vez de `null` como o primeiro argumento. Quando esse argumento não for `null`, ele representa o componente GUI pai do diálogo de mensagem (nesse caso, a janela de aplicativo é o componente pai) e permite ao diálogo estar centralizado sobre esse componente quando o diálogo for exibido. `ButtonFrame.this` representa a referência `this` do objeto de classe de primeiro nível `ButtonFrame`.

Observação de engenharia de software 14.4

Quando utilizada em uma classe interna, a palavra-chave `this` referencia o objeto de classe interna atual sendo manipulado. Um método de classe interna pode utilizar `this` do seu objeto de classe externa precedendo `this` com o nome de classe externa e um ponto, como em `ButtonFrame.this`.

14.10 Botões que mantêm o estado

Os componentes Swing GUI contêm três tipos de **botões de estado** — **`JToggleButton`**, **`JCheckBox`** e **`JRadioButton`** — isso tem valores ativado/desativado ou verdadeiro/falso. As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton` (Figura 14.14). Um `JRadioButton` é diferente de um `JCheckBox` no sentido de que normalmente vários `JRadioButton`s estão agrupados e são mutuamente exclusivos — apenas um no grupo pode ser selecionado de cada vez, assim como os botões em um rádio de carro de antigamente. Primeiro discutimos a classe `JCheckBox`.

14.10.1 JCheckBox

O aplicativo das figuras 14.17 e 14.18 usa dois `JcheckBoxes` para selecionar o estilo desejado de fonte do texto exibido em um `JTextField`. Quando selecionado, um aplica o estilo negrito e o outro, o estilo itálico. Se ambos forem selecionados, o estilo é negrito e itálico. Quando o aplicativo é inicialmente executado, nenhuma `JCheckBox` está marcada (isto é, ambas são `false`), então a fonte é simples. A classe `CheckBoxTest` (Figura 14.18) contém o método `main` que executa esse aplicativo.

```
1  // Figura 14.17: CheckBoxFrame.java
2  // Creating JCheckBox buttons.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13    private JTextField textField; // exibe o texto na alteração de fontes
```

```
14       private JCheckBox boldJCheckBox; // para selecionar estilo negrito/remover a seleção do estilo negrito
15       private JCheckBox italicJCheckBox; // para selecionar itálico/remover a seleção do estilo itálico
16
17       // construtor CheckBoxFrame adiciona JCheckBoxes ao JFrame
18       public CheckBoxFrame()
19       {
20          super( "JCheckBox Test" );
21          setLayout( new FlowLayout() ); // configura o layout de frame
22
23          // configura JTextField e sua fonte
24          textField = new JTextField( "Watch the font style change", 20 );
25          textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26          add( textField ); // adiciona textField ao JFrame
27
28          boldJCheckBox = new JCheckBox( "Bold" ); // cria caixa de seleção para negrito
29          italicJCheckBox = new JCheckBox( "Italic" ); // cria itálico
30          add( boldJCheckBox ); // adiciona caixa de seleção de estilo negrito ao JFrame
31          add( italicJCheckBox ); // adiciona caixa de seleção de itálico ao JFrame
32
33          // listeners registradores para JCheckBoxes
34          CheckBoxHandler handler = new CheckBoxHandler();
35          boldJCheckBox.addItemListener( handler );
36          italicJCheckBox.addItemListener( handler );
37       } // fim do construtor CheckBoxFrame
38
39       // classe interna private para tratamento de evento ItemListener
40       private class CheckBoxHandler implements ItemListener
41       {
42          // responde aos eventos de caixa de seleção
43          public void itemStateChanged( ItemEvent event )
44          {
45             Font font = null; // armazena a nova Font
46
47             // determina que CheckBoxes estão marcadas e cria Font
48             if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
49                font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
50             else if (boldJCheckBox.isSelected() )
51                font = new Font( "Serif", Font.BOLD, 14 );
52             else if (italicJCheckBox.isSelected() )
53                font = new Font( "Serif", Font.ITALIC, 14 );
54             else
55                font = new Font( "Serif", Font.PLAIN, 14 );
56
57             textField.setFont( font ); // configura a fonte do textField
58          } // fim do método itemStateChanged
59       } // fim da classe CheckBoxHandler interna private
60    } // fim da classe CheckBoxFrame
```

Figura 14.17 | Botões `JCheckBox` e eventos de item.

```
1    // Figura 14.18: CheckBoxTest.java
2    // Testando CheckBoxFrame.
3    import javax.swing.JFrame;
4
5    public class CheckBoxTest
6    {
7       public static void main( String[] args )
8       {
9          CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
10         checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11         checkBoxFrame.setSize( 275, 100 ); // configura o tamanho do frame
12         checkBoxFrame.setVisible( true ); // exibe o frame
13      } // fim de main
14   } // fim da classe CheckBoxTest
```

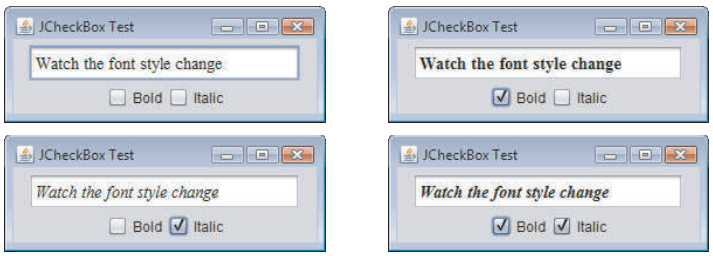


Figura 14.18 | Teste a classe `CheckBoxFrame`.

Depois de o `JTextField` ser criado e inicializado (Figura 14.17, linha 24), a linha 25 usa o método **`setFont`** (herdado por `JTextField` indiretamente da classe `Component`) para configurar a fonte do `JTextField` como um novo objeto de classe **`Font`** (pacote `java.awt`). A nova `Font` é inicializada com `"Serif"` (um nome de fonte para representar uma fonte genérica como Times e é suportada em todas as plataformas do Java), estilo `Font.PLAIN` e corpo `14`. Em seguida, as linhas 28–29 criam dois objetos `JCheckBox`. A `String` passada para o construtor `JCheckBox` é o **rótulo de caixa de seleção** que aparece à direita da `JCheckBox` por padrão.

Quando o usuário clicar em uma `JCheckBox`, um **`ItemEvent`** ocorre. Esse evento pode ser tratado por um objeto **`ItemListener`**, que deve implementar o método **`itemStateChanged`**. Nesse exemplo, o tratamento de evento é realizado por uma instância da classe interna `private` `CheckBoxHandler` (linhas 40–59). As linhas 34–36 criam uma instância de classe `CheckBoxHandler` e a registram com o método **`addItemListener`** como o ouvinte de ambos os objetos `JCheckBox`.

O método `CheckBoxHandler` `itemStateChanged` (linhas 43–58) é chamado quando o usuário clica no `boldJCheckBox` ou `italicJCheckBox`. Nesse exemplo, não precisamos saber qual dos dois `JCheckBoxes` foi clicado, somente se cada um está marcado. A linha 48 utiliza o método `JCheckBox` **`isSelected`** para determinar se ambos os `JCheckBoxes` estão selecionados. Se estiverem, a linha 49 cria uma fonte em itálico e negrito adicionando as constantes `Font` `Font.BOLD` e `Font.ITALIC` ao argumento de estilo da fonte do construtor `Font`. A linha 50 determina se `boldJCheckBox` está selecionado e, se estiver, a linha 51 cria uma fonte em negrito. A linha 52 determina se `italicJCheckBox` está selecionado e, se estiver, a linha 53 cria uma fonte em itálico. Se nenhuma das condições precedentes for verdadeira, a linha 55 cria uma fonte redonda que utiliza a constante `Font` `Font.PLAIN`. Por fim, a linha 57 configura a nova fonte do `textField`, que altera a fonte no `JTextField` na tela.

Relacionamento entre uma classe interna e sua classe de primeiro nível

Você pode ter notado que a classe `CheckBoxHandler` usou variáveis `boldJCheckBox` (Figura 14.17, linhas 48–50), `italicJCheckBox` (linhas 48–52) e `textField` (linha 57), mesmo que essas variáveis não estejam declaradas na classe interna. Lembre-se de que uma classe interna tem uma relação especial com sua classe de nível superior — ela tem permissão para acessar todas as variáveis e métodos da classe superior. O método `CheckBoxHandler` `itemStateChanged` (linhas 43–58) utiliza essa relação para determinar quais `JCheckBoxes` estão marcadas e estabelecer a fonte no `JTextField`. Observe que nenhum código na classe interna `CheckBoxHandler` exige uma referência explícita ao objeto de primeiro nível de classe.

14.10.2 JRadioButton

Os **botões de opção** (declarados com a classe `JRadioButton`) são semelhantes a caixas de seleção porque têm dois estados — selecionado e não selecionado. Entretanto, os botões de opção normalmente aparecem como um **grupo** em que apenas um botão pode ser selecionado por vez (ver a saída da Figura 14.20). Selecionar um botão de opção diferente força a remoção da seleção de todos os outros que estão selecionados. Os botões de opção são utilizados para representar **opções mutuamente exclusivas** (isto é, não é possível selecionar múltiplas opções no grupo ao mesmo tempo). O relacionamento lógico entre botões de opção é mantido por um objeto **`ButtonGroup`** (pacote `javax.swing`), que em si não é um componente GUI. Um objeto `ButtonGroup` organiza um grupo de botões e ele mesmo não é exibido em uma interface com o usuário. Em seu lugar, os objetos individuais `JRadioButton` do grupo são exibidos na GUI.

Erro comum de programação 14.3

Adicionar um objeto `ButtonGroup` (ou um objeto de qualquer outra classe que não deriva de `Component`) a um contêiner resulta em um erro de compilação.

O aplicativo das figuras 14.19 e 14.20 é semelhante ao das figuras 14.17 e 14.18. O usuário pode alterar o estilo da fonte do texto de um `JTextField`. O aplicativo utiliza botões de opção que permitem que apenas um único estilo de fonte no grupo seja selecionado de cada vez. A classe `RadioButtonTest` (Figura 14.20) contém o método `main` que executa esse aplicativo.

```
1  // Figura 14.19: RadioButtonFrame.java
2  // Criando botões de opção utilizando ButtonGroup e JRadioButton.
3  import java.awt.FlowLayout;
```

```
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14    private JTextField textField; // utilizado para exibir alterações de fonte
15    private Font plainFont; // fonte para texto simples
16    private Font boldFont; // fonte para texto em negrito
17    private Font italicFont; // fonte para texto em itálico
18    private Font boldItalicFont; // fonte para texto em negrito e itálico
19    private JRadioButton plainJRadioButton; // seleciona texto simples
20    private JRadioButton boldJRadioButton; // seleciona texto em negrito
21    private JRadioButton italicJRadioButton; // seleciona texto em itálico
22    private JRadioButton boldItalicJRadioButton; // negrito e itálico
23    private ButtonGroup radioGroup; // buttongroup para armazenar botões de opção
24
25    // construtor RadioButtonFrame adiciona JRadioButtons ao JFrame
26    public RadioButtonFrame()
27    {
28       super( "RadioButton Test" );
29       setLayout( new FlowLayout() ); // configura o layout de frame
30
31       textField = new JTextField( "Watch the font style change", 25 );
32       add( textField ); // adiciona textField ao JFrame
33
34       // cria botões de opção
35       plainJRadioButton = new JRadioButton( "Plain", true );
36       boldJRadioButton = new JRadioButton( "Bold", false );
37       italicJRadioButton = new JRadioButton( "Italic", false );
38       boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
39       add( plainJRadioButton ); // adiciona botão no estilo simples ao JFrame
40       add( boldJRadioButton ); // adiciona botão de negrito ao JFrame
41       add( italicJRadioButton ); // adiciona botão de itálico ao JFrame
42       add( boldItalicJRadioButton ); // adiciona botão de negrito e itálico
43
44       // cria relacionamento lógico entre JRadioButtons
45       radioGroup = new ButtonGroup(); // cria ButtonGroup
46       radioGroup.add( plainJRadioButton ); // adiciona simples ao grupo
47       radioGroup.add( boldJRadioButton ); // adiciona negrito ao grupo
48       radioGroup.add( italicJRadioButton ); // adiciona itálico ao grupo
49       radioGroup.add( boldItalicJRadioButton ); // adiciona negrito e itálico
50
51       // cria fonte objetos
52       plainFont = new Font( "Serif", Font.PLAIN, 14 );
53       boldFont = new Font( "Serif", Font.BOLD, 14 );
54       italicFont = new Font( "Serif", Font.ITALIC, 14 );
55       boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
56       textField.setFont( plainFont ); // configura a fonte inicial como simples
57
58       // registra eventos para JRadioButtons
59       plainJRadioButton.addItemListener(
60          new RadioButtonHandler( plainFont ) );
61       boldJRadioButton.addItemListener(
62          new RadioButtonHandler( boldFont ) );
63       italicJRadioButton.addItemListener(
64          new RadioButtonHandler( italicFont ) );
65       boldItalicJRadioButton.addItemListener(
66          new RadioButtonHandler( boldItalicFont ) );
67    } // fim do construtor RadioButtonFrame
68
69    // classe interna private para tratar eventos de botão de opção
70    private class RadioButtonHandler implements ItemListener
71    {
72       private Font font; // fonte associada com esse listener
```

```
73
74          public RadioButtonHandler( Font f )
75          {
76             font = f; // configura a fonte desse listener
77          } // fim do construtor RadioButtonHandler
78
79          // trata eventos de botão de opção
80          public void itemStateChanged( ItemEvent event )
81          {
82             textField.setFont( font ); // configura fonte de textField
83          } // fim do método itemStateChanged
84       } // fim da classe RadioButtonHandler interna private
85    } // fim da classe RadioButtonFrame
```

Figura 14.19 | `JRadioButtons` e `ButtonGroups`.

```
1  // Figura 14.20: RadioButtonTest.java
2  // Testando RadioButtonFrame.
3  import javax.swing.JFrame;
4
5  public class RadioButtonTest
6  {
7     public static void main( String[] args )
8     {
9        RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
10       radioButtonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       radioButtonFrame.setSize( 300, 100 ); // configura o tamanho do frame
12       radioButtonFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe RadioButtonTest
```

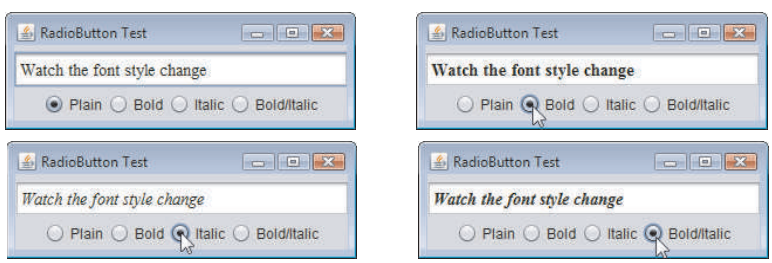


Figura 14.20 | Classe de teste para `RadioButtonFrame`.

As linhas 35–42 no construtor (Figura 14.19) criam quatro objetos `JRadioButton` e os adicionam ao `JFrame`. Cada `JRadioButton` é criado com uma chamada de construtor como aquela na linha 35. Esse construtor especifica o rótulo que aparece à direita do `JRadioButton` por padrão e o estado inicial do `JRadioButton`. Um segundo argumento `true` indica que o `JRadioButton` deve aparecer selecionado quando for exibido.

A linha 45 instancia o objeto `ButtonGroup` `radioGroup`. Esse objeto é a "cola" que forma o relacionamento lógico entre os quatro objetos `JRadioButton` e permite que somente um dos quatro seja selecionado por vez. É possível que nenhum `JRadioButton` em um `ButtonGroup` seja selecionado, mas isso pode ocorrer somente se nenhum `JRadioButton` pré-selecionado for adicionado ao `ButtonGroup` e o usuário ainda não tiver selecionado um `JRadioButton`. As linhas 46–49 utilizam o método `ButtonGroup` **`add`** para associar cada um dos `JRadioButtons` com `radioGroup`. Se mais de um objeto `JRadioButton` selecionado for adicionado ao grupo, aquele que tiver sido adicionado primeiro será selecionado quando a GUI for exibida.

`JRadioButtons`, como `JCheckBoxes`, geram `ItemEvents` quando se clica neles. As linhas 59–66 criam quatro instâncias de classe interna `RadioButtonHandler` (declaradas nas linhas 70–84). Nesse exemplo, cada objeto ouvinte de evento é registrado para tratar o `ItemEvent` gerado quando o usuário clica em um `JRadioButton` particular. Note que todo objeto `RadioButtonHandler` é inicializado com um objeto `Font` particular (criado nas linhas 52–55).

A classe `RadioButtonHandler` (linhas 70–84) implementa a interface `ItemListener` para que ela possa tratar `ItemEvents` gerados por `JRadioButtons`. O construtor armazena o objeto `Font` que ele recebe como um argumento na variável de instância `font` do objeto ouvinte de evento (declarada na linha 72). Quando o usuário clica em um `JRadioButton`, `radioGroup` desliga o `JRadioButton` anteriormente selecionado e o método `itemStateChanged` (linhas 80–83) configura o `JTextField` como a `Font` armazenada no objeto ouvinte de evento correspondente do `JRadioButton`. Observe que a linha 82 da classe interna `RadioButtonHandler` utiliza a variável de instância `textField` da classe de primeiro nível para configurar a fonte.

14.11 JComboBox e uso de uma classe interna anônima para tratamento de evento

Uma caixa de combinação (às vezes chamada de **lista drop-down**) permite ao usuário selecionar um item de uma lista (Figura 14.22). As caixas de combinação são implementadas com a classe **JComboBox**, que estende a classe `JComponent`. `JComboBoxes` geram `ItemEvents` como `JCheckBoxes` e `JRadioButtons`. Esse exemplo também demonstra uma forma especial de classe interna que costuma ser utilizada no tratamento de evento. O aplicativo (figuras 14.21–14.22) utiliza uma `JComboBox` para fornecer uma lista de quatro nomes de arquivo de imagem a partir da qual o usuário pode selecionar uma imagem para ser exibida. Quando o usuário seleciona um nome, o aplicativo exibe a imagem correspondente como um `Icon` em um `Jlabel`. A classe `ComboBoxTest` (Figura 14.22) contém o método `main` que executa esse aplicativo. As capturas de tela para esse aplicativo mostram a lista `JComboBox` depois que a seleção foi feita para ilustrar qual nome de arquivo de imagem foi selecionado.

```
1  // Figura 14.21: ComboBoxFrame.java
2  // JComboBox que exibe uma lista de nomes de imagem.
3  import java.awt.FlowLayout;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JComboBox;
9  import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14    private JComboBox imagesJComboBox; // caixa de combinação para armazenar nomes de ícones
15    private JLabel label; // rótulo para exibir ícone selecionado
16
17    private static final String[] names =
18       { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
19    private Icon[] icons = {
20       new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21       new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22       new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23       new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
25    // construtor ComboBoxFrame adiciona JComboBox ao JFrame
26    public ComboBoxFrame()
27    {
28       super( "Testing JComboBox" );
29       setLayout( new FlowLayout() ); // configura o layout de frame
30
31       imagesJComboBox = new JComboBox( names ); // configura JComboBox
32       imagesJComboBox.setMaximumRowCount( 3 ); // exibe três linhas
33
34       imagesJComboBox.addItemListener(
35          new ItemListener() // classe interna anônima
36          {
37             // trata evento JComboBox
38             public void itemStateChanged( ItemEvent event )
39             {
40                // determina se o item selecionado
41                if ( event.getStateChange() == ItemEvent.SELECTED )
42                   label.setIcon( icons[
43                      imagesJComboBox.getSelectedIndex() ] );
44             } // fim do método itemStateChanged
45          } // fim da classe interna anônima
46       ); // fim da chamada para addItemListener
47
48       add( imagesJComboBox ); // adiciona combobox ao JFrame
49       label = new JLabel( icons[ 0 ] ); // exibe primeiro ícone
50       add( label ); // adiciona rótulo ao JFrame
51    } // fim do construtor ComboBoxFrame
52 } // fim da classe ComboBoxFrame
```

Figura 14.21 | `JComboBox` que exibe uma lista de nomes de imagem.

```
1  // Figura 14.22: ComboBoxTest.java
2  // Testando ComboBoxFrame.
3  import javax.swing.JFrame;
4
5  public class ComboBoxTest
6  {
7     public static void main( String[] args )
8     {
9        ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
10       comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       comboBoxFrame.setSize( 350, 150 ); // configura o tamanho do frame
12       comboBoxFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe ComboBoxTest
```

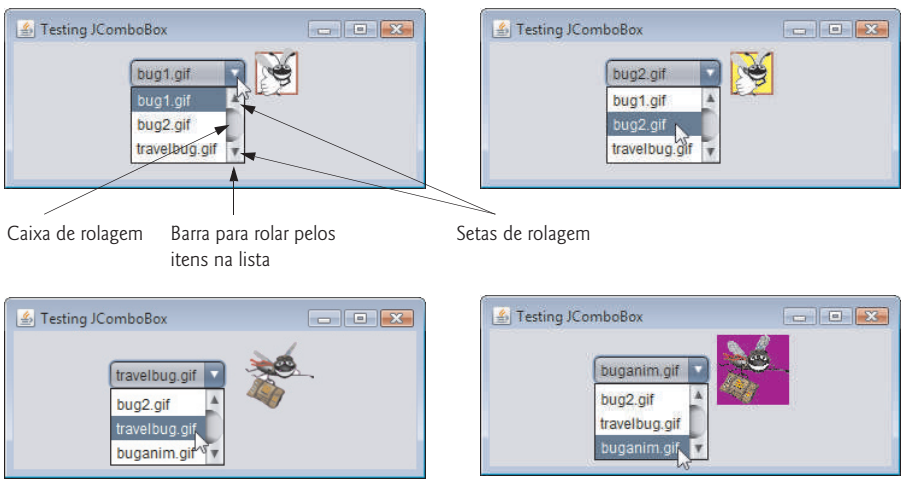


Figura 14.22 | Testando `ComboBoxFrame`.

As linhas 19–23 (Figura 14.21) declaram e inicializam o array `icons` com quatro novos objetos `ImageIcon`. O array `String` `names` (linhas 17–18) contém os nomes dos quatro arquivos de imagem que estão armazenados no mesmo diretório do aplicativo.

Na linha 31, o construtor inicializa um objeto `JComboBox`, com `String`s no array `names` como os elementos na lista. Todo item na lista tem um **índice**. O primeiro item é adicionado no índice 0, o próximo no índice 1 etc. O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida. Outros itens são selecionados clicando na `JComboBox`, então selecionando um item da lista que aparece.

A linha 32 usa o método `JComboBox` **`setMaximumRowCount`** para configurar o número máximo de elementos que é exibido quando o usuário clica em `JComboBox`. Se houver itens adicionais, a `JComboBox` fornece uma **barra de rolagem** (ver a primeira captura de tela) que permite ao usuário rolar por todos os elementos na lista. O usuário pode clicar nas **setas de rolagem** na parte superior e inferior da barra de rolagem para mover-se para cima e para baixo pela lista um elemento por vez ou então arrastar a **caixa de rolagem** no meio da barra de rolagem para cima e para baixo. Para arrastar a caixa de rolagem, posicione o cursor de mouse sobre ela, segure o botão do mouse e mova o mouse. [*Nota:* nesse exemplo, a lista drop-down é muito curta para arrastar a caixa de rolagem, portanto, você pode clicar nas setas para cima e para baixo ou utilizar a roda do mouse para rolar pelos quatro itens na lista.]

Observações sobre a aparência e comportamento 14.11

Configure a contagem máxima de linha para uma `JComboBox` como um número de linhas que impede a lista de expandir para fora dos limites da janela em que ela é utilizada.

A linha 48 anexa `JComboBox` ao `ComboBoxFrame` `FlowLayout` (linha posicionada 29). A linha 49 cria o `JLabel` que exibe `ImageIcon`s e o inicializa com o primeiro `ImageIcon` no array `icons`. A linha 50 anexa o `JLabel` ao `FlowLayout` do `ComboBoxFrame`.

Utilizando uma classe interna anônima para tratamento de evento

As linhas 34–46 são uma instrução que declara a classe do ouvinte de evento, cria um objeto dessa classe e registra-o como o ouvinte de `ItemEvents` da `imagesJComboBox`. O objeto ouvinte de evento é uma instância de uma **classe interna anônima** — uma classe interna que é declarada sem nome e, em geral, aparece dentro de uma declaração de método. Como com outras classes internas, uma classe interna anônima pode acessar os membros de sua classe de primeiro nível. Contudo, uma classe interna anônima tem acesso

limitado às variáveis locais do método em que ela é declarada. Como uma classe interna anônima não tem nome, deve-se criar um objeto da classe interna anônima no ponto em que a classe é declarada (iniciando na linha 35).

Observação de engenharia de software 14.5

Uma classe interna anônima declarada em um método pode acessar as variáveis de instância e métodos do objeto de classe de primeiro nível que a declararam, bem como as variáveis locais `final` do método, mas não pode acessar variáveis locais não `final` do método.

As linhas 34–46 são uma chamada ao método `addItemListener` da `imagesJComboBox`. O argumento para esse método deve ser um objeto que *é um* `ItemListener` (isto é, qualquer objeto de uma classe que implemente `ItemListener`). As linhas 35–45 são uma expressão de criação da instância de classe que declara uma classe interna anônima e cria um objeto dessa classe. Uma referência àquele objeto então é passada como o argumento para `addItemListener`. A sintaxe `ItemListener()` depois `new` inicia a declaração de uma classe interna anônima que implementa a interface `ItemListener`. Isso é semelhante a começar uma declaração de classe com

```
public class MyHandler implements ItemListener
```

Os parênteses depois de `ItemListener` indicam uma chamada ao construtor padrão da classe interna anônima.

A chave de abertura esquerda ({) na linha 36 e a chave de fechamento direita (}) na linha 45 delimitam o corpo da classe interna anônima. As linhas 38–44 declaram método `itemStateChanged` de `ItemListener`. Quando o usuário fizer uma seleção de `imagesJComboBox`, esse método configura `Icon` do `label`. O `Icon` é selecionado a partir do array `icons` determinando o índice do item selecionado na `JComboBox` com o método **`getSelectedIndex`** na linha 43. Observe que para cada item selecionado de um `JComboBox`, a seleção de outro item é primeiro removida — então ocorrem dois `ItemEvent`s quando um item é selecionado. Pretendemos exibir apenas o ícone do item que o usuário acabou de selecionar. Por essa razão, a linha 41 determina se o método `ItemEvent` **`getStateChange`** retorna `ItemEvent.SELECTED`. Se retornar, as linhas 42–43 configuram o ícone de `label`.

Observação de engenharia de software 14.6

Como qualquer outra classe, quando uma classe interna anônima implementa uma interface, a classe deve implementar cada método na interface.

A sintaxe mostrada nas linhas 35–45 para criar um handler de evento com uma classe interna anônima é semelhante ao código que seria gerado por um ambiente de desenvolvimento integrado Java (Integrated Development Environment — IDE). Em geral, um IDE permite-lhe projetar uma GUI visualmente; então, ele gera o código que implementa a GUI. Simplesmente insira instruções nos métodos de tratamento de evento que declaram a maneira como tratar cada evento.

14.12 JList

Uma lista exibe uma série de itens a partir da qual o usuário pode selecionar um ou mais itens (ver a saída da Figura 14.24). As listas são criadas com a classe `JList`, que estende diretamente a classe `JComponent`. A classe `JList` suporta **listas de uma única seleção** (que permitem que apenas um item seja selecionado por vez) e **listas de seleção múltipla** (que permitem que qualquer número de itens seja selecionado). Nesta seção, discutimos listas de uma única seleção.

O aplicativo das figuras 14.23–14.24 cria uma `JList` contendo 13 nomes de cores. Quando se clica em um nome de cor na `JList`, ocorre um **`ListSelectionEvent`** e o aplicativo muda a cor de fundo da janela de aplicativo para a cor selecionada. A classe `ListTest` (Figura 14.24) contém o método `main` que executa esse aplicativo.

```
1  // Figura 14.23: ListFrame.java
2  // JList que exibe uma lista de cores.
3  import java.awt.FlowLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JList;
7  import javax.swing.JScrollPane;
8  import javax.swing.event.ListSelectionListener;
9  import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class ListFrame extends JFrame
13 {
14    private JList colorJList; // lista para exibir cores
15    private static final String[] colorNames = { "Black", "Blue", "Cyan",
16       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
```

```
17          "Orange", "Pink", "Red", "White", "Yellow" };
18       private static final Color[] colors = { Color.BLACK, Color.BLUE,
19          Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20          Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21          Color.RED, Color.WHITE, Color.YELLOW };
22
23       // construtor ListFrame adiciona JScrollPane que contém JList ao JFrame
24       public ListFrame()
25       {
26          super( "List Test" );
27          setLayout( new FlowLayout() ); // configura o layout de frame
28
29          colorJList = new JList( colorNames ); // cria com colorNames
30          colorJList.setVisibleRowCount( 5 ); // exibe cinco linhas de uma vez
31
32          // não permite múltiplas seleções
33          colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
34
35          // adiciona um JScrollPane que contém JList ao frame
36          add( new JScrollPane( colorJList ) );
37
38          colorJList.addListSelectionListener(
39             new ListSelectionListener() // classe interna anônima
40             {
41                // trata eventos de seleção de lista
42                public void valueChanged( ListSelectionEvent event )
43                {
44                   getContentPane().setBackground(
45                      colors[colorJList.getSelectedIndex() ] );
46                } // fim do método valueChanged
47             } // fim da classe interna anônima
48          ); // fim da chamada para addListSelectionListener
49       } // fim do construtor ListFrame
50    } // fim da classe ListFrame
```

Figura 14.23 | `JList` que exibe uma lista de cores.

```
1  // Figura 14.24: ListTest.java
2  // Selecionando as cores de uma JList.
3  import javax.swing.JFrame;
4
5  public class ListTest
6  {
7     public static void main( String[] args )
8     {
9        ListFrame listFrame = new ListFrame(); // cria ListFrame
10       listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       listFrame.setSize( 350, 150 ); // configura o tamanho do frame
12       listFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe ListTest
```

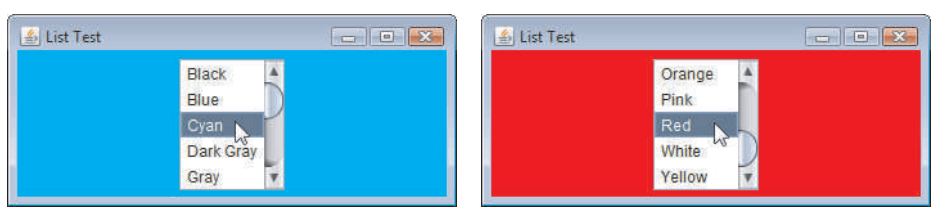


Figura 14.24 | Teste a classe `ListFrame`.

A linha 29 (Figura 14.23) cria o objeto `JList` `colorJList`. O argumento para o construtor `JList` é o array de `Objects` (nesse caso `Strings`) para exibir na lista. A linha 30 utiliza o método `JList` **`setVisibleRowCount`** para determinar o número de itens visíveis na lista.

A linha 33 utiliza o método `JList` **`setSelectionMode`** para especificar o **modo de seleção** da lista. A classe **`ListSelectionModel`** (do pacote `javax.swing`) declara três constantes que especificam o modo de seleção de uma `Jlist` — **`SINGLE_SELECTION`** (que permite que apenas um item seja selecionado por vez), **`SINGLE_INTERVAL_SELECTION`** (para uma lista de seleção múltipla que permite a seleção de vários itens contíguos) e **`MULTIPLE_INTERVAL_SELECTION`** (para uma lista de seleção múltipla que não restringe os itens que podem ser selecionados).

Ao contrário de uma `JComboBox`, uma `JList` *não* fornece uma barra de rolagem se houver mais itens na lista do que o número de linhas visíveis. Nesse caso, um objeto **`JScrollPane`** é utilizado para fornecer a capacidade de rolagem. A linha 36 adiciona uma nova instância da classe `JScrollPane` ao `JFrame`. O construtor `JScrollPane` recebe como seu argumento o `JComponent` que precisa de funcionalidades de rolagem (nesse caso, `colorJList`). Observe nas capturas de tela que uma barra de rolagem criada pelo `JScrollPane` aparece no lado direito da `JList`. Por padrão, a barra de rolagem só aparece quando o número de itens na `JList` excede o número de itens visíveis.

As linhas 38–48 usam o método `Jlist` **`addListSelectionListener`** para registrar um objeto que implementa **`ListSelectionListener`** (pacote `javax.swing.event`) como o listener para os eventos de seleção de `JList`. Mais uma vez, usamos uma instância de uma classe interna anônima (linhas 39–47) como o listener. Nesse exemplo, quando o usuário faz uma seleção de `colorJList`, o método **`valueChanged`** (linha 42–46) deve mudar a cor de fundo do `ListFrame` para a cor selecionada. Isso é realizado nas linhas 44–45. Observe o uso do método `JFrame` **`getContentPane`** na linha 44. Todo `JFrame` realmente consiste em três camadas — o fundo, o painel de conteúdo e o painel transparente. O painel de conteúdo aparece na frente do fundo e é onde os componentes GUI no `JFrame` são exibidos. O painel transparente é usado para exibir dicas de ferramenta e outros itens que devem aparecer na frente dos componentes GUI na tela. O painel de conteúdo oculta completamente o fundo do `JFrame`; então, para mudar a cor de fundo atrás dos componentes GUI, você deve mudar a cor de fundo do painel de conteúdo. O método `getContentPane` retorna uma referência ao painel de conteúdo do `JFrame` (um objeto da classe `Container`). Na linha 44, utilizamos depois essa referência para chamar o método **`setBackground`**, que configura a cor de fundo do painel de conteúdo como um elemento no array `colors`. A cor é selecionada a partir do array utilizando-se o índice do item selecionado. O método `JList` **`getSelectedIndex`** retorna o índice do item selecionado. Como com arrays e `JComboBoxes`, a indexação de `JList` é baseada em zero.

14.13 Listas de seleção múltipla

Uma **lista de seleção múltipla** permite ao usuário selecionar muitos itens de uma `JList` (ver a saída da Figura 14.26). Uma lista `SINGLE_INTERVAL_SELECTION` permite selecionar um intervalo contíguo de itens. Para fazer isso, clique no primeiro item, então pressione e mantenha a tecla *Shift* pressionada ao clicar no último item no intervalo. Uma lista `MULTIPLE_INTERVAL_SELECTION` (a padrão) permite a seleção de intervalo contínuo como descrito para uma lista `SINGLE_INTERVAL_SELECTION`. Uma lista como essa também permite que diversos itens sejam selecionados pressionando e mantendo a tecla *Ctrl* pressionada (às vezes chamada de tecla *Control*) ao clicar em cada item a ser selecionado. Para remover a seleção de um item, pressione e segure a tecla *Ctrl* ao clicar no item uma segunda vez.

O aplicativo das figuras 14.25–14.26 utiliza listas de seleção múltipla para copiar itens de uma `JList` para outra. Uma lista é uma lista `MULTIPLE_INTERVAL_SELECTION` e a outra é uma lista `SINGLE_INTERVAL_SELECTION`. Ao executar o aplicativo, tente utilizar as técnicas de seleção descritas anteriormente para selecionar itens das duas listas.

```
1  // Figura 14.25: MultipleSelectionFrame.java
2  // Copiando itens de uma Lista para outra.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JList;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14    private JList colorJList; // lista para armazenar nomes de cor
15    private JList copyJList; // lista para copiar nomes de cor em
16    private JButton copyJButton; // botão para copiar nomes selecionados
17    private static final String[] colorNames = { "Black", "Blue", "Cyan",
18       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19       "Pink", "Red", "White", "Yellow" };
20
21    // construtor MultipleSelectionFrame
22    public MultipleSelectionFrame()
23    {
24       super( "Multiple Selection Lists" );
```

```
25          setLayout( new FlowLayout() ); // configura o layout de frame
26
27          colorJList = new JList( colorNames ); // armazena nomes de todas as cores
28          colorJList.setVisibleRowCount( 5 ); // mostra cinco linhas
29          colorJList.setSelectionMode(
30             ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
31          add( new JScrollPane( colorJList ) ); // adiciona lista com scrollpane
32
33          copyJButton = new JButton( "Copy >>>" ); // cria botão de cópia
34          copyJButton.addActionListener(
35
36             new ActionListener() // classe interna anônima
37             {
38                // trata evento de botão
39                public void actionPerformed( ActionEvent event )
40                {
41                   // coloca valores selecionados na copyJList
42                   copyJList.setListData( colorJList.getSelectedValues() );
43                } // fim do método actionPerformed
44             } // fim da classe interna anônima
45          ); // fim da chamada para addActionListener
46
47          add( copyJButton ); // adiciona botão de cópia ao JFrame
48
49          copyJList = new JList(); // cria lista para armazenar nomes de cor copiados
50          copyJList.setVisibleRowCount( 5 ); // mostra 5 linhas
51          copyJList.setFixedCellWidth( 100 ); // configura a largura
52          copyJList.setFixedCellHeight( 15 ); // configura a altura
53          copyJList.setSelectionMode(
54             ListSelectionModel.SINGLE_INTERVAL_SELECTION );
55          add( new JScrollPane( copyJList ) ); // adiciona lista com scrollpane
56       } // fim do construtor MultipleSelectionFrame
57    } // fim da classe MultipleSelectionFrame
```

Figura 14.25 | `JList` que permite múltiplas seleções.

```
1  // Figura 14.26: MultipleSelectionTest.java
2  // Testando MultipleSelectionFrame.
3  import javax.swing.JFrame;
4
5  public class MultipleSelectionTest
6  {
7     public static void main( String[] args )
8     {
9        MultipleSelectionFrame multipleSelectionFrame =
10          new MultipleSelectionFrame();
11       multipleSelectionFrame.setDefaultCloseOperation(
12          JFrame.EXIT_ON_CLOSE );
13       multipleSelectionFrame.setSize( 350, 150 ); // configura o tamanho do frame
14       multipleSelectionFrame.setVisible( true ); // exibe o frame
15    } // fim de main
16 } // fim da classe MultipleSelectionTest
```

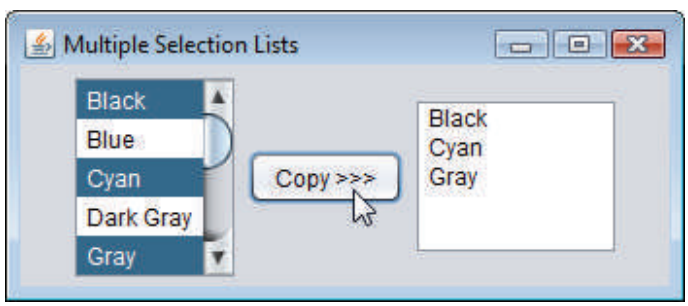


Figura 14.26 | Teste a classe `MultipleSelectionFrame`.

A linha 27 da Figura 14.25 cria `JList colorJList` e a inicializa com as `Strings` no array `colorNames`. A linha 28 configura o número de linhas visíveis em `colorJList` como `5`. As linhas 29–30 especificam que `colorJList` é uma lista `MULTIPLE_INTERVAL_SELECTION`. A linha 31 adiciona um novo `JScrollPane` contendo `colorJList` ao `JFrame`. As linhas 49–55 executam tarefas semelhantes para a `copyJList`, que é declarada como uma lista `SINGLE_INTERVAL_SELECTION`. Se uma `JList` não contiver itens ela não será exibida em um `FlowLayout`. Por essa razão, as linhas 51–52 utilizam os métodos `JList` **`setFixedCellWidth`** e **`setFixedCellHeight`** para configurar a largura da `copyJList` como 100 pixels e a altura de cada item na `JList` como 15 pixels, respectivamente.

Não há nenhum evento para indicar que um usuário fez múltiplas seleções em uma lista de seleção múltipla. Normalmente, um evento gerado por outro componente GUI (conhecido como um **evento externo**) especifica quando as múltiplas seleções em uma `JList` devem ser processadas. Nesse exemplo, o usuário clica no `JButton` chamado `copyJButton` para desencadear o evento que copia os itens selecionados em `colorJList` para `copyJList`.

As linhas 34–45 declaram, criam e registram um `ActionListener` para o `copyJButton`. Quando o usuário clica em `copyJButton`, o método `actionPerformed` (linhas 39–43) utiliza o método `JList` **`setListData`** para configurar os itens exibidos em `copyJList`. A linha 42 chama o método **`getSelectedValues`** da `colorJList`, que retorna um array de `Objects` para representar os itens selecionados em `colorJList`. Nesse exemplo, o array retornado é passado como o argumento para o método `setListData` de `copyJList`.

Você poderia estar se perguntando por que `copyJList` pode ser utilizada na linha 42 embora o aplicativo não crie o objeto referenciado por `copyJList` até a linha 49. Lembre-se de que o método `actionPerformed` (linhas 39–43) não executa até que o usuário pressione `copyJButton`, o que não pode ocorrer até que o construtor complete a execução e o aplicativo exiba a GUI. Nesse ponto na execução do aplicativo, `copyJList` já é inicializado com um novo objeto `Jlist`.

14.14 Tratamento de evento de mouse

Esta seção apresenta as interfaces ouvintes de evento **`MouseListener`** e **`MouseMotionListener`** para tratar **eventos de mouse**. Os eventos de mouse podem ser capturados por qualquer componente GUI que deriva de `java.awt.Component`. Os métodos de interfaces `MouseListener` e `MouseMotionListener` são resumidos na Figura 14.27 O pacote `javax.swing.event` contém a interface **`MouseInputListener`**, que estende as interfaces `MouseListener` e `MouseMotionListener` para criar uma única interface que contém todos os métodos `MouseListener` e `MouseMotionListener`. Os métodos `MouseListener` e `MouseMotionListener` são chamados quando o mouse interage com um `Component` se objetos listeners de evento apropriados forem registrados para esse `Component`.

Métodos de interface `MouseListener` e `MouseMotionListener`

Métodos de interface `MouseListener`

`public void mousePressed( MouseEvent event )`

Chamado quando um botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente.

`public void mouseClicked( MouseEvent event )`

Chamado quando um botão do mouse é pressionado e liberado enquanto o cursor do mouse pairar sobre um componente. Esse evento é sempre precedido por uma chamada para `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Chamado quando um botão do mouse é liberado depois de ser pressionado. Esse evento sempre é precedido por uma chamada para `mousePressed` e uma ou mais chamadas para `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

Chamado quando o cursor do mouse entra nos limites de um componente.

`public void mouseExited( MouseEvent event )`

Chamado quando o cursor do mouse deixa os limites de um componente.

Métodos de interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Chamado quando o botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente e o mouse é movido enquanto o botão do mouse permanecer pressionado. Esse evento é sempre precedido por uma chamada para `mousePressed`. Todos os eventos de arrastar são enviados para o componente em que o usuário começou a arrastar o mouse.

`public void mouseMoved( MouseEvent event )`

Chamado quando o mouse é movido (sem pressionamentos de botões do mouse) quando o cursor do mouse está sobre um componente. Todos os eventos de movimento são enviados para o componente sobre o qual o mouse atualmente está posicionado.

Figura 14.27 | Métodos de interface `MouseListener` e `MouseMotionListener`.

Cada um dos métodos de tratamento de evento de mouse recebe como um argumento um objeto **`MouseEvent`** que contém informações sobre o evento de mouse que ocorreu, incluindo as coordenadas *x* e *y* da localização em que ocorreu o evento. Essas coordenadas são medidas do canto superior esquerdo do componente GUI em que o evento ocorreu. As coordenadas *x* iniciam em 0 e aumentam da esquerda para a direita. As coordenadas *y* iniciam em 0 e aumentam de cima para baixo. Os métodos e as constantes de classe **`InputEvent`** (superclasse de `MouseEvent`) permitem-lhe determinar o botão do mouse em que o usuário clicou.

Observação de engenharia de software 14.7

As chamadas de método para `mouseDragged` são enviadas para o `MouseMotionListener` do `Component` em que a operação de arrastar o mouse se iniciou. De maneira semelhante, a chamada de método `mouseReleased` no fim de uma operação de arrastar é enviada para o `MouseListener` do `Component` em que a operação de arrastar iniciou.

O Java também fornece a interface **`MouseWheelListener`** para permitir que aplicativos respondam à rotação da roda de um mouse. Essa interface declara o método **`mouseWheelMoved`**, que recebe um **`MouseWheelEvent`** como seu argumento. A classe `MouseWheelEvent` (uma subclasse de `MouseEvent`) contém métodos que permitem que o handler de evento obtenha as informações sobre a quantidade de rotação de roda.

Monitorando eventos de mouse em um `JPanel`

O aplicativo `MouseTracker` (figuras 14.28–14.29) demonstra os métodos de interface `MouseListener` e `MouseMotionListener`. A classe de handler de evento (linhas 36–90) implementa ambas as interfaces. Observe que você deve declarar os sete métodos dessas duas interfaces quando a classe implementar as duas interfaces. Cada evento de mouse nesse exemplo exibe uma `String` no `JLabel` chamada `statusBar` que é anexada à parte inferior da janela.

```
1  // Figura 14.28: MouseTrackerFrame.java
2  // Demonstrando os eventos de mouse.
3  import java.awt.Color;
4  import java.awt.BorderLayout;
5  import java.awt.event.MouseListener;
6  import java.awt.event.MouseMotionListener;
7  import java.awt.event.MouseEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14    private JPanel mousePanel; // painel em que os eventos de mouse ocorrerão
15    private JLabel statusBar; // rótulo que exibe informações de evento
16
17    // construtor MouseTrackerFrame configura GUI e
18    // registra handlers de evento de mouse
19    public MouseTrackerFrame()
20    {
21       super( "Demonstrating Mouse Events" );
22
23       mousePanel = new JPanel(); // cria painel
24       mousePanel.setBackground( Color.WHITE ); // configura cor de fundo
25       add( mousePanel, BorderLayout.CENTER ); // adiciona painel ao JFrame
26
27       statusBar = new JLabel( "Mouse outside JPanel" );
28       add( statusBar, BorderLayout.SOUTH ); // adiciona rótulo ao JFrame
29
30       // cria e registra listener para mouse e eventos de movimento de mouse
31       MouseHandler handler = new MouseHandler();
32       mousePanel.addMouseListener( handler );
33       mousePanel.addMouseMotionListener( handler );
34    } // construtor de MouseTrackerFrame de fim
35
36    private class MouseHandler implements MouseListener,
37       MouseMotisnListener
38    {
39       // Handlers de evento de MouseListener
40       // trata evento quando o mouse é liberado imediatamente depois de ter sido pressionado
41       public void mouseClicked( MouseEvent event )
```

```
42          {
43             statusBar.setText( String.format( "Clicked at [%d, %d]",
44                event.getX(),event.getY() ) );
45          } // fim do método mouseClicked
46
47          // trata evento quando mouse é pressionado
48          public void mousePressed( MouseEvent event )
49          {
50             statusBar.setText( String.format( "Pressed at [%d, %d]",
51                event.getX(),event.getY() ) );
52          } // fim do método mousePressed
53
54          // trata o evento quando o mouse é liberado
55          public void mouseReleased( MouseEvent event )
56          {
57             statusBar.setText( String.format( "Released at [%d, %d]",
58                event.getX(),event.getY() ) );
59          } // fim do método mouseReleased
60
61          // trata evento quando mouse entra na área
62          public void mouseEntered( MouseEvent event )
63          {
64             statusBar.setText( String.format( "Mouse entered at [%d, %d]",
65                event.getX(),event.getY() ) );
66             mousePanel.setBackground( Color.GREEN );
67          } // fim do método mouseEntered
68
69          // trata evento quando mouse sai da área
70          public void mouseExited( MouseEvent event )
71          {
72             statusBar.setText( "Mouse outside JPanel" );
73             mousePanel.setBackground( Color.WHITE );
74          } // mouseExited fim do método
75
76          // Handlers de evento de MouseMotionListener
77          // trata evento quando usuário arrasta o mouse com o botão pressionado
78          public void mouseDragged( MouseEvent event )
79          {
80             statusBar.setText( String.format( "Dragged at [%d, %d]",
81                event.getX(),event.getY() ) );
82          } // fim do método mouseDragged
83
84          // trata evento quando usuário move o mouse
85          public void mouseMoved( MouseEvent event )
86          {
87             statusBar.setText( String.format( "Moved at [%d, %d]",
88                event.getX(),event.getY() ) );
89          } // fim do método mouseMoved
90       } // fim da classe interna MouseHandler
91    } // fim da classe MouseTrackerFrame
```

Figura 14.28 | Tratamento de evento de mouse.

```
1  // Figura 14.29: MouseTrackerFrame.java
2  // Testando MouseTrackerFrame.
3  import javax.swing.JFrame;
4
5  public class MouseTracker
6  {
7     public static void main( String[] args )
8     {
9        MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
10       mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       mouseTrackerFrame.setSize( 300, 100 ); // configura o tamanho do frame
12       mouseTrackerFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe MouseTracker
```

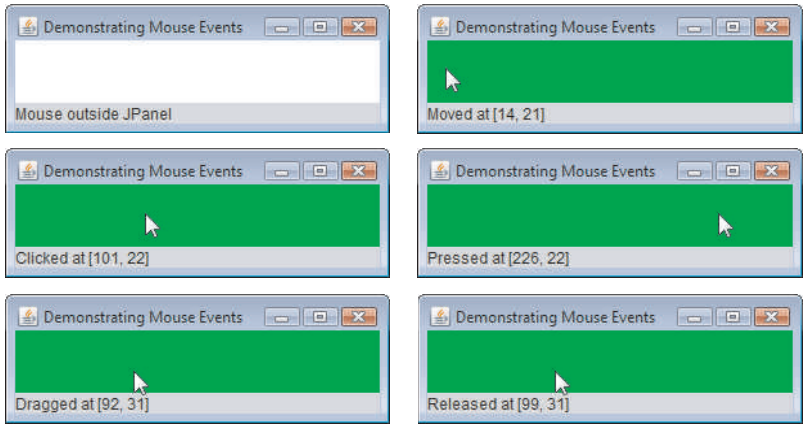


Figura 14.29 | Teste a classe `MouseTrackerFrame`.

A linha 23 na Figura 14.28 cria `JPanel mousePanel`. Os eventos de mouse desse `JPanel` serão monitorados pelo aplicativo. A linha 24 configura a cor de fundo do `mousePanel` como branco. Quando o usuário mover o mouse no `mousePanel`, o aplicativo irá mudar a cor de fundo do `mousePanel` para verde. Quando o usuário mover o mouse fora do `mousePanel`, o aplicativo mudará a cor de fundo para branco. A linha 25 anexa o `mousePanel` ao `JFrame`. Como você aprendeu na Seção 14.5, geralmente deve-se especificar o layout dos componentes GUI em um `JFrame`. Nessa seção, introduzimos o gerenciador de layout `FlowLayout`. Aqui usamos o layout padrão do painel de conteúdo de um `JFRAME` — **`BorderLayout`**. Esse gerenciador de layout organiza componentes em cinco regiões: **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** e **`CENTER`**. `NORTH` corresponde à parte superior do contêiner. Esse exemplo utiliza as regiões `CENTER` e `SOUTH`. A linha 25 utiliza uma versão de dois argumentos do método `add` para colocar `mousePanel` na região `CENTER`. O `BorderLayout` dimensiona o componente no `CENTER` automaticamente para utilizar todo o espaço em `JFrame` que não é ocupado pelos componentes nas outras regiões. A Seção 14.18.2 discute `BorderLayout` mais detalhadamente.

As linhas 27–28 no construtor declaram `JLabel statusBar` e o anexam à região `SOUTH` de `JFrame`. Esse `JLabel` ocupa a largura do `JFrame`. A altura da região é determinada pelo `JLabel`.

A linha 31 cria uma instância da classe interna `MouseHandler` (linhas 36–90) chamada `handler` que responde aos eventos de mouse. As linhas 32–33 registram `handler` como o ouvinte de evento de mouse do `mousePanel`. Os métodos **`addMouseListener`** e **`addMouseMotionListener`** são herdados indiretamente da classe `Component` e podem ser usados para registrar `MouseListeners` e `MouseMotionListeners`, respectivamente. Um objeto `MouseHandler` *é um* `MouseListener` e *é um* `MouseMotionListener` porque a classe implementa ambas as interfaces. [Observe que escolhemos implementar ambas as interfaces aqui para demonstrar uma classe que implementa mais de uma interface, mas poderíamos ter implementado a interface `MouseInputListener` em vez disso.]

Quando o mouse entrar e sair da área de `mousePanel`, os métodos `mouseEntered` (linhas 62–67) e `mouseExited` (linhas 70–74) são chamados, respectivamente. O método `mouseEntered` exibe uma mensagem no `statusBar` que indica que o mouse entrou em `JPanel` e muda a cor de fundo para verde. O método `mouseExited` exibe uma mensagem no `statusBar` que indica que o mouse está fora do `JPanel` (ver a primeira janela de saída de exemplo) e muda a cor de fundo para branco.

Quando qualquer um dos outros cinco eventos ocorrer, ele exibe uma mensagem no `statusBar` que inclui uma string contendo o evento e as coordenadas em que ele ocorreu. Os métodos `MouseEvent` **`getX`** e **`getY`** retornam as coordenadas *x* e *y*, respectivamente, do mouse no momento em que o evento ocorreu.

14.15 Classes adaptadoras

Muitas interfaces listener de eventos, como `MouseListener` e `MouseMotionListener`, contêm múltiplos métodos. Não é sempre desejável declarar cada método em uma interface ouvinte de evento. Por exemplo, um aplicativo pode precisar apenas da rotina de tratamento de evento `mouseClicked` a partir do `MouseListener` ou da rotina `mouseDragged` a partir do `MouseMotionListener`. A interface `WindowListener` especifica sete métodos de tratamento de evento de janela. Para muitas das interfaces listener que têm vários métodos, os pacotes `java.awt.event` e `javax.swing.event` fornecem classes adaptadoras listeners de evento. Uma **classe adaptadora** implementa uma interface e fornece uma implementação padrão (com o corpo de um método vazio) de cada método na interface. A Figura 14.30 mostra várias classes adaptadoras `java.awt.event` e as interfaces que elas implementam. Você pode estender uma classe adaptadora para herdar a implementação padrão de cada método e, subsequentemente, sobrescrever somente o(s) método(s) necessário(s) para o tratamento de evento.

Observação de engenharia de software 14.8

Quando uma classe implementar uma interface, a classe terá um relacionamento é um *com essa interface. Todas as subclasses diretas e indiretas dessa classe herdam essa interface. Portanto, um objeto de uma classe que estende uma classe adaptadora de evento é um objeto do tipo ouvinte de evento correspondente (por exemplo, um objeto de uma subclasse de* `MouseAdapter` *é um* `MouseListener`*).*

Classe adaptadora de evento no `java.awt.event`	Implementa a interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Figura 14.30 | Classes adaptadoras de evento e as interfaces que elas implementam no pacote `java.awt.event`.

Estendendo `MouseAdapter`

O aplicativo das figuras 14.31 e 14.32 demonstra como determinar o número de cliques de mouse (isto é, a contagem de cliques) e como distinguir entre os diferentes botões do mouse. O ouvinte de evento nesse aplicativo é um objeto da classe interna `MouseClickHandler` (linhas 25–45) que estende `MouseAdapter`, então podemos declarar somente o método `mouseClicked` de que precisamos nesse exemplo.

```
1  // Figura 14.31: MouseDetailsFrame.java
2  // Demonstrando cliques de mouse e distinguindo entre botões do mouse.
3  import ons.awt.BorderLayout;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import javax.swing.Jframe;
7  import javax.swing.Jlabel;
8
9  public class MouseDetailsFrame extends Jframe
10 {
11    private String details; // A string que é exibida na barra de status
12    private Jlabel statusBar; // Jlabel que aparece na parte inferior da janela
13
14    // construtor configura String de barra de título e registra o listener de mouse
15    public MouseDetailsFrame()
16    {
17       super( "Mouse clicks and buttons" );
18
19       statusBar = new Jlabel( "Click the mouse" );
20       add( statusBar, BorderLayout.SOUTH );
21       addMouseListener( new MouseClickHandler() ); // adiciona handler
22    } // fim do construtor MouseDetailsFrame
23
24    // classe interna para tratar eventos de mouse
25    private class MouseClickHandler extends MouseAdapter
26    {
27       // trata evento de clique de mouse e determina qual botão foi pressionado
28       public void mouseClicked( MouseEvent event )
29       {
30          int xPos = event.getX(); // obtém a posição x do mouse
31          int yPos = event.getY(); // obtém a posição y do mouse
32
33          details = String.format( "Clicked %d time(s)",
34             event.getClickCount() );
35
36          if (event.isMetaDown() ) // botão direito do mouse
37             details += " with right mouse button";
38          else if (event.isAltDown() ) // botão do meio do mouse
39             details += " with center mouse button";
40          else // botão esquerdo do mouse
41             details += " with left mouse button";
42
43          statusBar.setText( details ); // exibe mensagem em statusBar
44       } // fim do método mouseClicked
45    } // fim da classe interna private MouseClickHandler
46 } // fim da classe MouseDetailsFrame
```

Figura 14.31 | Clique dos botões esquerdo, central e direito do mouse.

```
1  // Figura 14.32: MouseDetails.java
2  // Testando MouseDetailsFrame.
3  import javax.swing.Jframe;
4
5  public class MouseDetails
6  {
7     public static void main( String[] args )
8     {
9        MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10       mouseDetailsFrame.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
11       mouseDetailsFrame.setSize( 400, 150 ); // configura o tamanho do frame
12       mouseDetailsFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe MouseDetails
```

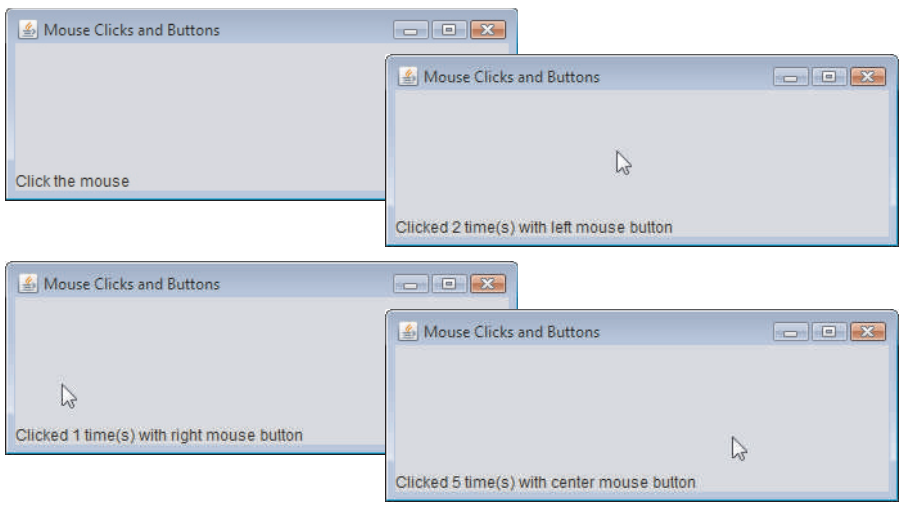


Figura 14.32 | Teste a classe `MouseDetailsFrame`.

Erro comum de programação 14.4

Se você estender uma classe adaptadora e digitar incorretamente o nome do método que você está sobrescrevendo, o método simplesmente torna-se outro método na classe. Esse é um erro de lógica difícil de ser detectado, visto que o programa chamará a versão vazia do método herdado da classe adaptadora.

Um usuário de um programa Java pode estar em um sistema com um, dois ou três botões do mouse. O Java fornece um mecanismo para distinguir os botões do mouse. A classe `MouseEvent` herda vários métodos da classe `InputEvent` que podem distinguir entre o botão do mouse em um mouse de múltiplos botões ou podem simular um mouse de múltiplos botões com uma combinação de um clique do teclado e um clique de botão do mouse. A Figura 14.33 mostra os métodos `InputEvent` utilizados para distinguir os cliques do botão do mouse. O Java assume que cada mouse contém um botão do mouse esquerdo. Portanto, é simples testar um clique com o botão esquerdo do mouse. Entretanto, os usuários com mouse de um ou dois botões devem utilizar uma combinação de pressionamentos de tecla e cliques do botão

Método `InputEvent`	Descrição
`isMetaDown()`	Retorna `true` quando o usuário clica no botão direito do mouse em um mouse com dois ou três botões. Para simular um clique de botão direito com um mouse de um botão, o usuário pode manter pressionada a tecla *Meta* no teclado e clicar no botão do mouse.
`isAltDown()`	Retorna `true` quando o usuário clica no botão do mouse do meio em um mouse com três botões. Para simular um clique com o botão do meio do mouse em um mouse com um ou dois botões, o usuário pode pressionar a tecla *Alt* e clicar no único botão ou no botão esquerdo do mouse, respectivamente.

Figura 14.33 | Métodos `InputEvent` que ajudam a distinguir entre os cliques do botão esquerdo, do centro e direito do mouse.

do mouse ao mesmo tempo para simular os botões ausentes no mouse. No caso de um mouse com um ou dois botões, um aplicativo Java assume que o botão do meio do mouse é clicado se o usuário mantém pressionada a tecla *Alt* e clica no botão esquerdo em um mouse de dois botões ou no único botão em um mouse de um único botão. Em caso de um mouse com um único botão, um aplicativo Java supõe que o botão do mouse direito é clicado se o usuário mantiver a tecla *Meta* pressionada (às vezes chamada de tecla *Command* ou tecla "Maçã" em um Mac) e clicar no botão do mouse.

A linha 21 da Figura 14.31 registra um `MouseListener` para o `MouseDetailsFrame`. O ouvinte de evento é um objeto da classe `MouseClickHandler`, que estende `MouseAdapter`. Isso permite declarar apenas o método `mouseClicked` (linhas 28–44). Esse método primeiro captura as coordenadas em que o evento ocorreu e as armazena nas variáveis locais `xPos` e `yPos` (linhas 30–31). As linhas 33–34 criam uma `String` chamada `details` contendo o número de cliques consecutivos de mouse, que é retornado pelo método `MouseEvent` **`getClickCount`** na linha 34. As linhas 36–41 utilizam os métodos **`isMetaDown`** e **`isAltDown`** para determinar o botão do mouse em que o usuário clicou e acrescentar uma `String` adequada aos `details` em cada caso. A `String` resultante é exibida em `statusBar`. A classe `MouseDetails` (Figura 14.32) contém o método `main` que executa o aplicativo. Tente clicar repetidamente com cada um dos botões do mouse para ver o incremento de contagem de cliques.

14.16 Subclasse Jpanel para desenhar com o mouse

A Seção 14.14 mostrou como monitorar eventos de mouse em um `Jpanel`. Nesta seção, usamos um `Jpanel` como uma **área dedicada de desenho** em que o usuário pode desenhar arrastando o mouse. Além disso, esta seção demonstra um ouvinte de evento que estende uma classe adaptadora.

Método `paintComponent`

Os componentes Swing leves que estendem a classe `Jcomponent` (como `Jpanel`) contêm o método **`paintComponent`**, que é chamado quando um componente Swing leve é exibido. Por sobrescrever esse método, você pode especificar como desenhar formas utilizando capacidades de imagens gráficas do Java. Ao personalizar um `Jpanel` para uso como uma área dedicada de desenho, a subclasse deve sobrescrever o método `paintComponent` e chamar a versão de superclasse de `paintComponent` como a primeira instrução no corpo do método sobrescrito para assegurar que o componente será exibido corretamente. A razão é que as subclasses de `Jcomponent` suportam a **transparência**. Para exibir um componente corretamente, o programa deve determinar se o componente é transparente. O código que determina isso está na implementação `paintComponent` da superclasse `Jcomponent`. Quando um componente for transparente, `paintComponent` não limpará seu fundo quando o programa exibir o componente. Quando um componente for **opaco**, `paintComponent` limpará o fundo do componente antes de o componente ser exibido. Se a versão de superclasse de `paintComponent` não for chamada, geralmente um componente GUI opaco não será exibido corretamente na interface com o usuário. Além disso, se a versão de superclasse é chamada depois de realizar as instruções de desenho personalizadas, os resultados geralmente serão apagados. A transparência de um componente Swing leve pode ser configurada com o método **`setOpaque`** (um argumento `false` indica que o componente é transparente).

Observações sobre a aparência e comportamento 14.12

A maioria dos componentes Swing GUI pode ser transparente ou opaca. Se um componente Swing GUI for opaco, seu fundo será limpo quando seu método `paintComponent` for chamado. Apenas os componentes opacos podem ser exibidos com uma cor de fundo personalizada. Os objetos `Jpanel` são opacos por padrão.

Dica de prevenção de erro 14.1

No método `paintComponent` de uma subclasse `Jcomponent`, a primeira instrução deve sempre chamar o método da superclasse `paintComponent` a fim de assegurar que um objeto da subclasse seja exibido corretamente.

Erro comum de programação 14.5

Se um método `paintComponent` sobrescrito não chamar a versão da superclasse, o componente de subclasse pode não ser exibido adequadamente. Se um método `paintComponent` sobrescrito chamar a versão da superclasse depois que outro desenho for realizado, o desenho será apagado.

Definindo a área personalizada de desenho

O aplicativo `Painter` das figuras 14.34 e 14.35 demonstra uma subclasse personalizada de `Jpanel` que é utilizada para criar uma área dedicada de desenho. O aplicativo utiliza o handler de evento `mouseDragged` para criar um aplicativo de desenho simples. O usuário pode desenhar figuras arrastando o mouse sobre o `Jpanel`. Esse exemplo não utiliza o método `mouseMoved`, então nossa classe ouvinte de evento (a classe interna anônima nas linhas 22–34) estende `MouseMotionAdapter`. Visto que essa classe já declara tanto `mouseMoved` como `mouseDragged`, podemos simplesmente sobrescrever `mouseDragged` para fornecer o tratamento de evento requerido por esse aplicativo.

```
1  // Figura 14.34: PaintPanel.java
2  // Utilizando a classe MouseMotionAdapter.
3  import ons.awt.Point;
4  import ons.awt.Graphics;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.MouseMotionAdapter;
7  import javax.swing.Jpanel;
8
9  public class PaintPanel extends Jpanel
10 {
11    private int pointCount = 0; // número de contagem de pontos
12
13    // array de 10000 referências ons.awt.Point
14    private Point[] points = new Point[ 10000 ];
15
16    // configura GUI e registra handler de evento de mouse
17    public PaintPanel()
18    {
19       // trata evento de movimento de mouse do frame
20       addMouseMotionListener(
21
22          new MouseMotionAdapter() // classe interna anônima
23          {
24             // armazena coordenadas de arrastar e repinta
25             public void mouseDragged( MouseEvent event )
26             {
27                if ( pointCount < points.length )
28                {
29                   points[ pointCount ] = event.getPoint(); // localiza o ponto
30                   pointCount++; // incrementa número de pontos em array
31                   repaint(); // repinta Jframe
32                } // fim do if
33             } // fim do método mouseDragged
34          } // fim da classe interna anônima
35       ); // fim da chamada para addMouseMotionListener
36    } // fim do construtor PaintPanel
37
38    // desenha ovais em um quadro delimitador de 4 por 4 nas localizações especificadas na janela
39    public void paintComponent( Graphics g )
40    {
41       super.paintComponent( g ); // limpa a área de desenho
42
43       // desenha todos os pontos no array
44       for ( int i = 0; i < pointCount; i++ )
45          g.fillOval(points[ i ].x, points[ i ].y, 4, 4 );
46    } // fim do método paintComponent
47 } // fim da classe PaintPanel
```

Figura 14.34 | Classe adaptadora utilizada para implementar handlers de evento.

A classe `PaintPanel` (Figura 14.34) estende `Jpanel` para criar a área dedicada de desenho. As linhas 3–7 importam as classes utilizadas na classe `PaintPanel`. A classe **`Point`** (pacote java.awt) representa uma coordenada x-y. Utilizamos objetos dessa classe para armazenar as coordenadas de cada evento de arrastar com o mouse. A classe **`Graphics`** é utilizada para desenhar.

Nesse exemplo, utilizamos um array de 10.000 `Points` (linha 14) para armazenar a localização em que cada evento de arrastar com o mouse ocorre. Como você verá, o método `paintComponent` utiliza esses `Points` para desenhar. A variável de instância `pointCount` (linha 11) mantém o número total de `Points` capturados a partir dos eventos de arrastar com o mouse até agora.

As linhas 20–35 registram um `MouseMotionListener` para ouvir os eventos de movimento do mouse `PaintPanel`. As linhas 22–34 criam um objeto de uma classe interna anônima que estende a classe adaptadora `MouseMotionAdapter`. Lembre-se de que `MouseMotionAdapter` implementa `MouseMotionListener`, portanto o objeto de classe interna anônima é um `MouseMotionListener`. A classe interna anônima herda as implementações padrão `mouseMoved` e `mouseDragged`, portanto já implementa métodos de toda a interface. Entretanto, os métodos padrão não fazem nada quando são chamados. Portanto, sobrescrevemos o método `mouseDragged` nas linhas 25–33 para capturar as coordenadas de um evento de arrastar com o mouse e as armazenamos como um objeto `Point`. A linha 27 assegura que armazenamos as coordenadas do evento somente se ainda houver elementos vazios no array. Se houver, a linha 29 invoca o método **`getPoint`** de `MouseEvent` para obter `Point` em que o evento ocorreu e armazená-lo no array no índice `pointCount`. A linha 30 incrementa o `pointCount` e a linha 31 chama o método **`repaint`** (herdado indiretamente da classe `Component`) para indicar que o `PaintPanel` deve ser atualizado na tela o mais rápido possível com uma chamada para o método `paintComponent` de `PaintPanel`.

O método `paintComponent` (linhas 39–46), que recebe um parâmetro `Graphics`, é chamado automaticamente a qualquer hora que `PaintPanel` precisar ser exibido na tela (como quando a GUI é inicialmente exibida) ou atualizado na tela (como quando o método `repaint` é chamado ou quando o componente GUI foi oculto por outra janela na tela e subsequentemente torna-se visível novamente).

Observações sobre a aparência e comportamento 14.13

Chamar `repaint` para um componente Swing GUI indica que o componente deve ser atualizado na tela o mais rápido possível. O fundo do componente GUI é limpo somente se o componente for opaco. Para o método `Jcomponent` `setOpaque` pode ser passado um argumento `boolean` que indica se o componente é opaco (`true`) ou transparente (`false`).

A linha 41 invoca a versão de superclasse de `paintComponent` para limpar o fundo de `PaintPanel` (`Jpanel`s são opacos por padrão). As linhas 44–45 desenham uma oval na localização especificada por cada `Point` no array (até o `pointCount`). O método `Graphics` **`fillOval`** desenha uma oval sólida. Os quatro parâmetros do método representam uma área retangular (chamada de quadro delimitador) em que a oval é exibida. Os dois primeiros parâmetros são a coordenada *x* superior esquerda e a coordenada *y* superior esquerda da área retangular. As duas últimas coordenadas representam a largura e altura da área retangular. O método `fillOval` desenha a oval de tal modo que ela toque no meio de cada lado da área retangular. Na linha 45, os dois primeiros argumentos são especificados utilizando as duas variáveis de instância `public` da classe `Point` — `x` e `y`. O loop termina quando os pontos `pointCount` tiverem sido exibidos. Você aprenderá mais sobre os recursos `Graphics` no Capítulo 15.

Observações sobre a aparência e comportamento 14.14

O desenho em qualquer componente GUI é realizado com as coordenadas que são medidas a partir do canto superior esquerdo (0, 0) desse componente GUI, não o canto superior esquerdo da tela.

Utilizando o Jpanel personalizado em um aplicativo

A classe `Painter` (Figura 14.35) contém o método `main` que executa esse aplicativo. A linha 14 cria um objeto `PaintPanel` em que o usuário pode arrastar o mouse para desenhar. A linha 15 anexa o `PaintPanel` ao `Jframe`.

```
1  // Figura 14.35: Painter.java
2  // Testando o PaintPanel.
3  import ons.awt.BorderLayout;
4  import javax.swing.Jframe;
5  import javax.swing.Jlabel;
6
7  public class Painter
8  {
9     public static void main( String[] args )
10    {
11       // cria o Jframe
12       Jframe application = new Jframe( "A simple paint program" );
13
14       PaintPanel paintPanel = new PaintPanel(); // cria o painel de pintura
15       application.add( paintPanel, BorderLayout.CENTER ); // no centro
16
17       // cria um rótulo e o coloca em SOUTH do BorderLayout
18       application.add( new Jlabel( "Drag the mouse to draw" ),
19          BorderLayout.SOUTH );
20
21       application.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
22       application.setSize( 400, 200 ); // configura o tamanho do frame
23       application.setVisible( true ); // exibe o frame
24    } // fim de main
25 } // fim da classe Painter
```

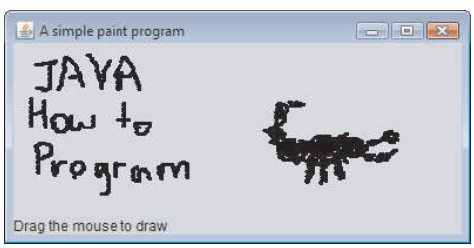


Figura 14.35 | Classe de teste para `PaintFrame`.

14.17 Tratamento de evento-chave

Esta seção apresenta a interface `KeyListener` para tratar **eventos de teclado**. Eventos de teclado são gerados quando as teclas no teclado são pressionadas e liberadas. Uma classe que implementa `KeyListener` deve fornecer declarações para métodos **`keyPressed`**, **`keyReleased`** e **`keyTyped`**, cada um dos quais recebe um `KeyEvent` como seu argumento. A classe `KeyEvent` é uma subclasse de `InputEvent`. O método `keyPressed` é chamado em resposta ao pressionamento de qualquer tecla. O método `keyTyped` é chamado em resposta ao pressionamento de qualquer tecla que não seja uma **tecla de ação**. (As teclas de ação são qualquer tecla de seta, *Home*, *End*, *Page Up*, *Page Down*, qualquer tecla de função etc.) O método `keyReleased` é chamado quando a tecla é lançada depois de qualquer evento `keyPressed` ou `keyTyped`.

O aplicativo das figuras 14.36–14.37 demonstra os métodos `KeyListener`. A classe `KeyDemoFrame` implementa a interface `KeyListener`, então todos os três métodos são definidos no aplicativo.

```
1  // Figura 14.36: KeyDemoFrame.java
2  // Demonstrando os eventos de pressionamento de tecla.
3  import ons.awt.Color;
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.Jframe;
7  import javax.swing.JtextArea;
8
9  public class KeyDemoFrame extends Jframe implements KeyListener
10 {
11    private String line1 = ""; // primeira linha de textarea
12    private String line2 = ""; // segunda linha de textarea
13    private String line3 = ""; // terceira linha de textarea
14    private JtextArea textArea; // textarea a exibir saída
15
16    // construtor KeyDemoFrame
17    public KeyDemoFrame()
18    {
19       super( "Demonstrating Keystroke Events" );
20
21       textArea = new JtextArea( 10, 15 ); // configura JtextArea
22       textArea.setText( "Press any key on the keyboard..." );
23       textArea.setEnabled( false ); // desativa textarea
24       textArea.setDisabledTextColor( Color.BLACK ); // configura a cor do texto
25       add( textArea ); // adiciona textarea ao Jframe
26
27       addKeyListener( this ); // permite que o frame processe os eventos de teclado
28    } // fim do construtor KeyDemoFrame
29
30    // trata pressionamento de qualquer tecla
31    public void keyPressed( KeyEvent event )
32    {
33       line1 = String.format( "Key pressed: %s",
34          KeyEvent.getKeyText( event.getKeyCode() ) ); // mostra a tecla pressionada
35       setLines2and3( event ); // configura a saída das linhas dois e três
36    } // fim do método keyPressed
37
38    // trata liberação de qualquer tecla
39    public void keyReleased( KeyEvent event )
40    {
41       line1 = String.format( "Key released: %s",
42          KeyEvent.getKeyText( event.getKeyCode() ) ); // mostra a tecla liberada
43       setLines2and3( event ); // configura a saída das linhas dois e três
44    } // fim do método keyReleased
45
46    // trata pressionamento de uma tecla de ação
47    public void keyTyped( KeyEvent event )
48    {
49       line1 = String.format( "Key typed: %s", event.getKeyChar() );
50       setLines2and3( event ); // configura a saída das linhas dois e três
51    } // fim do método keyTyped
52
53    // configura segunda e terceira linhas de saída
54    private void setLines2and3( KeyEvent event )
```

```
55      {
56         line2 = String.format( "This key is %san action key",
57            (event.isActionKey() ? "" : "not " ) );
58
59         String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
60
61         line3 = String.format( "Modifier keys pressed: %s",
62            ( temp.equals( "" ) ? "none" : temp ) ); // modificadores de saída
63
64         textArea.setText( String.format( "%s\n%s\n%s\n",
65            line1, line2, line3 ) ); // gera saída de três linhas de texto
66      } // fim do método setLines2and3
67   } // fim da classe KeyDemoFrame
```

Figura 14.36 | Tratamento de evento de teclado.

```
1   // Figura 14.37: KeyDemo.java
2   // Testando KeyDemoFrame.
3   import javax.swing.Jframe;
4
5   public class KeyDemo
6   {
7      public static void main( String[] args )
8      {
9         KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
10        keyDemoFrame.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
11        keyDemoFrame.setSize( 350, 100 ); // configura o tamanho do frame
12        keyDemoFrame.setVisible( true ); // exibe o frame
13     } // fim de main
14  } // fim da classe KeyDemo
```

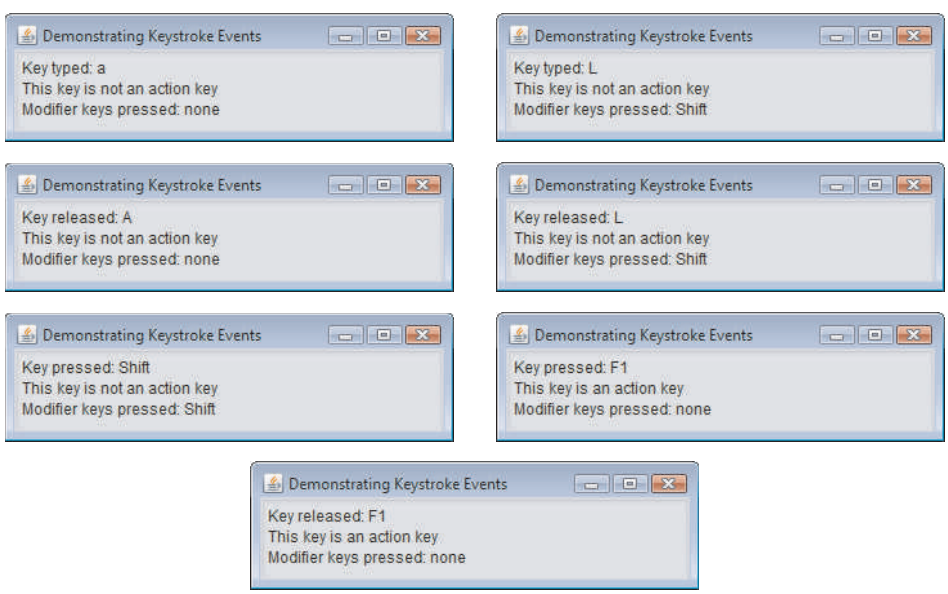


Figura 14.37 | Classe de teste para `KeyDemoFrame`.

O construtor (Figura 14.36, linhas 17–28) registra o aplicativo para tratar seus próprios eventos de teclado utilizando o método **`addKeyListener`** na linha 27. O método `addKeyListener` é declarado na classe `Component`, então cada subclasse de `Component` pode notificar objetos `KeyListener` de eventos de teclado desse `Component`.

Na linha 25, o construtor adiciona `JtextArea` `textArea` (onde a saída do aplicativo é exibida) ao `Jframe`. Uma `JtextArea` é uma área com várias linhas em que você pode exibir o texto. (Discutimos `Jtext`Areas mais detalhadamente na Seção 14.20.) Note nas capturas de tela que `textArea` ocupa a janela inteira. Isso é devido ao `BorderLayout` padrão de `Jframe` (discutido na Seção 14.18.2 e demonstrado na Figura 14.41). Quando um único `Component` é adicionado a um `BorderLayout`, o `Component` ocupa o `Container` inteiro. A linha 23

desativa `JtextArea`, portanto, o usuário não pode digitar nela. Isso faz com que o texto na `JtextArea` torne-se cinza. A linha 24 utiliza o método **`setDisabledTextColor`** para mudar a cor do texto na `JtextArea` para preto a fim de melhorar a legibilidade.

Os métodos `keyPressed` (linhas 31–36) e `keyReleased` (linhas 39–44) utilizam o método `KeyEvent` **`getKeyCode`** para obter o **código de tecla virtual** da tecla pressionada. A classe `KeyEvent` contém constantes de código de tecla virtuais que representam cada tecla no teclado. Essas constantes podem ser comparadas com o valor de retorno de `getKeyCode` para testar as teclas individuais no teclado. O valor retornado por `getKeyCode` é passado para o método `static` `KeyEvent` **`getKeyText`**, que retorna uma string contendo o nome da tecla que foi pressionada. Para uma lista completa de constantes de tecla virtual, veja a documentação on-line para a classe `KeyEvent` (pacote `java.awt.event`). O método `keyTyped` (linhas 47–51) utiliza o método `KeyEvent` **`getKeyChar`** (que retorna um `char`) para obter o valor Unicode do caractere digitado.

Todos os três métodos de tratamento de evento terminam chamando o método `setLines2and3` (linhas 54–66) e passando para ele o objeto `KeyEvent`. Esse método utiliza o método `KeyEvent` **`isActionKey`** (linha 57) para determinar se a tecla no evento é uma tecla de ação. Além disso, o método `InputEvent` **`getModifiers`** é chamado (linha 59) para determinar se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) foi pressionada quando o evento de teclado ocorreu. O resultado desse método é passado para o método `static` `KeyEvent` **`getKeyModifiersText`**, que produz uma string contendo os nomes das teclas modificadoras pressionadas.

[*Nota:* se você precisar testar uma tecla específica no teclado, a classe `KeyEvent` fornece uma **tecla constante** para cada uma delas. Essas constantes podem ser utilizadas a partir dos handlers de evento de teclado para determinar se uma tecla particular foi pressionada. Além disso, para determinar se as teclas *Alt*, *Ctrl*, *Meta* e *Shift* são pressionadas individualmente, os métodos `InputEvent` **`isAltDown`**, **`isControlDown`**, **`isMetaDown`** e **`isShiftDown`** retornam um `boolean` indicando se a tecla particular foi pressionada durante o evento de teclado.]

14.18 Introdução a gerenciadores de layout

Os **gerenciadores de layout** organizam componentes GUI em um contêiner para propósitos de apresentação. Você pode utilizar os gerenciadores de layout para obter as capacidades de layout básicas em vez de determinar a posição e o tamanho exatos de cada componente GUI. Essa funcionalidade permite que você se concentre na aparência e comportamento básicos e deixa os gerenciadores de layout processarem a maioria dos detalhes de layout. Todos os gerenciadores de layout implementam a interface **`LayoutManager`** (no pacote `java.awt`). O método `setLayout` da classe `Container` aceita um objeto que implementa a interface `LayoutManager` como um argumento. Há basicamente três maneiras de organizar componentes em uma GUI:

1. Posicionamento absoluto: esse fornece o maior nível de controle sobre a aparência de uma GUI. Configurando o layout de um `Container` como `null`, você pode especificar a posição absoluta de cada componente GUI em relação ao canto superior esquerdo do `Container` usando métodos `Component` `setSize` e `setLocation` ou `setBounds`. Se fizer isso, você também deve especificar o tamanho de cada componente GUI. A programação de uma GUI com posicionamento absoluto pode ser tediosa a menos que você tenha um ambiente de desenvolvimento integrado (IDE) que pode gerar o código para você.
2. Gerenciadores de layout: utilizar os gerenciadores de layout para posicionar elementos pode ser mais simples e mais rápido que criar uma GUI com posicionamento absoluto, mas você perde algum controle sobre o tamanho e o posicionamento precisos de componentes GUI.
3. Programação visual em um IDE: os IDEs fornecem ferramentas que facilitam a criação de GUIs. Em geral, todo IDE fornece uma **ferramenta de design** GUI que permite arrastar e soltar componentes GUI de uma caixa de ferramenta em uma área de desenho. Você então pode posicionar, dimensionar e alinhar componentes GUI como quiser. O IDE gera o código Java que cria a GUI. Além disso, em geral, você pode adicionar o código de tratamento de evento de um componente particular dando um clique duplo no componente. Algumas ferramentas de desenho também permitem utilizar os gerenciadores de layout descritos neste capítulo e no Capítulo 25.

Observações sobre a aparência e comportamento 14.15

A maioria dos IDEs Java fornece ferramentas de projeto para projetar visualmente uma GUI; as ferramentas então escrevem o código Java que cria a GUI. Essas ferramentas costumam fornecer maior controle sobre o tamanho, posição e alinhamento de componentes GUI do que os gerenciadores de layouts integrados.

Observações sobre a aparência e comportamento 14.16

É possível configurar o layout de um `Container` como `null`, que indica que nenhum gerenciador de layout deve ser utilizado. Em um `Container` sem gerenciador de layout, você deve posicionar e dimensionar os componentes no contêiner dado e tomar o cuidado de que, em eventos de redimensionamento, todos os componentes sejam reposicionados conforme necessário. Os eventos de redimensionamento de um componente podem ser processados por um `ComponentListener`.

A Figura 14.38 resume os gerenciadores de layout apresentados neste capítulo. Outros são discutidos no Capítulo 25 e o poderoso gerenciador de layout `GroupLayout` é discutido no Apêndice I.

Gerenciador de layout	Descrição
`FlowLayout`	Padrão para `javax.swing.Jpanel`. Coloca os componentes sequencialmente (da esquerda para a direita) na ordem em que foram adicionados. Também é possível especificar a ordem dos componentes utilizando o método `Container` `add` que aceita um `Component` e uma posição de índice do tipo inteiro como argumentos.
`BorderLayout`	Padrão para `JFrames` (e outras janelas). Organiza os componentes em cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`.
`GridLayout`	Organiza os componentes nas linhas e colunas.

Figura 14.38 | Gerenciadores de layout.

14.18.1 FlowLayout

`FlowLayout` é o gerenciador de layout mais simples. Os componentes GUI são colocados em um contêiner da esquerda para a direita na ordem em que são adicionados ao contêiner. Quando a borda do contêiner for alcançada, os componentes continuarão a ser exibidos na próxima linha. A classe `FlowLayout` permite aos componentes GUI ser alinhados à esquerda, centralizados (o padrão) e alinhados à direita.

O aplicativo das figuras 14.39–14.40 cria três objetos `JButton` e os adiciona ao aplicativo, usando um gerenciador de layout `FlowLayout`. Os componentes são centralizados por padrão. Quando o usuário clica em **Left**, o alinhamento do gerenciador de layout muda para um `FlowLayout` alinhado à esquerda. Quando o usuário clica em **Right**, o alinhamento do gerenciador de layout muda para um `FlowLayout` alinhado à direita. Quando o usuário clica em **Center**, o alinhamento do gerenciador de layout muda para um `FlowLayout` alinhado ao centro. Cada botão tem seu próprio handler de evento que é declarado com uma classe interna anônima que implementa `ActionListener`. As janelas de saída de exemplo mostram cada um dos alinhamentos `FlowLayout`. Além disso, a última janela de saída de exemplo mostra o alinhamento centralizado depois que a janela foi redimensionada para uma largura menor. Observe que o botão **Right** flui em uma nova linha.

Como visto anteriormente, um layout do contêiner é configurado com o método `setLayout` da classe `Container`. A linha 25 configura o gerenciador de layout como o `FlowLayout` declarado na linha 23. Normalmente, o layout é configurado antes de qualquer componente GUI ser adicionado a um contêiner.

Observações sobre a aparência e comportamento 14.17

Cada contêiner individual pode ter apenas um gerenciador de layout, mas vários contêineres no mesmo aplicativo podem utilizar, cada um, gerenciadores de layout diferentes.

```
1  // Figura 14.39: FlowLayoutFrame.java
2  // Demonstrando os alinhamentos FlowLayout.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12    private JButton leftJButton; // botão para configurar alinhamento à esquerda
13    private JButton centerJButton; // botão para configurar alinhamento centralizado
14    private JButton rightJButton; // botão para configurar alinhamento à direita
15    private FlowLayout layout; // objeto de layout
16    private Container container; // contêiner para configurar layout
17
18    // configura GUI e registra listeners de botão
19    public FlowLayoutFrame()
20    {
21       super( "FlowLayout Demo" );
22
23       layout = new FlowLayout(); // cria FlowLayout
24       container = getContentPane(); // obtém contêiner para layout
```

```
25          setLayout( layout ); // configura o layout de frame
26
27          // configura leftJButton e registra listener
28          leftJButton = new JButton( "Left" ); // cria botão Left
29          add( leftJButton ); // adiciona o botão Left ao frame
30          leftJButton.addActionListener(
31
32             new ActionListener() // classe interna anônima
33             {
34                // processa o evento leftJButton
35                public void actionPerformed( ActionEvent event )
36                {
37                   layout.setAlignment( FlowLayout.LEFT );
38
39                   // realinha os componentes anexados
40                   layout.layoutContainer( container );
41                } // fim do método actionPerformed
42             } // fim da classe interna anônima
43          ); // fim da chamada para addActionListener
44
45          // configura centerJButton e registra o listener
46          centerJButton = new JButton( "Center" ); // cria botão Center
47          add( centerJButton ); // adiciona botão Center ao frame
48          centerJButton.addActionListener(
49
50             new ActionListener() // classe interna anônima
51             {
52                // processa evento centerJButton
53                public void actionPerformed( ActionEvent event )
54                {
55                   layout.setAlignment( FlowLayout.CENTER );
56
57                   // realinha os componentes anexados
58                   layout.layoutContainer( container );
59                } // fim do método actionPerformed
60             } // fim da classe interna anônima
61          ); // fim da chamada para addActionListener
62
63          // configura rightJButton e registra o listener
64          rightJButton = new JButton( "Right" ); // cria botão Right
65          add( rightJButton ); // adiciona botão Right ao frame
66          rightJButton.addActionListener(
67
68             new ActionListener() // classe interna anônima
69             {
70                // processo evento rightJButton
71                public void actionPerformed( ActionEvent event )
72                {
73                   layout.setAlignment( FlowLayout.RIGHT );
74
75                   // realinha os componentes anexados
76                   layout.layoutContainer( container );
77                } // fim do método actionPerformed
78             } // fim da classe interna anônima
79          ); // fim da chamada para addActionListener
80       } // fim do construtor FlowLayoutFrame
81    } // FlowLayoutFrame fim da classe
```

Figura 14.39 | `FlowLayout` permite que os componentes fluam sobre múltiplas linhas.

```
1    // Figura 14.40: FlowLayoutDemo.java
2    // Testando FlowLayoutFrame.
3    import javax.swing.JFrame;
4
5    public class FlowLayoutDemo
6    {
7       public static void main( String[] args )
8       {
9          FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
```

```
10          flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11          flowLayoutFrame.setSize( 300, 75 ); // configura o tamanho do frame
12          flowLayoutFrame.setVisible( true ); // exibe o frame
13       } // fim de main
14    } // fim da classe FlowLayoutDemo
```

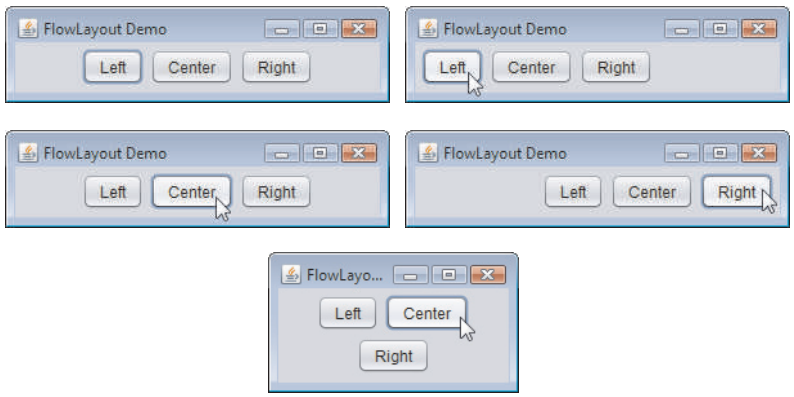


Figura 14.40 | Classe de teste para `FlowLayoutFrame`.

Observe que a rotina de tratamento de evento de cada botão é especificada com um objeto de classe interna anônima separado (Figura 14.39, linhas 30–43, 48–61 e 66–79, respectivamente) e o método `actionPerformed` executa duas instruções em cada caso. Por exemplo, a linha 37 no handler de eventos para `leftJButton` utiliza o método `FlowLayout` **`setAlignment`** para mudar o alinhamento do `FlowLayout` para um `FlowLayout` alinhado à esquerda (**`FlowLayout.LEFT`**). A linha 40 utiliza o método **`layoutContainer`** (que é herdado por todos os gerenciadores de layout) da interface `LayoutManager` para especificar que o `JFrame` deve ser reorganizado com base no layout ajustado. Dependendo do botão clicado, o método `actionPerformed` de cada botão configura o alinhamento de `FlowLayout` como `FlowLayout.LEFT` (linha 37), **`FlowLayout.CENTER`** (linha 55) ou **`FlowLayout.RIGHT`** (linha 73).

14.18.2 BorderLayout

O gerenciador de layout `BorderLayout` (o gerenciador de layout padrão de um `JFrame`) organiza componentes em cinco regiões: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`. `NORTH` corresponde à parte superior do contêiner. A classe `BorderLayout` estende `Object` e implementa a interface **`LayoutManager2`** (uma subinterface de `LayoutManager` que adiciona vários métodos para obter um processamento de layout aprimorado).

Um `BorderLayout` limita um `Container` a conter no máximo cinco componentes — um em cada região. O componente colocado em cada região pode ser um contêiner ao qual os outros componentes são anexados. Os componentes colocados nas regiões `NORTH` e `SOUTH` estendem-se horizontalmente para os lados do contêiner e têm a mesma altura que o componente mais alto colocado nessas regiões. As regiões `EAST` e `WEST` se expandem verticalmente entre `NORTH` e `SOUTH` e são tão largas quanto os componentes colocados nessas regiões. O componente colocado na região `CENTER` se expande para preencher todo o espaço restante no layout (que é a razão de `JTextArea` na Figura 14.37 ocupar a janela inteira). Se todas as cinco regiões são ocupadas, o espaço do contêiner inteiro é coberto por componentes GUI. Se a região `NORTH` ou `SOUTH` não for ocupada, os componentes GUI nas regiões `EAST`, `CENTER` e `WEST` se expandem verticalmente para preencher o espaço restante. Se a região `EAST` ou `WEST` não for ocupada, o componente GUI na região `CENTER` expande horizontalmente para preencher o espaço restante. Se a região `CENTER` não for ocupada, a área é deixada vazia — os outros componentes GUI não se expandem para preencher o espaço restante. O aplicativo das figuras 14.41–14.42 demonstra o gerenciador de layout `BorderLayout` utilizando cinco `JButtons`.

```
1  // Figura 14.41: BorderLayoutFrame.java
2  // Demonstrando BorderLayout.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11    private JButton[] buttons; // array de botões para ocultar partes
12    private static final String[] names = { "Hide North", "Hide South",
13       "Hide East", "Hide West", "Hide Center" };
14    private BorderLayout layout; // objeto borderlayout
```

```
15
16       // configura GUI e tratamento de evento
17       public BorderLayoutFrame()
18       {
19          super( "BorderLayout Demo" );
20
21          layout = new BorderLayout( 5, 5 ); // espaços de 5 pixels
22          setLayout( layout ); // configura o layout de frame
23          buttons = new JButton[ names.length ]; // configura o tamanho do array
24
25          // cria JButtons e registra ouvintes para eles
26          for ( int count = 0; count < names.length; count++ )
27          {
28             buttons[ count ] = new JButton( names[ count ] );
29             buttons[ count ].addActionListener( this );
30          } // for final
31
32          add( buttons[ 0 ], BorderLayout.NORTH ); // adiciona botão para o norte
33          add( buttons[ 1 ], BorderLayout.SOUTH ); // adiciona botão para o sul
34          add( buttons[ 2 ], BorderLayout.EAST ); // adiciona botão para o leste
35          add( buttons[ 3 ], BorderLayout.WEST ); // adiciona botão para o oeste
36          add( buttons[ 4 ], BorderLayout.CENTER ); // adiciona botão para o centro
37       } // fim do construtor BorderLayoutFrame
38
39       // trata os eventos de botão
40       public void actionPerformed( ActionEvent event )
41       {
42          // verifica a origem de evento e o painel de conteúdo de layout correspondentemente
43          for ( JButton button : buttons )
44          {
45             if ( event.getSource() == button )
46                button.setVisible( false ); // oculta o botão clicado
47             else
48                button.setVisible( true ); // mostra outros botões
49          } // for final
50
51          layout.layoutContainer( getContentPane() ); // faz o layout do painel de conteúdo
52       } // fim do método actionPerformed
53    } // fim da classe BorderLayoutFrame
```

Figura 14.41 | `BorderLayout` que contém cinco botões.

```
1  // Figura 14.42: BorderLayoutDemo.java
2  // Testando BorderLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class BorderLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10       borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       borderLayoutFrame.setSize( 300, 200 ); // configura o tamanho do frame
12       borderLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe BorderLayoutDemo
```

Figura 14.42 | Teste a classe `BorderLayoutFrame`.

A linha 21 da Figura 14.41 cria um `BorderLayout`. Os argumentos de construtor especificam o número de pixels entre componentes que estão organizados horizontalmente (**espaçamento horizontal**) e entre componentes que são organizados verticalmente (**espaçamento vertical**), respectivamente. O padrão tem horizontal e verticalmente um pixel de espaçamento. A linha 22 utiliza o método `setLayout` para configurar o layout do painel de conteúdo como `layout`.

Adicionamos `Components` a um `BorderLayout` com outra versão do método `Container` `add` que aceita dois argumentos — o `Component` para adicionar e a região em que o `Component` deva aparecer. Por exemplo, a linha 32 especifica que `buttons[ 0 ]` deve aparecer na região `NORTH`. Os componentes podem ser adicionados em qualquer ordem, mas apenas um componente deve ser adicionado a cada região.

Observações sobre a aparência e comportamento 14.18

Se nenhuma região for especificada ao se adicionar um `Component` a um `BorderLayout`, o gerenciador de layout assume que o `Component` deve ser adicionado à região `BorderLayout.CENTER`.

Erro comum de programação 14.6

Quando mais de um componente for adicionado a uma região em um `BorderLayout`, somente o último componente adicionado a essa região será exibido. Não há nenhum erro que indica esse problema.

Observe que a classe `BorderLayoutFrame` implementa `ActionListener` diretamente nesse exemplo, então `BorderLayoutFrame` tratará os eventos de `JButtons`. Por essa razão, a linha 29 passa a referência `this` para o método `addActionListener` de cada `Jbutton`. Quando o usuário clica em um `JButton` particular no layout, o método `actionPerformed` (linhas 40–52) é executado. A instrução `for` aprimorada nas linhas 43–49 utiliza um `if...else` para ocultar o `JButton` particular que gerou o evento. O método **`setVisible`** (herdado em `JButton` da classe `Component`) é chamado com um argumento `false` (linha 46) para ocultar `JButton`. Se o `JButton` atual no array não é o que gerou o evento, o método `setVisible` é chamado com um argumento `true` (linha 48) para assegurar que o `JButton` é exibido na tela. A linha 51 utiliza o método `LayoutManager` `layoutContainer` para recalcular o layout do painel de conteúdo. Note nas capturas de tela da Figura 14.42 que certas regiões no `BorderLayout` alteram a forma quando os `JButtons` são ocultados e exibidos em

outras regiões. Tente redimensionar a janela do aplicativo para ver como as várias regiões redimensionam com base na largura e altura da janela. Para layouts mais complexos, agrupe componentes em `JPanel`s, cada um com gerenciador de layout separado. Coloque os `JPanel`s no `JFrame` utilizando o padrão `BorderLayout` ou algum outro layout.

14.18.3 GridLayout

O gerenciador de layout **`GridLayout`** divide o contêiner em uma grade de modo que os componentes podem ser colocados nas linhas e colunas. A classe `GridLayout` estende diretamente a classe `Object` e implementa a interface `LayoutManager`. Cada `Component` em um `GridLayout` tem a mesma largura e altura. Os componentes são adicionados a um `GridLayout` iniciando a célula na parte superior esquerda da grade e prosseguindo da esquerda para a direita até a linha estar cheia. Então o processo continua da esquerda para a direita na próxima linha da grade e assim por diante. O aplicativo das figuras 14.43 e 14.44 demonstra o gerenciador de layout `GridLayout` utilizando seis `JButton`s.

```
1  // Figura 14.43: GridLayoutFrame.java
2  // Demonstrando GridLayout.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12    private JButton[] buttons; // array de botões
13    private static final String[] names =
14       { "one", "two", "three", "four", "five", "six" };
15    private boolean toggle = true; // alterna entre dois layouts
16    private Container container; // contêiner do frame
17    private GridLayout gridLayout1; // primeiro gridlayout
18    private GridLayout gridLayout2; // segundo gridlayout
19
20    // construtor sem argumentos
21    public GridLayoutFrame()
22    {
23       super( "GridLayout Demo" );
24       gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 por 3; lacunas de 5
25       gridLayout2 = new GridLayout( 3, 2 ); // 3 por 2; nenhuma lacuna
26       container = getContentPane(); // obtém painel de conteúdo
27       setLayout( gridLayout1 ); // configura o layout JFrame
28       buttons = new JButton[ names.length ]; // cria array de JButtons
29
30       for ( int count = 0; count < names.length; count++ )
31       {
32          buttons[ count ] = new JButton( names[ count ] );
33          buttons[ count ].addActionListener( this ); // ouvinte registrado
34          add( buttons[ count ] ); // adiciona o botão ao JFrame
35       } // for final
36    } // fim do construtor GridLayoutFrame
37
38    // trata eventos de botão alternando entre layouts
39    public void actionPerformed( ActionEvent event )
40    {
41       if ( toggle )
42          container.setLayout( gridLayout2 ); // configura layout como segundo
43       else
44          container.setLayout( gridLayout1 ); // configura layout como primeiro
45
46       toggle = !toggle; // alterna para valor oposto
47       container.validate(); // refaz o layout do contêiner
48    } // fim do método actionPerformed
49 } // fim da classe GridLayoutFrame
```

Figura 14.43 | `GridLayout` que contém seis botões.

As linhas 24–25 criam dois objetos `GridLayout`. O construtor `GridLayout` utilizado na linha 24 especifica um `GridLayout` com 2 linhas, 3 colunas, 5 pixels de espaçamento horizontal entre os `Components` na grade e 5 pixels de espaçamento vertical entre `Components` na grade. O construtor `GridLayout` utilizado na linha 25 especifica um `GridLayout` com 3 linhas e 2 colunas que utiliza o espaçamento padrão (1 pixel).

Os objetos `JButton` nesse exemplo são inicialmente organizados utilizando-se `gridLayout1` (configura para o painel de conteúdo na linha 27 com o método `setLayout`). O primeiro componente é adicionado à primeira coluna da primeira linha. O próximo componente é adicionado à segunda coluna da primeira linha e assim por diante. Quando um `JButton` é pressionado, o método `actionPerformed` (linhas 39–48) é chamado. Toda chamada para `actionPerformed` alterna o layout entre `gridLayout2` e `gridLayout1`, utilizando a variável `boolean toggle` para determinar o próximo layout a ser configurado.

A linha 47 mostra outra maneira de reformatar um contêiner cujo layout foi alterado. O método `Container` **`validate`** recalcula o layout do contêiner com base no gerenciador de layout atual para o `Container` e o conjunto atual de componentes GUI exibidos.

```
1  // Figura 14.44: GridLayoutDemo.java
2  // Testando GridLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class GridLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10       gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       gridLayoutFrame.setSize( 300, 200 ); // configura o tamanho do frame
12       gridLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe GridLayoutDemo
```

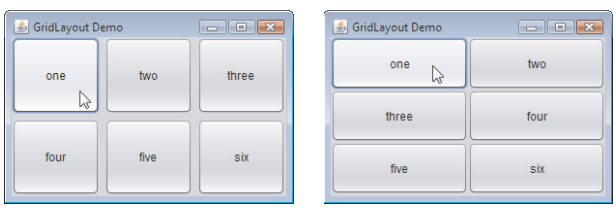


Figura 14.44 | Classe de teste para `GridLayoutFrame`.

14.19 Utilizando painéis para gerenciar layouts mais complexos

GUIs complexas (como a Figura 14.1) exigem que cada componente seja colocado em uma localização exata. Elas frequentemente consistem em múltiplos painéis, com os componentes de cada painel organizados em um layout específico. A classe `JPanel` estende `JComponent` e `JComponent` estende a classe `Container`, então todo `JPanel` é um `Container`. Assim, todo `JPanel` pode ter componentes, incluindo outros painéis, anexados, com o método `Container` `add`. O aplicativo das figuras 14.45–14.46 demonstra como um `JPanel` pode ser utilizado para criar um layout mais complexo em que vários `JButtons` são colocados na região `SOUTH` de um `BorderLayout`.

Depois de o `JPanel` `buttonJPanel` ser declarado (linha 11) e criado (linha 19), a linha 20 configura o layout de `buttonJPanel` como um `GridLayout` de uma linha e cinco colunas (há cinco `JButtons` no array `buttons`). As linhas 23–27 adicionam `JButtons` no array ao `JPanel`. A linha 26 adiciona os botões diretamente ao `JPanel` — a classe `JPanel` não tem um painel de conteúdo, ao contrário de um `JFrame`. A linha 29 utiliza o padrão `BorderLayout` de `JFrame` para adicionar `buttonJPanel` à região `SOUTH`. Observe que a região `SOUTH` é tão alta quanto os botões no `buttonJPanel`. Um `JPanel` é dimensionado aos componentes que ele contém. À medida que mais componentes são adicionados, o `JPanel` cresce (de acordo com as restrições de seu gerenciador de layout) para acomodar os componentes. Redimensione a janela para ver como o gerenciador de layout afeta o tamanho dos `JButtons`.

```
1  // Figura 14.45: PanelFrame.java
2  // Utilizando um JPanel para ajudar a fazer o layout dos componentes.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
```

```
9  public class PanelFrame extends JFrame
10 {
11    private JPanel buttonJPanel; // painel para armazenar botões
12    private JButton[] buttons; // array de botões
13
14    // construtor sem argumentos
15    public PanelFrame()
16    {
17       super( "Panel Demo" );
18       buttons = new JButton[ 5 ]; // cria botões de array
19       buttonJPanel = new JPanel(); // configura painel
20       buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );
21
22       // cria e adiciona botões
23       for ( int count = 0; count < buttons.length; count++ )
24       {
25          buttons[ count ] = new JButton( "Button " + ( count + 1 ) );
26          buttonJPanel.add( buttons[ count ] ); // adiciona botão ao painel
27       } // for final
28
29       add( buttonJPanel, BorderLayout.SOUTH ); // adiciona painel ao JFrame
30    } // fim do construtor PanelFrame
31 } // fim da classe PanelFrame
```

Figura 14.45 | `JPanel` com cinco `JButtons` anexados à região SOUTH de um `BorderLayout`.

```
1  // Figura 14.46: PanelDemo.java
2  // Testando PanelFrame.
3  import javax.swing.JFrame;
4
5  public class PanelDemo extends JFrame
6  {
7     public static void main( String[] args )
8     {
9        PanelFrame panelFrame = new PanelFrame();
10       panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       panelFrame.setSize( 450, 200 ); // configura o tamanho do frame
12       panelFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe PanelDemo
```

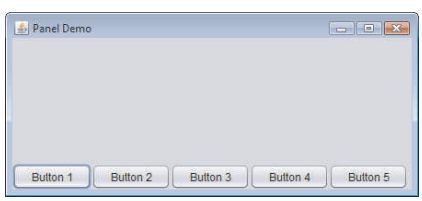


Figura 14.46 | Classe de teste para `PanelFrame`.

14.20 JTextArea

`JTextArea` fornece uma área para manipular múltiplas linhas de texto. Como a classe `JTextField`, `JTextArea` é uma subclasse de `JTextComponent`, que declara métodos comuns aos `JTextFields`, `JTextAreas` e vários outros componentes GUI baseados em texto.

O aplicativo nas figuras 14.47–14.48 demonstra as `JTextAreas`. Uma `JTextArea` exibe texto que o usuário pode selecionar. A outra é não editável e é usada para exibir o texto que o usuário selecionou na primeira `JTextArea`. Diferentemente de `JTextFields`, `JTextAreas` não têm eventos de ação — quando você pressiona *Enter* ao digitar uma `JTextArea`, o cursor simplesmente se move para a linha seguinte. Como com `JLists` de seleção múltipla (Seção 14.13), um evento externo de outro componente GUI indica quando processar o texto em uma `JTextArea`. Por exemplo, ao digitar uma mensagem de correio eletrônico, você normalmente clica em um botão **Send** para enviar o texto da mensagem para o destinatário. De maneira semelhante, ao editar um documento em um processador de texto, você normalmente salva o arquivo selecionando o item **Save** ou **Save As...** de um menu. Nesse programa, o botão **Copy >>>** gera o evento externo que copia o texto selecionado na `JTextArea` à esquerda e o exibe na `JtextArea` à direita.

```
1  // Figura 14.47: TextAreaFrame.java
2  // Copiando texto selecionado de uma textarea para outra.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13    private JTextArea textArea1; // exibe a string demo
14    private JTextArea textArea2; // texto destacado é copiado aqui
15    private JButton copyJButton; // começa a copiar o texto
16
17    // construtor sem argumentos
18    public TextAreaFrame()
19    {
20       super( "TextArea Demo" );
21       Box box = Box.createHorizontalBox(); // cria box
22       String demo = "This is a demo string to\n" +
23          "illustrate copying text\nfrom one textarea to \n" +
24          "another textarea using an\nexternal event\n";
25
26       textArea1 = new JTextArea( demo, 10, 15 ); // cria textarea1
27       box.add( new JScrollPane( textArea1 ) ); // adiciona scrollpane
28
29       copyJButton = new JButton( "Copy >>>" ); // cria botão de cópia
30       box.add( copyJButton ); // adiciona o botão de cópia à box
31       copyJButton.addActionListener(
32
33          new ActionListener() // classe interna anônima
34          {
35             // configura texto em textArea2 como texto selecionado de textArea1
36             public void actionPerformed( ActionEvent event )
37             {
38                textArea2.setText( textArea1.getSelectedText() );
39             } // fim do método actionPerformed
40          } // fim da classe interna anônima
41       ); // fim da chamada para addActionListener
42
43       textArea2 = new JTextArea( 10, 15 ); // cria segunda textarea
44       textArea2.setEditable( false ); // desativa a edição
45       box.add( new JScrollPane( textArea2 ) ); // adiciona scrollpane
46
47       add( box ); // adiciona box ao frame
48    } // fim do construtor TextAreaFrame
49 } // fim da classe TextAreaFrame
```

Figura 14.47 | Copiando texto selecionado de uma `JTextArea` para outra.

```
1  // Figura 14.48: TextAreaDemo.java
2  // Copiando texto selecionado de uma textarea para outra.
3  import javax.swing.JFrame;
4
5  public class TextAreaDemo
6  {
7     public static void main( String[] args )
8     {
9        TextAreaFrame textAreaFrame = new TextAreaFrame();
10       textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textAreaFrame.setSize( 425, 200 ); // configura o tamanho do frame
12       textAreaFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe TextAreaDemo
```

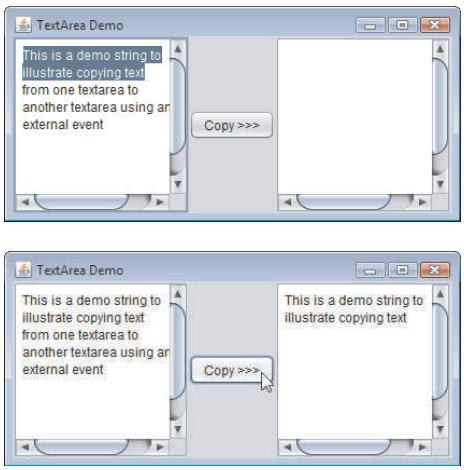


Figura 14.48 | Classe de teste para `TextAreaFrame`.

No construtor (linhas 18–48), a linha 21 cria um contêiner **`Box`** (pacote `javax.swing`) para organizar os componentes GUI. `Box` é uma subclasse de `Container` que utiliza um gerenciador de layout **`BoxLayout`** (discutido em detalhes na Seção 25.9) para organizar os componentes GUI horizontal ou verticalmente. O método `static` **`createHorizontalBox`** de `Box` cria uma `Box` que organiza componentes da esquerda para a direita na ordem que eles são anexados.

As linhas 26–43 criam as `JTextAreas` `textArea1` e `textArea2`. A linha 26 utiliza o construtor de três argumentos de `JTextArea`, que aceita uma `String` que representa o texto inicial e dois `int`s para especificar que a `JTextArea` tem `10` linhas e `15` colunas. A linha 43 utiliza o construtor de dois argumentos da `JTextArea`, especificando que a `JTextArea` tem `10` linhas e `15` colunas. A linha 26 especifica que `demo` deve ser exibido como o conteúdo `JTextArea` padrão. Uma `JTextArea` não fornece barras de rolagem se ela não puder exibir seu conteúdo completo. Desse modo, a linha 27 cria um objeto `JScrollPane`, inicializa-o com `textArea1` e o anexa ao contêiner `box`. Por padrão, as barras de rolagem horizontais e verticais aparecem conforme necessário em um `JscrollPane`.

As linhas 29–41 criam objeto `JButton` `copyJButton` com o rótulo `"Copy >>>"`, adicionam `copyJButton` ao contêiner `box` e registram o handler de evento ao `ActionEvent` de `copyJButton`. Esse botão fornece o evento externo que determina quando o programa deve copiar o texto selecionado na `textArea1` para `textArea2`. Quando o usuário clicar em `copyJButton`, a linha 38 em `actionPerformed` indica que o método **`getSelectedText`** (herdado em `JTextArea` de `JTextComponent`) deve retornar o texto selecionado de `textArea1`. O usuário seleciona texto arrastando o mouse sobre o texto desejado para destacá-lo. O método `setText` muda o texto em `textArea2` para a string retornada por `getSelectedText`.

As linhas 43–45 criam `textArea2`, configuram sua propriedade editável como `false` e adicionam essa propriedade ao contêiner `box`. A linha 47 adiciona `label3` a `JFrame`. A partir da Seção 14.18, lembre-se de que o layout padrão de um `JFrame` é um `BorderLayout` e que o método `add` por padrão anexa seu argumento ao `CENTER` do `BorderLayout`.

Quando o texto alcançar o canto direito de uma `JTextArea`, o texto pode recorrer para a próxima linha. Isso é referido como **mudança de linha automática**. Por padrão, `JTextArea` não muda de linha automaticamente.

Observações sobre a aparência e comportamento 14.19

*Para fornecer a funcionalidade de mudança de linha automática para uma `JTextArea`, invoque o método `JTextArea` **`setLineWrap`** com um argumento `true`.*

Diretivas de barra de rolagem `JScrollPane`

Esse exemplo utiliza um `JScrollPane` para fornecer rolagem para uma `JTextArea`. Por padrão, `JScrollPane` exibe apenas barras de rolagem se elas forem necessárias. As **diretivas de barra de rolagem** horizontal e vertical de um `JScrollPane` podem ser configuradas quando ele for construído. Se um programa tiver uma referência a um `JScrollPane`, o programa pode utilizar os métodos `JScrollPane` **`setHorizontalScrollBarPolicy`** e **`setVerticalScrollBarPolicy`** para alterar as diretivas de barra de rolagem a qualquer hora. A classe `JScrollPane` declara as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
```

para indicar que uma barra de rolagem sempre deve aparecer, as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

para indicar que uma barra de rolagem deve aparecer somente se necessário (os padrões) e as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que uma barra de rolagem nunca deve aparecer. Se a diretiva de barra de rolagem horizontal for configurada como `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER`, uma `JTextArea` anexada ao `JScrollPane` mudará automaticamente de linhas.

14.21 Conclusão

Neste capítulo, você aprendeu muitos componentes GUI e a implementar o tratamento de evento. Você também aprendeu sobre as classes aninhadas, classes internas e classes internas anônimas. Você viu o relacionamento especial entre um objeto de classe interna e um objeto de sua classe de primeiro nível. Você aprendeu a utilizar diálogos de `JOptionPane` para obter entrada de texto do usuário e a exibir mensagens para o usuário. Você também aprendeu a criar aplicativos que são executados em suas próprias janelas. Discutimos a classe `JFrame` e componentes que permitem ao usuário interagir com um aplicativo. Também mostramos como exibir texto e imagens para o usuário. Você aprendeu a personalizar `Jpanel`s para criar áreas de desenho personalizadas, que serão extensamente utilizadas no próximo capítulo. Viu como organizar componentes em uma janela utilizando gerenciadores de layout e como criar GUIs mais complexas utilizando `JPanel`s para organizar componentes. Por fim, você aprendeu sobre o componente `JTextArea` em que um usuário pode inserir texto e um aplicativo pode exibir texto. No Capítulo 25, você aprenderá sobre os componentes GUI mais avançados, como controles deslizantes, menus e gerenciadores de layout mais complexos. No próximo capítulo, você aprenderá a adicionar imagens gráficas ao aplicativo GUI. Os recursos gráficos permitem desenhar formas e texto com cores e estilos.

Resumo

Seção 14.1 Introdução

- Uma interface gráfica com usuário (Graphical User Interface — GUI) apresenta um mecanismo amigável ao usuário para interagir com um aplicativo. Uma GUI fornece a um aplicativo uma "aparência" e "comportamento" distintos.
- Fornecer diferentes aplicativos com componentes de interface com usuário consistentes e intuitivos dá aos usuários uma sensação de familiaridade com um novo aplicativo, para que possam aprendê-lo mais rapidamente.
- As GUIs são construídas a partir de componentes GUI — às vezes chamados controles ou widgets.

Seção 14.2 A nova interface Nimbus do Java

- Desde a atualização 10 do Java SE 6, o Java vem com uma interface nova, elegante e compatível com várias plataformas conhecidas como Nimbus.
- Para configurar o Nimbus como o padrão de todos os aplicativos Java, você deve criar um arquivo de texto `swing.properties` na pasta `lib` tanto da pasta de instalação JDK como da pasta de instalação JRE. Insira a seguinte linha do código no arquivo:

  ```
  swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
  ```

- Para selecionar o Nimbus individualmente por aplicativo, coloque o seguinte argumento de linha de comando depois do comando `java` e antes do nome do aplicativo quando você executar o aplicativo:

  ```
  -Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
  ```

Seção 14.3 Entrada/saída baseada em GUI simples com `JOptionPane`

- A maioria dos aplicativos utiliza janelas ou caixas de diálogo (também chamadas de diálogos) para interagir com o usuário.
- A classe `JOptionPane` (pacote `javax.swing`) fornece caixas de diálogo predefinidas tanto para entrada como para saída. O método `JOptionPane` `static` `showInputDialog` exibe um diálogo de entrada.
- Em geral, um prompt utiliza maiúsculas e minúsculas no estilo de frases — empregando a maiúscula inicial apenas na primeira palavra da frase a menos que a palavra seja um nome próprio.
- Um diálogo de entrada só pode inserir `String`s de entrada. Isso é típico da maioria dos componentes GUI.
- O método `JOptionPane` `static` `showMessageDialog` exibe um diálogo de mensagem.

Seção 14.4 Visão geral de componentes Swing

- A maioria dos componentes Swing GUI está localizada no pacote `javax.swing`.

- Juntas, a aparência e a maneira como o usuário interage com o aplicativo são conhecidas como a aparência e comportamento desse aplicativo. Os componentes Swing GUI permitem especificar uniformemente a aparência e comportamento para o aplicativo em todas as plataformas ou utilizar a aparência e comportamento personalizados de cada plataforma.
- Os componentes Swing leves não são amarrados aos componentes GUI reais suportados pela plataforma subjacente em que um aplicativo é executado.
- Vários componentes Swing são componentes pesados que exigem interação direta com o sistema de janela local, que pode restringir sua aparência e funcionalidades.
- A classe `Component` (pacote `java.awt`) declara muitos dos atributos e comportamentos comuns aos componentes GUI em pacotes `java.awt` e `javax.swing`.
- A classe `Container` (pacote `java.awt`) é uma subclasse de `Component`. `Components` são anexados a `Containers`, desse modo, os `Components` podem ser organizados e exibidos na tela.
- A classe `JComponent` (pacote `javax.swing`) é uma subclasse de `Container`. `JComponent` é a superclasse de todos os componentes Swing leves e declara seus atributos e comportamentos comuns.
- Alguns recursos de `JComponent` comuns incluem uma aparência e comportamento plugáveis, teclas de atalho chamadas de mnemônicas, dicas de ferramenta, suporte para tecnologias de apoio a deficientes e suporte para localização de interface com o usuário.

Seção 14.5 Exibição de texto e imagens em uma janela

- A classe `JFrame` fornece os atributos e comportamentos básicos de uma janela.
- Um `JLabel` exibe texto somente de leitura, uma imagem ou tanto texto como imagem. Normalmente, o texto em um `JLabel` emprega maiúsculas e minúsculas no estilo de frases.
- Cada componente GUI deve ser anexado a um contêiner, como uma janela criada com um `JFrame`.
- Muitos IDEs fornecem ferramentas de design de GUI em que você pode especificar o tamanho e localização exata de um componente utilizando o mouse; então, o IDE gerará o código GUI para você.
- O método `JComponent` `setToolTipText` especifica a dica de ferramenta que é exibida quando o usuário posiciona o cursor do mouse sobre um componente leve.
- O método `Container` `add` anexa um componente GUI a um `Container`.
- A classe `ImageIcon` suporta vários formatos de imagem, incluindo GIF, PNG e JPEG.
- O método `getClass` (da classe `Object`) recupera uma referência ao objeto `Class` que representa a declaração de classe do objeto em que o método é chamado.
- O método `Class` `getResource` retorna a localização de seu argumento como um URL. O método `getResource` utiliza o carregador de classe do objeto `Class` para determinar a localização do recurso.
- A interface `SwingConstants` (pacote `javax.swing`) declara constantes de inteiro comuns que são utilizadas com muitos componentes Swing.
- Os alinhamentos horizontais e verticais de um `JLabel` podem ser configurados com os métodos `setHorizontalAlignment` e `setVerticalAlignment`, respectivamente.
- O método `JLabel` `setText` configura o texto exibido em um rótulo. O método correspondente `getText` recupera o texto atual exibido em um rótulo.
- O método `JLabel` `setIcon` especifica o `Icon` a ser exibido em um rótulo. Um método correspondente `getIcon` recupera o `Icon` atual exibido em um rótulo.
- Os métodos `JLabel` `setHorizontalTextPosition` e `setVerticalTextPosition` especificam a posição de texto no rótulo.
- O método `JFrame` `setDefaultCloseOperation` com a constante `JFrame.EXIT_ON_CLOSE` como o argumento indica o que o programa deve terminar quando a janela é fechada pelo usuário.
- O método `Component` `setSize` especifica a largura e altura de um componente.
- O método `Component` `setVisible` com o argumento `true` exibe um `JFrame` na tela.

Seção 14.6 Campos de texto e uma introdução ao tratamento de evento com classes aninhadas

- As GUIs são baseadas em evento — quando o usuário interage com um componente GUI, os eventos guiam o programa para realizar tarefas.
- O código que realiza uma tarefa em resposta a um evento é chamado handler de evento e o processo total de responder a eventos é conhecido como tratamento de evento.
- A classe `JTextField` estende a classe `JTextComponent` (pacote `javax.swing.text`), que fornece recursos de componentes baseados em texto comuns. A classe `JPasswordField` estende `JTextField` e adiciona vários métodos que são específicos ao processamento de senhas.
- Um `JPasswordField` mostra que os caracteres estão sendo digitados à medida que o usuário os insere, mas oculta os caracteres reais com caracteres de eco.
- Um componente recebe o foco quando o usuário clica no componente.
- O método `JTextComponent` `setEditable` pode ser utilizado para tornar um campo de texto não editável.

- Para responder a um evento de determinado componente GUI, você deve: 1) Criar uma classe que representa o handler de evento. 2) Implementar uma interface apropriada de ouvinte de evento, na classe do *Passo 1*. 3) Indicar que um objeto da classe dos *Passos 1* e *2* deve ser notificado quando ocorrer o evento. Isso é conhecido como registrar handler de evento.
- As classes aninhadas podem ser `static` ou não `static`. As classes não `static` aninhadas são chamadas de classes internas e são frequentemente utilizadas para tratamento de evento.
- Um objeto de uma classe interna não `static` deve ser criado por um objeto da classe de nível superior que contém a classe interna.
- Um objeto de classe interna pode acessar diretamente as variáveis de instância e métodos de sua classe de primeiro nível.
- Uma classe aninhada que é `static` não exige um objeto de sua classe de primeiro nível e não tem implicitamente uma referência a um objeto da classe de primeiro nível.
- Pressionar *Enter* em um `JTextField` ou `JPasswordField` gera um `ActionEvent` (do pacote `java.awt.event`) que pode ser tratado por um `ActionListener` (pacote `java.awt.event`).
- O método `JTextField addActionListener` registra a rotina de tratamento de evento para esse `ActionEvent` de um campo de texto. Esse método recebe como seu argumento um objeto `ActionListener`.
- O componente GUI com o qual o usuário interage é a origem de evento.
- Um objeto `ActionEvent` contém informações sobre o evento que acabou de ocorrer, como a origem de evento e o texto no campo de texto.
- O método `ActionEvent getSource` retorna uma referência à origem de evento. O método `ActionEvent getActionCommand` retorna o texto que o usuário digitou em um campo de texto ou o rótulo em um `JButton`.
- O método `JPasswordField getPassword` retorna a senha digitada pelo usuário.

Seção 14.7 Tipos comuns de eventos GUI e interfaces ouvintes

- Todo tipo de objeto de evento costuma ter uma interface de ouvinte do evento correspondente que especifica um ou vários métodos de tratamento de evento que devem ser declarados na classe que implementa a interface.

Seção 14.8 Como o tratamento de evento funciona

- Quando um evento ocorre, o componente GUI com o qual o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte.
- Cada `JComponent` tem uma `EventListenerList` (pacote `javax.swing.event`) na qual as referências a ouvintes registrados são armazenadas.
- Cada componente GUI suporta vários tipos de evento. Quando um evento ocorre, o evento é despachado apenas para os ouvintes de evento do tipo apropriado. O componente GUI recebe um evento ID único que especifica o tipo de evento, que ele utiliza para decidir o tipo de listener para o qual o evento deve ser despachado e qual método chamar em cada objeto listener.

Seção 14.9 `JButton`

- Um botão é um componente em que o usuário clica para acionar uma ação. Todos os tipos de botão são subclasses de `AbstractButton` (pacote `javax.swing`). Em geral, os rótulos de botões utilizam letras maiúsculas e minúsculas no estilo de título de livro.
- Os botões de comando são criados com a classe `JButton`.
- Um `JButton` pode exibir um `Icon`. Um `JButton` também pode ter um `Icon` rollover — um `Icon` que é exibido quando o usuário posiciona o mouse sobre o botão.
- O método `setRolloverIcon` (da classe `AbstractButton`) especifica a imagem exibida em um botão quando o usuário posiciona o mouse sobre ele.

Seção 14.10 Botões que mantêm o estado

- Há três tipos de botão de estado Swing — `JToggleButton`, `JCheckBox` e `JRadioButton`.
- As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton`.
- O método `Component setFont` configura a fonte do componente como um novo objeto `Font` (pacote `java.awt`).
- Clicar um `JCheckBox` causa `ItemEvent` que pode ser tratado por um objeto `ItemListener` (que deve implementar o método `ItemStateChanged`). O método `addItemListener` registra o ouvinte para o `ItemEvent` de um `JCheckBox` ou objeto `JRadioButton`.
- O método `JCheckBox isSelected` determina se um `JCheckBox` está selecionado.
- Os `JRadioButton`s são semelhantes às `JCheckBox`es porque eles têm dois estados — selecionado e não selecionado. Entretanto, os botões de opção normalmente aparecem como um grupo em que apenas um botão pode ser selecionado por vez. Selecionar um botão de opção diferente força a remoção da seleção de todos os outros que estão selecionados.
- `JRadioButton`s são utilizados para representar opções mutuamente exclusivas.
- A relação lógica entre `JRadioButton`s é mantida por um objeto `ButtonGroup`.
- O método `ButtonGroup add` associa cada `JRadioButton` com um `ButtonGroup`. Se mais de um objeto `JRadioButton` selecionado for adicionado a um grupo, aquele selecionado que foi adicionado primeiro será selecionado quando a GUI for exibida.
- Um segundo argumento `true` indica que o `JRadioButton` deve aparecer selecionado quando exibido.

Seção 14.11 JComboBox e uso de uma classe interna anônima para tratamento de evento

- Uma `JComboBox` fornece uma lista de itens em que o usuário pode fazer uma única seleção. `JComboBox`es geram `ItemEvent`s.
- Todo item em uma `JComboBox` tem um índice. O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida. Outros itens são selecionados clicando na `JComboBox`, que se expande em uma lista a partir da qual o usuário pode fazer uma seleção.
- O método `JcomboBox setMaximumRowCount` configura o número máximo de elementos que é exibido quando o usuário clica na `JComboBox`. Se houver mais itens, a `JComboBox` fornece uma barra de rolagem que permite que o usuário role por todos os elementos na lista.
- Uma classe interna anônima é uma forma especial de classe interna que é declarada sem nome e, em geral, aparece dentro de uma declaração de método. Como uma classe interna anônima não tem nome, deve-se criar um objeto da classe interna anônima no ponto em que a classe é declarada.
- O método `JComboBox getSelectedIndex` retorna o índice do item selecionado.

Seção 14.12 JList

- Uma `JList` exibe uma série de itens da qual o usuário pode selecionar um ou mais itens. A classe `JList` suporta listas de uma única seleção e listas de seleção múltipla.
- Quando o usuário clicar em um item de uma `JList`, um `ListSelectionEvent` ocorre. O método `JList addListSelectionListener` registra um `ListSelectionListener` para eventos de seleção de uma `JList`. Um `ListSelectionListener` (pacote `javax.swing.event`) deve implementar o método `valueChanged`.
- O método `JList setVisibleRowCount` especifica o número de itens que são visíveis na lista.
- O método `JList setSelectionMode` especifica o modo de seleção de uma lista.
- Uma `JList` não fornece automaticamente uma barra de rolagem se houver mais itens na lista que o número de linhas visíveis. Nesse caso, um objeto `JScrollPane` pode ser utilizado para fornecer a capacidade de rolagem.
- O método `JFrame getContentPane` retorna uma referência ao painel de conteúdo de `JFrame` em que os componentes GUI são exibidos.
- O método `JList getSelectedIndex` retorna o índice selecionado do item.

Seção 14.13 Listas de seleção múltipla

- Uma lista de seleção múltipla permite ao usuário selecionar muitos itens de uma `JList`.
- O método `JList setFixedCellWidth` configura a largura de uma `JList`. O método `setFixedCellHeight` configura a altura de cada item em uma `JList`.
- Não há nenhum evento para indicar que um usuário fez múltiplas seleções em uma lista de seleção múltipla. Normalmente, um evento externo gerado por outro componente GUI (como um `JButton`) especifica quando as múltiplas seleções em uma `JList` devem ser processadas.
- O método `JList setListData` configura os itens exibidos em uma `JList`. O método `JList getSelectedValues` retorna um array de `Objects` para representar os itens selecionados em uma `JList`.

Seção 14.14 Tratamento de evento de mouse

- As interfaces listener de eventos `MouseListener` e `MouseMotionListener` são utilizadas para tratar eventos de mouse. Os eventos de mouse podem ser interrompidos por qualquer componente GUI que estenda `Component`.
- A interface `MouseInputListener` (pacote `javax.swing.event`) estende as interfaces `MouseListener` e `MouseMotionListener` para criar uma interface simples que contém todos os seus métodos.
- Todo método de tratamento do evento de mouse recebe um objeto `MouseEvent` que contém informações sobre o evento, incluindo as coordenadas x e y em que o evento ocorreu. As coordenadas são medidas a partir do canto superior esquerdo do componente GUI em que o evento ocorreu.
- Os métodos e as constantes de classe `InputEvent` (superclasse de `MouseEvent`) permitem que um aplicativo determine o botão do mouse em que o usuário clicou.
- A interface `MouseWheelListener` permite aos aplicativos responder à rotação da roda de um mouse.

Seção 14.15 Classes adaptadoras

- Muitas interfaces listener de eventos contêm múltiplos métodos. Para muitas dessas interfaces, os pacotes `java.awt.event` e `javax.swing.event` fornecem classes adaptadoras listeners de evento. Uma classe adaptadora implementa uma interface e fornece implementações padrão de seus métodos. Estenda uma classe adaptadora para herdar as implementações de método padrão e, então, sobrescreva o(s) método(s) de que você precisa.
- O método `MouseEvent getClickCount` retorna o número de cliques consecutivos de botão do mouse. Os métodos `isMetaDown` e `isAltDown` determinam em que botão do mouse o usuário clicou.

Seção 14.16 Subclasse JPanel para desenhar com o mouse

- Os componentes Swing leves que estendem a classe `JComponent` contêm o método `paintComponent`, que é chamado quando um componente Swing leve é exibido. Por sobrescrever esse método, você pode especificar como desenhar formas utilizando capacidades de imagens gráficas do Java.

- Ao personalizar um `JPanel` para desenhar, a subclasse deve anular o método `paintComponent` e chamar a versão de superclasse como a primeira instrução no corpo do método sobrescrito.
- As subclasses de `JComponent` suportam transparência. Quando um componente for opaco, `paintComponent` limpará o fundo do componente antes de o componente ser exibido.
- A transparência de um componente Swing leve pode ser configurada com o método `setOpaque` (um argumento `false` indica que o componente é transparente).
- A classe `Point` (pacote `java.awt`) representa uma coordenada *x*-*y*.
- A classe `Graphics` é utilizada para desenhar.
- O método `MouseEvent` `getPoint` obtém o `Point` em que ocorreu um evento de mouse.
- O método `repaint` (herdado indiretamente de classe `Component`) indica que um componente deve ser atualizado na tela o mais rápido possível.
- O método `paintComponent` recebe um parâmetro `Graphics` e é chamado automaticamente a qualquer hora em que um componente leve precisar ser exibido na tela.
- O método `Graphics` `fillOval` desenha uma oval sólida. Os dois primeiros argumentos são a coordenada *x* superior esquerda e a coordenada *y* superior esquerda do quadro delimitador. Os dois últimos argumentos representam a largura e a altura do quadro delimitador.

Seção 14.17 Tratamento de evento chave

- A interface `KeyListener` é utilizada para tratar eventos de teclado que são gerados quando as teclas são pressionadas e liberadas. O método `addKeyListener` da classe `Component` registra um `KeyListener`.
- O método `KeyEvent` `getKeyCode` obtém o código de tecla virtual da tecla que foi pressionada. A classe `KeyEvent` mantém um conjunto de código de tecla virtual constante que representa cada tecla no teclado.
- O método `KeyEvent` `getKeyText` retorna uma string que contém o nome da tecla que foi pressionada.
- O método `KeyEvent` `getKeyChar` obtém o valor Unicode do caractere digitado.
- O método `KeyEvent` `isActionKey` determina se a tecla em um evento era uma tecla de ação.
- O método `InputEvent` `getModifiers` determina se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) foi pressionada quando o evento de teclado ocorreu.
- O método `KeyEvent` `getKeyModifiersText` produz uma string que contém os nomes das teclas modificadoras pressionadas.

Seção 14.18 Introdução a gerenciadores de layout

- Os gerenciadores de layout organizam componentes GUI em um contêiner para propósitos de apresentação.
- Todos os gerenciadores de layout implementam a interface `LayoutManager` (pacote `java.awt`).
- O método `container` `setLayout` especifica o layout de um contêiner.
- `FlowLayout` coloca componentes da esquerda para a direita na ordem em que são adicionados ao contêiner. Quando o canto do contêiner é alcançado, os componentes continuam a exibir na linha seguinte. `FlowLayout` permite que os componentes GUI sejam alinhados à esquerda, centrados (o padrão) e alinhados à direita.
- O método `FlowLayout` `setAlignment` muda o alinhamento para um `FlowLayout`.
- `BorderLayout` (o padrão para um `JFrame`) organiza componentes em cinco regiões: `NORTE`, `SOUTH`, `EAST`, `WEST` e `CENTER`. `NORTH` corresponde à parte superior do contêiner.
- Um `BorderLayout` limita um `Container` a conter no máximo cinco componentes — um em cada região.
- `GridLayout` divide um contêiner em uma grade de linhas e colunas.
- O método `Container` `validate` recalcula o layout de um contêiner com base no gerenciador de layout atual para o `Container` e para o conjunto atual de componentes GUI exibidos.

Seção 14.19 Utilizando painéis para gerenciar layouts mais complexos

- GUIs complexas frequentemente consistem em múltiplos painéis, com os componentes de cada painel organizados em um layout específico. Todo `JPanel` pode ter componentes, incluindo outros painéis, anexados a ele com o método `Container` `add`.

Seção 14.20 `JTextArea`

- Uma `JTextArea` fornece uma área para manipular múltiplas linhas do texto. `JTextArea` é uma subclasse de `JTextComponent`.
- A classe `Box` é uma subclasse de `Container` que utiliza um gerenciador de layout `BoxLayout` para organizar os componentes GUI horizontal ou verticalmente.
- O método `Box` `static` `createHorizontalBox` cria um `Box` que organiza componentes da esquerda para a direita na ordem em que eles são anexados.
- O método `getSelectedText` retorna o texto selecionado de uma `JTextArea`.
- Você pode configurar as diretivas de barra de rolagem horizontal e vertical de um `JScrollPane` quando ele é construído. Os métodos `JScrollPane` `setHorizontalScrollBarPolicy` e `setVerticalScrollBarPolicy` podem ser utilizados para mudar as diretivas de barra de rolagem a qualquer hora.

Terminologia

AbstractButton, classe, 435
Abstract Window Toolkit (AWT), 423
ActionEvent, classe, 432
ActionListener, interface, 432
addActionListener, método da classe JTextField, 432
add, método da classe ButtonGroup, 442
add, método da classe JFrame, 427
addItemListener, método da classe AbstractButton, 440
addKeyListener, método da classe Component, 459
addListSelectionListener, método da classe JList, 447
addMouseListener, método da classe Component, 452
addMouseMotionListener, método da classe Component, 452
aparência e funcionamento plugável (*Pluggable* Look-And-Feel — PLAF), 425
área dedicada de desenho, 455
AWT (Abstract Window Toolkit), 423
barra de menus, 419
barra de rolagem, 444
barra de título, 419
baseado em evento, 428
BorderLayout, classe, 452
botão, 419
botão de alternação, 435
botão de comando, 435
botão de estado, 438
botão de opção, 435
botão de opção, grupo, 440
botão, rótulo, 436
Box, classe, 470
BoxLayout, classe, 470
ButtonGroup, classe, 440
caixa de combinação, 419
caixa de diálogo, 421
caixa de diálogo modal, 422
caixa de rolagem, 444
caixa de seleção, 435
caixa de seleção, rótulo, 440
CENTER, constante da classe BorderLayout, 452
CENTER, constante da classe FlowLayout, 463
classe adaptadora, 452
classe aninhada, 431
classe de primeiro nível, 431
classe interior anônima, 444
classe interna, 431
código de tecla virtual, 460
Component, classe, 424
componente GUI de peso leve, 424
componente opaco, 455
componente transparente, 455
componentes de peso leve, 424
componentes de peso pesado, 424
componentes GUI Swing, 423
consumir um evento, 432
Container, classe, 424
controles, 419
createHorizontalBox, método da classe Box, 470
diálogo, 421
diálogo de entrada, 421
diálogo de mensagem, 421
dicas de ferramenta, 425
diretivas de barra de rolagem, 470
EAST, constante da classe BorderLayout, 452
echo, caractere da classe JPasswordField, 429
eco, caractere, 429
espaçamento horizontal, 465
espaçamento vertical, 465
EventListenerList, classe, 434
evento, 428
evento de mouse, 435
evento externo, 449
fechar (ocultar) um diálogo, 422
ferramenta de design, 460
fillOval, método da classe Graphics, 457
FlowLayout, classe, 427
foco, 429
Font, classe, 440
fonte de evento, 432
gerenciador de layout, 427
getActionCommand, método da classe ActionEvent, 433
getClass, método da classe Object, 427
getClickCount, método da classe MouseEvent, 455
getContentPane, método da classe JFrame, 447
getIcon, método da classe JLabel, 428
getKeyChar, método da classe KeyEvent, 460
getKeyCode, método da classe KeyEvent, 460
getKeyModifiersText, método da classe KeyEvent, 460
getKeyText, método da classe KeyEvent, 460
getModifiers, método da classe InputEvent, 460
getPassword, método da classe JPasswordField, 433
getPoint, método da classe MouseEvent, 456
getResource, método da classe Class, 427
getSelectedIndex, método da classe JComboBox, 445
getSelectedIndex, método da classe JList, 447
getSelectedText, método da classe JTextComponent, 470
getSelectedValues, método da classe JList, 449
getStateChange, método da classe ItemEvent, 445
getText, método da classe JLabel, 428
getX, método da classe MouseEvent, 452
getY, método da classe MouseEvent, 452
Graphics, classe, 456
Graphics Interchange Format (GIF), 427
GridLayout, classe, 466
GUI, componente, 419
handler de evento, 428
HORIZONTAL_SCROLLBAR_ALWAYS, constante da classe JScrollPane, 470
HORIZONTAL_SCROLLBAR_AS_NEEDED, constante da classe JScrollPane, 471
HORIZONTAL_SCROLLBAR_NEVER, constante da classe JScrollPane, 471
Icon, interface, 427
ID de evento, 435
ImageIcon, classe, 427
implementar múltiplas interfaces, 449
índice de um JComboBox, 444
InputEvent, classe, 450
interface gráfica com usuário (Graphical User Interface — GUI), 419
interface ouvinte de evento, 431
isActionKey, método da classe KeyEvent, 460
isAltDown, método da classe InputEvent, 460
isControlDown, método da classe InputEvent, 460
isMetaDown, método da classe InputEvent, 460
isShiftDown, método da classe InputEvent, 460
ItemEvent, classe, 440
ItemListener, interface, 440
janelas, sistema, 424
java.awt, pacote, 423
Java Foundation Classes (JFC), 423
javax.swing, pacote, 423
javax.swing.event, pacote, 433
JButton, classe, 436
JCheckBox, classe, 438
JComboBox, classe, 443
JFrame.EXIT_ON_CLOSE, 428
JLabel, classe, 425
Joint Photographic Experts Group (JPEG), 427
JOptionPane, classe, 421
JPasswordField, classe, 429
JRadioButton, classe, 438
JTextArea, classe, 468
JTextComponent, classe, 429
JTextField, classe, 429
JToggleButton, classe, 438
KeyEvent, classe, 435
KeyListener, interface, 435
keyPressed, método da interface KeyListener, 458
keyReleased, método da interface KeyListener, 458
keyTyped, método da interface KeyListener, 458
layoutContainer, método da interface LayoutManager, 463
LayoutManager, interface, 460
LayoutManager2, interface, 463
LEFT, constante da classe FlowLayout, 463
lista de seleção múltipla, 445
lista de seleção única, 445
lista drop-down, 443
ListSelectionEvent, classe, 445
ListSelectionListener, interface, 447
ListSelectionModel, classe, 447
ListSelectionModel, interface, 447
localização, 425
menu, 419
mnemônico, 425
modelo de evento de delegação, 434
modos de seleção, 447
MouseEvent, classe, 435
MouseInputListener, interface, 449
MouseListener, interface, 435
MouseMotionListener, interface, 435

MouseWheelEvent, classe, 450
MouseWheelListener, interface, 450
mouseWheelMoved, método da interface MouseWheelListener, 450
Nimbus, aparência e comportamento, 420
NORTH, constante da classe BorderLayout, 452
opções mutuamente exclusivas, 440
ouvir eventos, 431
paintComponent, método da classe JComponent, 455
PLAIN_MESSAGE, constante da classe JOptionPane, 422
Point, classe, 456
Portable Network Graphics (PNG), 427
quebra de linha, 470
registrar um handler de evento, 431
repaint, método da classe Component, 456
RIGHT, constante da classe FlowLayout, 463
rollover Icon, 437
rótulo, 425
seta de rolagem, 444
setAlignment, método da classe FlowLayout, 463
setBackground, método da classe Component, 447
setDefaultCloseOperation, método da classe JFrame, 428
setDisabledTextColor, método da classe JTextComponent, 460
setEditable, método da classe JTextComponent, 431
setFixedCellHeight, método da classe JList, 449
setFixedCellWidth, método da classe JList, 449
setFont, método da classe Component, 440
setHorizontalAlignment, método da classe JLabel, 428
setHorizontalScrollBarPolicy, método da classe JScrollPane, 470
setHorizontalTextPosition, método da classe JLabel, 428
setIcon, método da classe JLabel, 427
setLayout, método da classe Container, 427
setLineWrap, método da classe JTextArea, 470
setListData, método da classe JList, 449
setMaximumRowCount, método da classe JComboBox, 444
setOpaque, método da classe JComponent, 455
setRolloverIcon, método da classe AbstractButton, 438
setSelectionMode, método da classe JList, 447
setSize, método da classe JFrame, 428
setText, método da classe JLabel, 428
setToolTipText, método da classe JComponent, 427
setVerticalAlignment, método da classe JLabel, 428
setVerticalScrollBarPolicy, método da classe JScrollPane, 470
setVerticalTextPosition, método da classe JLabel, 428
setVisible, método da classe Component, 428
setVisibleRowCount, método da classe JList, 446
showInputDialog, método da classe JOptionPane, 422
showMessageDialog, método da classe JOptionPane, 422
SINGLE_INTERVAL_SELECTION, constante de interface ListSelectionModel, 447
SINGLE_SELECTION, constante de interface ListSelectionModel, 447
SOUTH, constante da classe BorderLayout, 452
string vazia, 433
SwingConstants, interface, 428
tecla de ação, 458
teclado, evento, 435
transparência de JComponent, 455
tratamento de evento, 428
uso de letras maiúsculas e minúsculas no estilo de frases, 422
uso de letras maiúsculas e minúsculas no estilo título de livro, 422
validate, método da classe Container, 467
valueChanged, método da interface ListSelectionListener, 447
VERTICAL_SCROLLBAR_ALWAYS, constante da classe JScrollPane, 470
VERTICAL_SCROLLBAR_AS_NEEDED, constante da classe JScrollPane, 471
VERTICAL_SCROLLBAR_NEVER, constante da classe JScrollPane, 471
WEST, constante da classe BorderLayout, 452
widgets, 419

Exercícios de autorrevisão

14.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) O método _____ é chamado quando o mouse é movido sem pressionamento de botões e um ouvinte de evento é registrado para tratar o evento.

b) O texto que não pode ser modificado pelo usuário é chamado texto _____.

c) Um(a) _____ organiza os componentes GUI em um `Container`.

d) O método `add` para anexar componentes GUI é um método da classe _____.

e) GUI é um acrônimo de _____.

f) O método _____ é utilizado para especificar o gerenciador de layout para um contêiner.

g) Uma chamada de método `mouseDragged` é precedida por uma chamada de método _____ e seguida por uma chamada de método _____.

h) A classe _____ contém métodos que exibem diálogos de mensagem e diálogos de entrada.

i) Um diálogo de entrada capaz de receber entrada do usuário é exibido com o método _____ da classe _____.

j) Um diálogo capaz de exibir uma mensagem para o usuário é exibido com o método _____ da classe _____.

k) Tanto `JTextFields` como `JTextAreas` estendem diretamente a classe _____.

14.2 Determine se cada sentença é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) `BorderLayout` é o gerenciador de layout padrão do painel de conteúdo de um `JFrame`.

b) Quando o cursor do mouse é movido nos limites de um componente GUI, o método `mouseOver` é chamado.

c) Um `JPanel` não pode ser adicionado a outro `JPanel`.

d) Em um `BorderLayout`, dois botões adicionados à região `NORTH` serão colocados lado a lado.

e) Um máximo de cinco componentes pode ser adicionado a um `BorderLayout`.

f) As classes internas não têm permissão de acessar os membros da classe que os envolve.

g) Um texto da `JTextArea` é sempre de leitura (read-only).

h) A classe `JTextArea` é uma subclasse direta da classe `Component`.

14.3 Localize o(s) erro(s) em cada uma das seguintes instruções e explique como corrigi-lo(s).

a)
```
buttonName = JButton( "Caption" );
```

b)
```
JLabel aLabel, JLabel; // cria referências
```

c)
```
txtField = new JTextField( 50, "Default Text" );
```

d)
```
setLayout( new BorderLayout() );
button1 = new JButton( "North Star" );
button2 = new JButton( "South Pole" );
add( button1 );
add( button2 );
```

Respostas dos exercícios de autorrevisão

14.1 a) `mouseMoved`. b) não editável (de leitura). c) gerenciador de layout. d) `Container`. e) interface gráfica com o usuário. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.

14.2 a) Verdadeira.

b) Falsa. O método `mouseEntered` é chamado.

c) Falsa. Um `JPanel` pode ser adicionado ao outro `JPanel`, porque `JPanel` é uma subclasse indireta de `Component`. Desse modo, um `JPanel` é um `Component`. Qualquer `Component` pode ser adicionado a um `Container`.

d) Falsa. Apenas o último botão adicionado será exibido. Lembre-se de que apenas um componente deve ser adicionado a cada região em um `BorderLayout`.

e) Verdadeira. [*Nota:* os painéis que contêm vários componentes podem ser adicionados a cada região.]

f) Falsa. As classes internas têm acesso a todos os membros da declaração de classe que os envolve.

g) Falsa. `JTextArea`s são editáveis por padrão.

h) Falsa. `JTextArea` deriva da classe `JTextComponent`.

14.3 a) `new` é necessário para criar um objeto.

b) `JLabel` é um nome de classe e não pode ser utilizado como um nome de variável.

c) Os argumentos passados para o construtor estão invertidos. A `String` deve ser passada primeiro.

d) `BorderLayout` foi configurado e os componentes que estão sendo adicionados sem especificar a região são ambos adicionados à região centro. As instruções `add` adequadas podem ser
```
add( button1, BorderLayout.NORTH );
add( button2, BorderLayout.SOUTH );
```

Exercícios

14.4 Preencha as lacunas em cada uma das seguintes afirmações:

a) A classe `JTextField` estende diretamente a classe _____.

b) O método `Container` _____ anexa um componente GUI a um contêiner.

c) O método _____ é chamado quando um botão de mouse é liberado (sem mover o mouse).

d) A classe _____ é utilizada para criar um grupo de `JRadioButton`s.

14.5 Determine se cada sentença é *verdadeira* ou *falsa* . Se *falsa*, explique por quê.

a) Apenas um gerenciador de layout pode ser utilizado por `Container`.

b) Os componentes GUI podem ser adicionados a um `Container` em qualquer ordem em um `BorderLayout`.

c) `JRadioButton`s fornecem uma série de opções mutuamente exclusivas (isto é, apenas uma pode ser `true` por vez).

d) O método `Graphics` `setFont` é utilizado para configurar a fonte para campos de texto.

e) Uma `JList` exibe uma barra de rolagem se houver mais itens na lista do que podem ser exibidos.

f) Um objeto `Mouse` tem um método chamado `mouseDragged`.

14.6 Determine se cada sentença é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Um `JPanel` é um `JComponent`.

b) Um `JPanel` é um `Component`.

c) Um `JLabel` é um `Container`.

d) Um `JList` é um `JPanel`.

e) Um `AbstractButton` é um `JButton`.

f) Um `JTextField` é um `Object`.

g) `ButtonGroup` é uma subclasse de `JComponent`.

14.7 Localize qualquer erro em cada uma das seguintes linhas de código e explique como corrigi-lo.

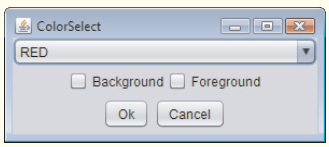
a) `import javax.swing.JFrame`

b) `panelObject.GridLayout( 8, 8 ); // configura GridLayout`

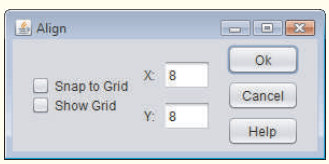
c) `container.setLayout( new FlowLayout( FlowLayout.DEFAULT ) );`

d) `container.add( eastButton, EAST ); // BorderLayout`

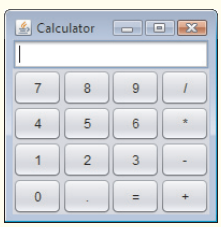
14.8 Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



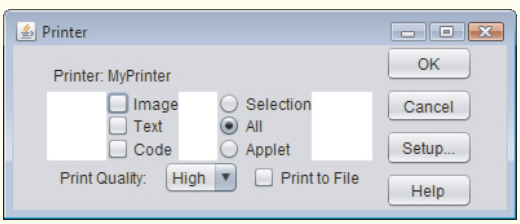
14.9 Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



14.10 Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



14.11 Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



14.12 (***Conversão de temperatura***) Escreva um aplicativo de conversão de temperatura que converte de Fahrenheit para Celsius. A temperatura de Fahrenheit deve ser inserida pelo teclado (via um `JTextField`). Um `JLabel` deve ser utilizado para exibir a temperatura convertida. Utilize a seguinte fórmula para a conversão:

$$Celsius = \frac{5}{9} \times (Fahrenheit - 32)$$

14.13 (***Modificação de conversão de temperatura***) Aprimore o aplicativo de conversão de temperatura do Exercício 14.12 adicionando a escala de temperatura Kelvin. O aplicativo também deve permitir ao usuário fazer conversões entre quaisquer duas escalas. Utilize a seguinte fórmula para a conversão entre Kelvin e Celsius (além da fórmula no Exercício 14.12):

Kelvin = Celsius + 273,15

14.14 (***Jogo Adivinhe o número***) Escreva um aplicativo que joga o "Adivinhe o número" como mostrado a seguir: Seu aplicativo escolhe o número a ser adivinhado selecionando um inteiro aleatoriamente no intervalo 1–1000. O aplicativo então exibe o seguinte em um rótulo:

```
I have a number between 1 and 1000. Can you guess my number?
Please enter your first guess.
```

Um `JTextField` deve ser usado para entrar a suposição. Conforme cada suposição é inserida, a cor de fundo deve mudar para vermelho ou azul. O vermelho indica que o usuário está ficando "mais quente", e o azul, "mais frio". Um `JLabel` deve exibir `"Too High"` ou `"Too Low"` para ajudar o usuário a zerar. Quando o usuário obtiver a resposta correta, `"Correct!"` deve ser exibido, e o `JTextField` usado para a entrada deve ser mudado para ser não editável. Um `JButton` deve ser fornecido para permitir ao usuário jogar de novo. Quando o `JButton` for clicado, um novo número aleatório deverá ser gerado e a entrada `JTextField` deve ser alterada para o estado editável.

14.15 (***Exibindo eventos***) Frequentemente, é útil exibir os eventos que ocorrem durante a execução de um aplicativo. Isso pode ajudá-lo a entender quando os eventos ocorrem e como eles são gerados. Escreva um aplicativo que permite ao usuário gerar e processar cada evento discutido

neste capítulo. O aplicativo deve fornecer os métodos das interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` e `KeyListener` para exibir as mensagens quando os eventos ocorrem. Utilize o método `toString` para converter os objetos de evento recebidos em cada handler de evento para `String`s que possam ser exibidas. O método `toString` cria uma `String` contendo todas as informações no objeto de evento.

14.16 (***Jogo de dados baseado em GUI***) Modifique o aplicativo da Seção 6.10 para fornecer uma GUI que permite ao usuário clicar em um `JButton` para lançar os dados. O aplicativo também deve exibir quatro `JLabel`s e quatro `JTextField`s, com um `JLabel` para cada `JTextField`. Os `JTextField`s devem ser usados para exibir os valores de cada dado e a soma dos dados depois de cada lançamento. O ponto deve ser exibido no quarto `JTextField` quando o usuário não ganhar ou perder no primeiro lançamento e deve continuar a ser exibido até que o jogo seja perdido.

(*Opcional*) *Exercício de Estudo de caso de GUI e imagens gráficas: expandindo a interface*

14.17 (***Aplicativo de desenho interativo***) Neste exercício, você implementará um aplicativo GUI que utiliza a hierarquia `MyShape` do Exercício 10.2 do estudo de caso sobre GUI para criar um aplicativo de desenho interativo. Você criará duas classes para a GUI e fornecerá uma classe de teste que carrega o aplicativo. As classes da hierarquia `MyShape` não exigem nenhuma alteração adicional.

A primeira classe a ser criada é uma subclasse de `JPanel` chamada `DrawPanel`, que representa a área em que o usuário desenha as formas. A classe `DrawPanel` deve ter as seguintes variáveis de instância:

a) Um array `shapes` do tipo `MyShape` que armazenará todas as formas que o usuário desenhar.

b) Um inteiro `shapeCount` que conta o número de formas no array.

c) Um inteiro `shapeType` que determina o tipo de forma a ser desenhado.

d) Um `MyShape` `currentShape` que representa a forma atual que o usuário está desenhando.

e) Um `Color` `currentColor` que representa a cor atual de desenho.

f) Um booleano `filledShape` que determina se desenhar ou não uma forma preenchida.

g) Um `JLabel` `statusLabel` que representa a barra de status. A barra de status exibirá as coordenadas da posição atual do mouse.

A classe `DrawPanel` também deve declarar os seguintes métodos:

a) Método `paintComponent` sobrescrito que desenha as formas no array. Utilize a variável de instância `shapeCount` para determinar quantas formas desenhar. O método `paintComponent` também deve chamar o método `draw` de `currentShape`, contanto que `currentShape` não seja `null`.

b) Configure métodos para `shapeType`, `currentColor` e `filledShape`.

c) O método `clearLastShape` deve eliminar a última forma desenhada decrementando a variável de instância `shapeCount`. Assegure que `shapeCount` nunca é menor que zero.

d) O método `clearDrawing` deve remover todas as formas no desenho atual configurando `shapeCount` como zero.

Os métodos `clearLastShape` e `clearDrawing` devem chamar o método `repaint` (herdado de `JPanel`) para atualizar o desenho no `DrawPanel` indicando que o sistema deve chamar o método `paintComponent`.

A classe `DrawPanel` também deve fornecer tratamento de evento para permitir ao usuário desenhar com o mouse. Crie uma única classe interna que tanto estenda `MouseAdapter` como implemente `MouseMotionListener` para tratar todos os eventos de mouse em uma classe.

Na classe interna, sobrescreva o método `mousePressed` para que ele atribua a `shapeType` uma nova forma do tipo especificado por `currentShape` e inicialize ambos os pontos como a posição do mouse. Em seguida, sobrescreva o método `mouseReleased` para terminar de desenhar a forma atual e colocá-la no array. Configure o segundo ponto de `currentShape` como a posição atual do mouse e adicione `currentShape` ao array. A variável de instância `shapeCount` determina o índice de inserção. Configure `currentShape` como `null` e chame o método `repaint` para atualizar o desenho com a nova forma.

Sobrescreva o método `mouseMoved` para configurar o texto do `statusLabel` de modo que ele exiba as coordenadas de mouse — isso atualizará o rótulo com as coordenadas toda vez que o usuário mover (mas não arrastar) o mouse dentro do `DrawPanel`. Em seguida, sobrescreva o método `mouseDragged` de modo que ele configure o segundo ponto do `currentShape` como a posição de mouse atual e chame o método `repaint`. Isso permitirá ao usuário ver a forma ao arrastar o mouse. Além disso, atualize o `JLabel` em `mouseDragged` com a posição atual do mouse.

Crie um construtor para `DrawPanel` que tem um único parâmetro `JLabel`. No construtor, inicialize `statusLabel` com o valor passado para o parâmetro. Também inicialize o array `shapes` com 100 entradas, `shapeCount` como 0, `shapeType` como o valor que representa uma linha, `currentShape` como `null` e `currentColor` como `Color.BLACK`. O construtor então deve configurar a cor de fundo do `DrawPanel` como `.WHITE` e registrar o `MouseListener` e `MouseMotionListener` para que o `JPanel` trate adequadamente os eventos de mouse.

Em seguida, crie uma subclasse `JFrame` chamada `DrawFrame` que forneça uma GUI para permitir ao usuário controlar vários aspectos de desenho. Para o layout do `DrawFrame`, recomendamos um `BorderLayout`, com os componentes na região `NORTH`, o principal painel de desenho na região `CENTER` e uma barra de status na região `SOUTH`, como na Figura 14.49. No painel superior, crie os componentes listados abaixo. O handler de evento de cada componente deve chamar o método adequado na classe `DrawPanel`.

a) Um botão para desfazer a última forma desenhada.

b) Um botão para eliminar todas as formas do desenho.

c) Uma caixa de combinação para selecionar a partir das 13 cores predefinidas.

d) Uma caixa de combinação para selecionar a forma desenhar.

e) Uma caixa de seleção que especifica se uma forma deve ou não ter preenchimento.

Declare e crie os componentes de interface no construtor de `DrawFrame`. Você precisará criar a barra de status `JLabel` antes de criar o `DrawPanel`, para que possa passar o `JLabel` como um argumento para o construtor do `DrawPanel`. Por fim, crie uma classe de teste que inicialize e exiba o `DrawFrame` para executar o aplicativo.

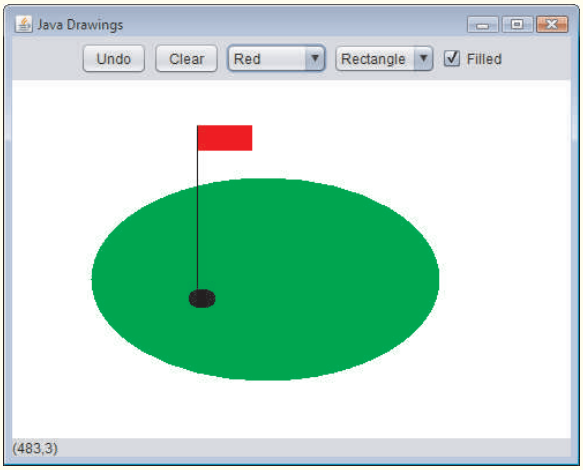


Figura 14.49 | Interface para desenhar formas.

14.18 (***Versão baseada em GUI do Estudo de Caso de ATM***) Reimplemente o Estudo de Caso de ATM dos capítulos 12–13 como um aplicativo baseado em GUI. Utilize componentes GUI para aproximar-se da interface com o usuário ATM mostrada na Figura 12.1. Para o dispensador de dinheiro e o slot de depósito, utilize `JButtons` rotulados de **Remove Cash** e **Insert Envelope**. Isso permitirá que o aplicativo receba eventos que indicam quando o usuário saca o dinheiro e quando insere um envelope de depósito, respectivamente.

Fazendo a diferença

14.19 (***Ecofont***) Ecofont (`www.ecofont.eu/ecofont_en.html`) — desenvolvida pela Spranq (uma empresa situada na Holanda) — é uma fonte de computador gratuita, distribuída sob o regime Open Source e projetada para reduzir 20% da quantidade de tinta empregada na impressão, reduzindo assim também o número de cartuchos de tinta usados e o impacto ambiental dos processos de fabricação e entrega (utilizando menos energia, menos combustível para despacho, e assim por diante). A fonte, baseada em Verdana sem serifa, tem pequenos "furos" circulares que não são visíveis em menores tamanhos — como os tamanhos 9 ou 10 pontos mais utilizados. Baixe a Ecofont e instale o arquivo de fontes `Spranq_eco_sans_regular.ttf` utilizando as instruções do site Ecofont. Depois, desenvolva um programa baseado em GUI que permite digitar uma linha de texto com a Ecofont. Crie os botões **Increase Font Size** e **Decrease Font Size** que permitem aumentar e diminuir o tamanho da fonte um ponto por vez. Comece com um tamanho padrão de 9 pontos. À medida que aumentar o tamanho, você será capaz de ver mais claramente os furos nas letras. À medida que você reduz, os furos serão menos aparentes. Qual é o menor tamanho de fonte em que você começa a notar os furos?

14.20 (***Professor de digitação: aprimorando uma habilidade crucial na era dos computadores***) Digitar rápida e corretamente é uma habilidade essencial para trabalhar efetivamente com computadores e Internet. Neste exercício, você construirá um aplicativo GUI que pode ajudar usuários a aprender a digitar corretamente sem olhar para o teclado. O aplicativo deve exibir um teclado virtual (Figura 14.50) e permitir ao usuário observar o que ele está digitando na tela sem olhar para o teclado real. Utilize `JButtons` para representar as teclas. À medida que o usuário pressiona cada tecla, o aplicativo destaca o `JButton` correspondente na GUI e adiciona o caractere a uma `JTextArea` que mostra o que o usuário digitou até o momento. [*Dica:* para destacar um `JButton`, utilize seu método `setBackground` para mudar a cor de fundo. Quando a tecla é liberada, reinicialize a cor de fundo original. Você pode obter a cor de fundo original do `JBUTTON` com o método `getBackground` antes de mudar sua cor.]

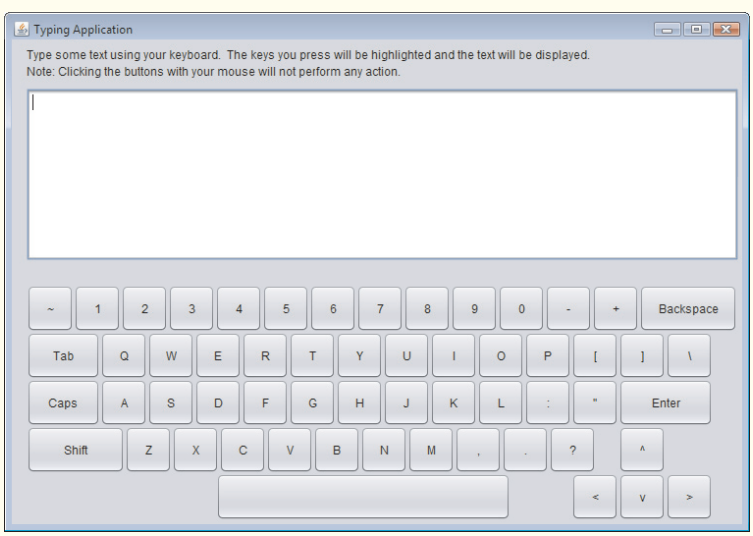


Figura 14.50 | Professor de digitação.

Você pode testar o seu programa digitando um pangrama — uma frase em que são usadas todas as letras do alfabeto pelo menos uma vez — como "The quick brown fox jumped over lazy dogs". Você pode encontrar outros pangramas na Web.

Para tornar o programa mais interessante, você poderia monitorar a exatidão do usuário. Poderia fazer o usuário digitar frases específicas que você teria pré-armazenado no programa e exibir na tela acima do teclado virtual. Poderia monitorar quantas teclas o usuário pressiona corretamente e quantas incorretamente. Você também poderia monitorar as teclas com que o usuário tem dificuldades e exibir um relatório que mostre essas teclas.

Uma imagem vale dez mil palavras.
— **Provérbio chinês**

Trate a natureza em termos de cilindros, esferas, cones, tudo em perspectiva.
— **Paul Cézanne**

As cores, como a expressão facial, mudam conforme as emoções.
— **Pablo Picasso**

Nada se torna real até ser experimentado – mesmo um provérbio não significa nada para você até sua vida ilustrá-lo.
— **John Keats**

15

Imagens gráficas e Java 2D™

Objetivos

Neste capítulo, você aprenderá:

- A entender contextos gráficos e objetos gráficos.
- A manipular cores.
- A manipular fontes.
- A utilizar métodos da classe `Graphics` para desenhar linhas, retângulos, retângulos com cantos arredondados, retângulos tridimensionais, ovais, arcos e polígonos.
- A utilizar métodos da classe `Graphics2D` da `Java 2D` API para desenhar linhas, retângulos, retângulos com cantos arredondados, elipses, arcos e caminhos gerais.
- A especificar as características de `Paint` e `Stroke` das formas exibidas com `Graphics2D`.

Sumário

15.1 Introdução
15.2 Contextos gráficos e objetos gráficos
15.3 Controle de cor
15.4 Manipulando fontes
15.5 Desenhando linhas, retângulos e ovais
15.6 Desenhando arcos
15.7 Desenhando polígonos e polilinhas
15.8 Java 2D API
15.9 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

15.1 Introdução

Neste capítulo, oferecemos uma visão geral de várias capacidades do Java para desenhar formas bidimensionais, controlar cores e controlar fontes. Um dos atrativos iniciais do Java foi seu suporte a imagens gráficas que permitia aos programadores aprimorar visualmente seus aplicativos. O Java agora contém muitas capacidades mais sofisticadas de desenho como parte da Java 2D™ API. Este capítulo inicia com uma introdução a muitas capacidades originais de desenho do Java. Em seguida, apresentamos várias capacidades mais poderosas da Java 2D, como controlar o estilo das linhas utilizado para desenhar formas e a maneira como formas são preenchidas com cores e padrões.

A Figura 15.1 mostra uma parte da hierarquia de classe Java que inclui várias das classes gráficas básicas e várias das classes e interfaces Java 2D API abordadas neste capítulo. A classe **`Color`** contém métodos e constantes para manipular cores. A classe `JComponent` contém o método `paintComponent`, que é utilizado para desenhar imagens gráficas em um componente. A classe **`Font`** contém métodos e constantes para manipular fontes. A classe **`FontMetrics`** contém métodos para obter informações de fonte. A classe **`Graphics`** contém métodos para desenhar strings, linhas, retângulos e outras formas. A classe **`Graphics2D`**, que estende a classe `Graphics`, é utilizada para desenhar com a Java 2D API. A classe **`Polygon`** contém métodos para criar polígonos. A metade inferior da figura lista várias classes e interfaces da Java 2D API. A classe **`BasicStroke`** ajuda a especificar as características do desenho de linhas. As classes **`GradientPaint`** e **`TexturePaint`** ajudam a especificar as características para preencher formas com cores ou padrões. As classes `GeneralPath`, `Line2D`, `Arc2D`, `Ellipse2D`, `Rectangle2D` e `RoundRectangle2D` representam várias formas 2D do Java. [*Nota:* começamos discutindo as capacidades gráficas originais do Java e, então, avançamos para a Java 2D API. As classes que eram parte das capacidades gráficas originais do Java são agora consideradas parte da API 2D do Java.]

Para começar a desenhar em Java, devemos primeiro entender o **sistema de coordenadas** do Java (Figura 15.2), que é um esquema para identificar cada ponto na tela. Por padrão, o canto superior esquerdo de um componente GUI (por exemplo, uma janela) tem as coordenadas (0, 0). Um par de coordenadas é composto de uma **coordenada *x*** (a **coordenada horizontal**) e uma **coordenada *y*** (a **coordenada vertical**). A coordenada *x* é a distância horizontal que vai do lado direito ao lado esquerdo da tela. A coordenada *y* é a distância vertical de baixo para cima da tela. O **eixo *x*** descreve cada coordenada horizontal e o **eixo *y***, cada coordenada vertical. As coordenadas são utilizadas para indicar onde as imagens gráficas devem ser exibidas em uma tela. As unidades coordenadas são medidas em **pixels** (que significa "picture element"). Um pixel é a menor unidade de exibição de resolução do monitor.

Dica de portabilidade 15.1

Monitores diferentes têm resoluções diferentes (isto é, a densidade dos pixels varia). Isso pode fazer as imagens gráficas aparecerem com tamanhos diferentes em diferentes monitores ou no mesmo monitor com diferentes configurações.

15.2 Contextos gráficos e objetos gráficos

Um **contexto gráfico** permite desenhar na tela. Um objeto `Graphics` gerencia um contexto gráfico e desenha pixels na tela que representam texto e outros objetos gráficos (por exemplo, linhas, elipses, retângulos e outros polígonos). Objetos `Graphics` contêm métodos para desenhar, manipular fontes, manipular cores e coisas do tipo.

A classe `Graphics` é uma classe `abstract` (isto é, objetos `Graphics` não podem ser instanciados). Isso contribui para a portabilidade do Java. Como o desenho é realizado de várias maneiras em cada plataforma que suporta o Java, não poderia haver somente uma implementação das capacidades de desenho em todos os sistemas. Por exemplo, as capacidades gráficas que permitem que um PC, executando o Windows Microsoft, desenhe um retângulo são diferentes daquelas que permitem a uma estação de trabalho Linux desenhar um retângulo — e ambas são diferentes das capacidades gráficas que permitem ao Macintosh desenhar um retângulo. Quando o Java é implementado em cada plataforma, é criada uma subclasse de `Graphics` que implementa as capacidades de desenho. Essa implementação permanece oculta pela classe `Graphics`, que fornece a interface que permite utilizar imagens gráficas de uma maneira independente de plataforma.

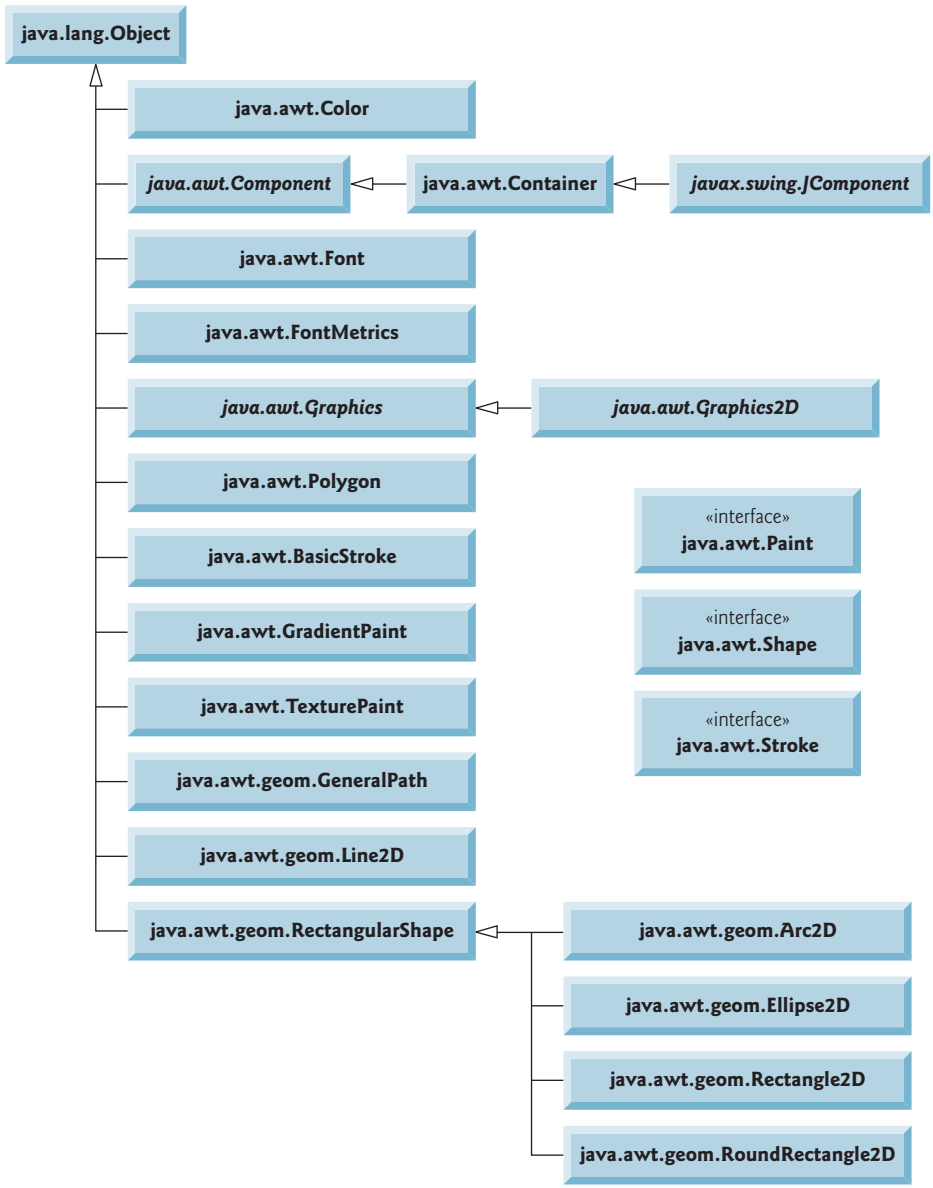


Figura 15.1 | As classes e interfaces utilizadas neste capítulo são provenientes das capacidades gráficas originais do Java e da Java 2D API.

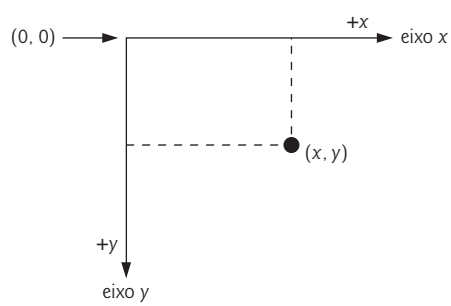


Figura 15.2 | Sistema de coordenadas Java. As unidades são medidas em pixels.

Lembre-se do Capítulo 14 de que a classe `Component` é a superclasse para muitas das classes no pacote `java.awt`. A classe `Jcomponent` (pacote `javax.swing`), que herda indiretamente da classe `Component`, contém um método `paintComponent` que pode ser utilizado para desenhar imagens gráficas. O método `paintComponent` recebe um objeto `Graphics` como um argumento. Esse objeto é passado para o método `paintComponent` pelo sistema quando um componente Swing, de peso leve, precisa ser repintado. O cabeçalho para o método `paintComponent` é

```
public void paintComponent( Graphics g )
```

O parâmetro `g` recebe uma referência a uma instância da subclasse específica de sistema que a `Graphics` estende. O cabeçalho do método anterior deve parecer familiar para você — é o mesmo que utilizamos em alguns aplicativos no Capítulo 14. Na verdade, a classe `JComponent` é uma superclasse de `JPanel`. Muitas capacidades da classe `JPanel` são herdadas de classe `JComponent`.

Você raramente chama o método `paintComponent` diretamente, porque desenhar elementos gráficos é um processo baseado em eventos. Como mencionamos no Capítulo 11, o Java utiliza um modelo multithread para a execução de programas. Cada thread é uma atividade paralela. Cada programa pode ter muitas threads. Ao criar um aplicativo baseado na GUI, uma dessas threads é conhecida como a **thread de despacho do evento (EDT)** e é utilizada para processar todos os eventos GUI. Todo o desenho e a manipulação dos componentes GUI devem ser realizados nessa thread. Quando um aplicativo GUI é executado, o contêiner de aplicativo chama o método `paintComponent` (na thread de despacho do evento) para cada componente leve à medida que a GUI é exibida. Para `paintComponent` ser chamado novamente, deve ocorrer um evento (como cobrir e descobrir o componente com uma outra janela).

Se precisar de `paintComponent` para executar (isto é, se quiser atualizar os elementos gráficos desenhados em um componente Swing), você poderá chamar o método **`repaint`**, que é herdado por todos os `JComponents` indiretamente da classe `Component` (pacote `java.awt`). O método `repaint` é frequentemente chamado para solicitar uma chamada ao método `paintComponent`. O cabeçalho para `repaint` é

```
public void repaint()
```

15.3 Controle de cor

A classe `Color` declara métodos e constantes para manipular cores em um programa Java. As constantes color pré-declaradas estão resumidas na Figura 15.3, e vários construtores e métodos color estão resumidos na Figura 15.4. Observe que dois dos métodos na Figura 15.4 são métodos `Graphics` específicos para cores.

Constante Color	Valor RGB
`public final static Color RED`	255, 0, 0
`public final static Color GREEN`	0, 255, 0
`public final static Color BLUE`	0, 0, 255
`public final static Color ORANGE`	255, 200, 0
`public final static Color PINK`	255, 175, 175
`public final static Color CYAN`	0, 255, 255
`public final static Color MAGENTA`	255, 0, 255
`public final static Color YELLOW`	255, 255, 0
`public final static Color BLACK`	0, 0, 0
`public final static Color WHITE`	255, 255, 255
`public final static Color GRAY`	128, 128, 128
`public final static Color LIGHT_GRAY`	192, 192, 192
`public final static Color DARK_GRAY`	64, 64, 64

Figura 15.3 | Constantes `Color` e seus valores de RGB.

Cada cor é criada a partir de um componente vermelho, um verde e um azul. Juntos, esses componentes são chamados **valores RGB**. Todos os três componentes RGB podem ser inteiros no intervalo de 0 a 255 ou podem ser valores de ponto flutuante nos intervalo de 0,0 a 1,0. O primeiro componente RGB especifica o valor de vermelho; o segundo, o valor de verde; e o terceiro, o valor de azul. Quanto maior o valor de RGB, maior a quantidade dessa cor particular. O Java permite escolher entre $256 \times 256 \times 256$ cores (aproximadamente 16,7 milhões). Nem todos os computadores são capazes de exibir todas essas cores. A tela de computador exibira a cor mais próxima possível.

Método	Descrição
Construtores e métodos `Color`	
`public Color( int r, int g, int b )`	Cria uma cor com base nos componentes azul, verde, vermelho expressos como valores de ponto flutuante de 0,0 a 1,0.
`public Color( float r, float g, float b )`	Retorna um valor entre 0 e 255 representando o componente vermelho.
`public int getRed()`	Retorna um valor entre 0 e 255 representando o conteúdo de vermelho.
`public int getGreen()`	Retorna um valor entre 0 e 255 representando o conteúdo de verde.
`public int getBlue()`	Retorna um valor entre 0 e 255 representando o conteúdo de azul.
Métodos `Graphics` *para manipular* `Colors`	
`public Color getColor()`	Retorna o objeto `Color` que representa as cores atuais no contexto gráfico.
`public void setColor( Color c )`	Configura a cor atual para desenho com o contexto gráfico.

Figura 15.4 | Métodos `Color` e métodos `Graphics` relacionados com cor.

Dois construtores da classe `Color` são mostrados na Figura 15.4 — um que recebe três argumentos `int` e outro que recebe três argumentos `float`, com cada argumento especificando a quantidade de vermelho, verde e azul. Os valores `int` devem estar no intervalo de 0 a 255 e os valores `float` devem estar no intervalo de 0,0 a 1,0. O novo objeto `Color` terá as quantidades de vermelho, verde e azul especificadas. Os métodos **`getRed`**, **`getGreen`** e **`getBlue`** retornam valores inteiros de 0 a 255 representando as quantidades de vermelho, verde e azul, respectivamente. O método `Graphics` **`getColor`** retorna um objeto `Color` representando a cor de desenho atual. O método `Graphics` **`setColor`** configura a cor de desenho atual.

Desenhando em cores diferentes

As figuras 15.5 e 15.6 demonstram vários métodos da Figura 15.4 desenhando retângulos preenchidos e `String`s em várias diferentes cores. Quando o aplicativo inicia a execução, o método `paintComponent` da classe `ColorJPanel` (linhas 10–37 da Figura 15.5) é chamado para pintar a janela. A linha 17 utiliza o método `Graphics` `setColor` para configurar as cores atuais de desenho. O método `setColor` recebe um objeto `Color`. A expressão `new Color( 255, 0, 0 )` cria um novo objeto `Color` que representa vermelho (valor vermelho `255` e `0` para os valores de azul e verde). A linha 18 utiliza o método `Graphics` **`fillRect`** para desenhar um retângulo preenchido com a cor atual. O método `fillRect` desenha um retângulo baseado em seus quatro argumentos. Os dois primeiros valores de inteiro representam a coordenada *x* superior esquerda e a coordenada *y* superior esquerda onde o objeto `Graphics` começa a desenhar o retângulo. O terceiro e quarto argumentos são números inteiros não negativos que representam a largura e a altura do retângulo em pixels, respectivamente. Um retângulo desenhado com o método `fillRect` é preenchido pela cor atual do objeto `Graphics`.

```
1  // Figura 15.5: ColorJPanel.java
2  // Demonstrando Colors.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import javax.swing.JPanel;
6
7  public class ColorJPanel extends JPanel
8  {
9     // desenha retângulos e Strings em cores diferentes
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g ); // chama o paintComponent da superclasse
13
14       this.setBackground( Color.WHITE );
15
16       // nova cor de desenho configurada utiliza inteiros
17       g.setColor( new Color( 255, 0, 0 ) );
18       g.fillRect( 15, 25, 100, 20 );
19       g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
20
21       // nova cor de desenho configurada utiliza floats
22       g.setColor( new Color( 0.50f, 0.75f, 0.0f ) );
23       g.fillRect( 15, 50, 100, 20 );
24       g.drawString( "Current RGB: " + g.getColor(), 130, 65 );
```

```
25
26          // nova cor de desenho configurada usa objetos Color estáticos
27          g.setColor( Color.BLUE );
28          g.fillRect( 15, 75, 100, 20 );
29          g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
30
31          // exibe valores individuais de RGB
32          Color color = Color.MAGENTA;
33          g.setColor( color );
34          g.fillRect( 15, 100, 100, 20 );
35          g.drawString( "RGB values: " + color.getRed() + ", " +
36             color.getGreen() + ", " + color.getBlue(), 130, 115 );
37       } // fim do método paintComponent
38    } // fim da classe ColorJPanel
```

Figura 15.5 | Exemplo de alteração de `Color` em um desenho.

```
1  // Figura 15.6: ShowColors.java
2  // Demonstrando Colors.
3  import javax.swing.JFrame;
4
5  public class ShowColors
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       // cria o frame para ColorJPanel
11       JFrame frame = new JFrame( "Using colors" );
12       frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14       ColorJPanel colorJPanel = new ColorJPanel(); // cria ColorJPanel
15       frame.add( colorJPanel ); // adiciona colorJPanel ao frame
16       frame.setSize( 400, 180 ); // configura o tamanho do frame
17       frame.setVisible( true ); // exibe o frame
18    } // fim de main
19 } // fim da classe ShowColors
```

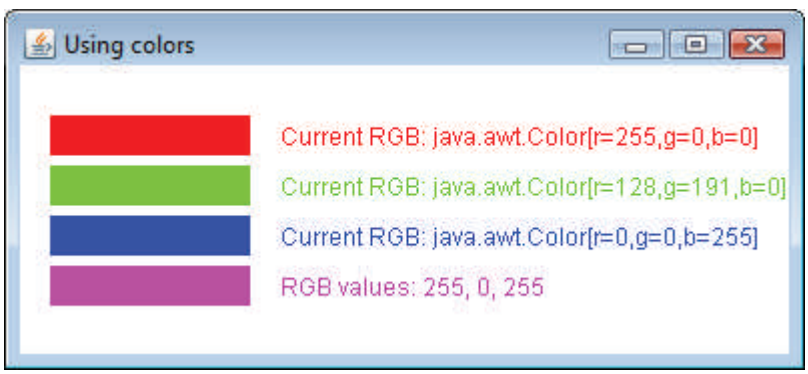


Figura 15.6 | Criando um `JFrame` para exibir cores no `JPanel`.

A linha 19 utiliza o método `Graphics` **`drawString`** para desenhar uma `String` com a cor atual. A expressão `g.getColor()` recupera a cor atual do objeto `Graphics`. Então, concatenamos a `Color` com a string `"Current RGB: "`, resultando em uma chamada implícita ao método `toString` da classe `Color`. A representação `String` de um `Color` contém o nome da classe e pacote (`java.awt.Color`) e os valores de vermelho, verde e azul.

Observações sobre a aparência e comportamento 15.1

Cada um percebe as cores de uma maneira diferente. Escolha suas cores cuidadosamente para assegurar que seu aplicativo é legível, tanto para pessoas que podem perceber as cores como para aquelas que são daltônicas. Tente evitar utilizar várias cores diferentes com valores muito próximos.

As linhas 22–24 e 27–29 realizam as mesmas tarefas novamente. A linha 22 utiliza o construtor `Color` com três argumentos `float` para criar uma cor verde escura (`0,50f` para vermelho, `0,75f` para verde e `0,0f` para azul). Observe a sintaxe dos valores. A letra `f` acrescentada a um literal de ponto flutuante indica que o literal deve ser tratado como tipo `float`. Lembre-se de que, por padrão, literais de ponto flutuante são tratados como um tipo `double`.

A linha 27 configura a cor atual do desenho como uma das constantes `Color` (`Color.BLUE`) pré-declaradas. As constantes `Color` são `static`, assim elas são criadas quando a classe `Color` é carregada na memória em tempo de execução.

A instrução nas linhas 35–36 faz chamadas aos métodos `Color` `getRed`, `getGreen` e `getBlue` na constante `Color.MAGENTA` pré-declarada. O método `main` da classe `ShowColors` (linhas 8–18 da Figura 15.6) cria a `JFrame` que conterá um objeto `ColorJPanel` onde as cores serão exibidas.

Observação de engenharia de software 15.1

Para alterar as cores, você deve criar um novo objeto `Color` (ou utilizar uma das constantes `Color` pré-declaradas). Como ocorre com objetos `String`, objetos `Color` são imutáveis (não modificáveis).

O pacote `javax.swing` fornece o componente GUI **`JColorChooser`** que permite aos usuários do aplicativo selecionar cores. O aplicativo das figuras 15.7 e 15.8 demonstra uma caixa de diálogo `JColorChooser`. Quando você clica no botão **Change Color**, um diálogo `JColorChooser` aparece. Quando você seleciona uma cor e pressiona o botão **OK** do diálogo, a cor de fundo da janela do aplicativo muda.

```
1  // Figura 15.7: ShowColors2JFrame.Java
2  // Escolhendo cores com JColorChooser.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JColorChooser;
10 import javax.swing.JPanel;
11
12 public class ShowColors2JFrame extends JFrame
13 {
14    private JButton changeColorJButton;
15    private Color color = Color.LIGHT_GRAY;
16    private JPanel colorJPanel;
17
18    // configura a GUI
19    public ShowColors2JFrame()
20    {
21       super( "Using JColorChooser" );
22
23       // cria JPanel para exibir as cores
24       colorJPanel = new JPanel();
25       colorJPanel.setBackground( color );
26
27       // configura changeColorJButton e registra seu handler de evento
28       changeColorJButton = new JButton( "Change Color" );
29       changeColorJButton.addActionListener(
30
31          new ActionListener() // classe interna anônima
32          {
33             // exibe JColorChooser quando o usuário clica no botão
34             public void actionPerformed( ActionEvent event )
35             {
36                color = JColorChooser.showDialog(
37                   ShowColors2JFrame.this, "Choose a color", color );
38
39                // configura a cor padrão, se nenhuma cor for retornada
40                if ( color == null )
41                   color = Color.LIGHT_GRAY;
42
43                // muda a cor de fundo do painel de conteúdo
44                colorJPanel.setBackground( color );
45             } // fim do método actionPerformed
46          } // fim da classe interna anônima
```

```
47          ); // fim da chamada para addActionListener
48
49          add( colorJPanel, BorderLayout.CENTER ); // adiciona colorJPanel
50          add( changeColorJButton, BorderLayout.SOUTH ); // adiciona botão
51
52          setSize( 400, 130 ); // configura o tamanho do frame
53          setVisible( true ); // exibe o frame
54       } // fim do construtor ShowColor2JFrame
55    } // fim da classe ShowColors2JFrame
```

Figura 15.7 | Caixa de diálogo `JColorChooser`.

```
1  // Figura 15.8: ShowColors2.java
2  // Escolhendo cores com JColorChooser.
3  import javax.swing.JFrame;
4
5  public class ShowColors2
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       ShowColors2JFrame application = new ShowColors2JFrame();
11       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
12    } // fim de main
13 } // fim da classe ShowColors2
```

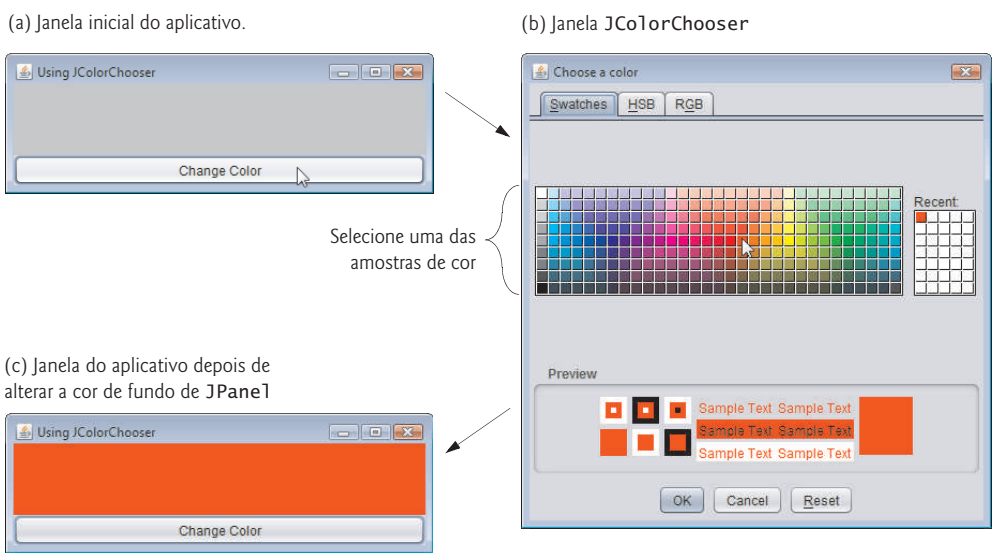


Figura 15.8 | Escolhendo cores com `JColorChooser`.

A classe `JColorChooser` fornece um método `static` **`showDialog`**, que cria um objeto `JColorChooser`, anexa-o a uma caixa de diálogo e exibe o diálogo. As linhas 36–37 da Figura 15.7 invocam esse método para exibir o diálogo do seletor de cores. O método `showDialog` retorna o objeto `Color` selecionado, ou `null` se o usuário pressionar **Cancel** ou fechar o diálogo sem pressionar **OK**. O método aceita três argumentos — uma referência ao seu `Component` pai, uma `String` para exibir na barra de título do diálogo e a `Color` inicial selecionada para o diálogo. O componente pai é uma referência à janela a partir da qual o diálogo é exibido (nesse caso `JFrame`, com o nome de referência `frame`). A caixa de diálogo será centralizada no pai. Se o pai for `null`, o diálogo é centralizado na tela. Enquanto o diálogo de seleção de cor está na tela, o usuário não pode interagir com o componente pai até que o diálogo seja liberado. Esse tipo de diálogo é chamado diálogo modal.

Depois de o usuário selecionar uma cor, as linhas 40–41 determinam se `color` é `null` e, nesse caso, configuram `color` como `Color.LIGHT_GRAY`. A linha 44 invoca o método `setBackground` para alterar a cor de fundo de `JPanel`. O método `setBackground` é um dos muitos métodos `Component` que podem ser utilizados na maioria dos componentes GUI. Observe que o usuário pode continuar a utilizar o botão **Change Color** para alterar a cor de fundo do aplicativo. A Figura 15.8 contém o método `main` que executa o programa.

A Figura 15.8(b) mostra a caixa de diálogo `JColorChooser` padrão que permite ao usuário selecionar uma cor a partir de várias **amostras de cores**. Observe que na verdade há três guias ao longo da parte superior da caixa de diálogo — Swatches, HSB e RGB. Esses representam três maneiras diferentes de selecionar uma cor. A guia HSB permite selecionar uma cor com base no **tom**, **saturação** e **brilho** — os valores utilizados para definir a quantidade de luz em uma cor. Não discutimos valores de HSB. Para informações adicionais sobre eles, visite `whatis.techtarget.com/definition/0,,sid9_gci212262,00.html`. A guia RGB permite selecionar uma cor utilizando controles deslizantes para selecionar os componentes vermelhos, verdes e azuis. As guias HSB e RGB são mostradas na Figura 15.9.

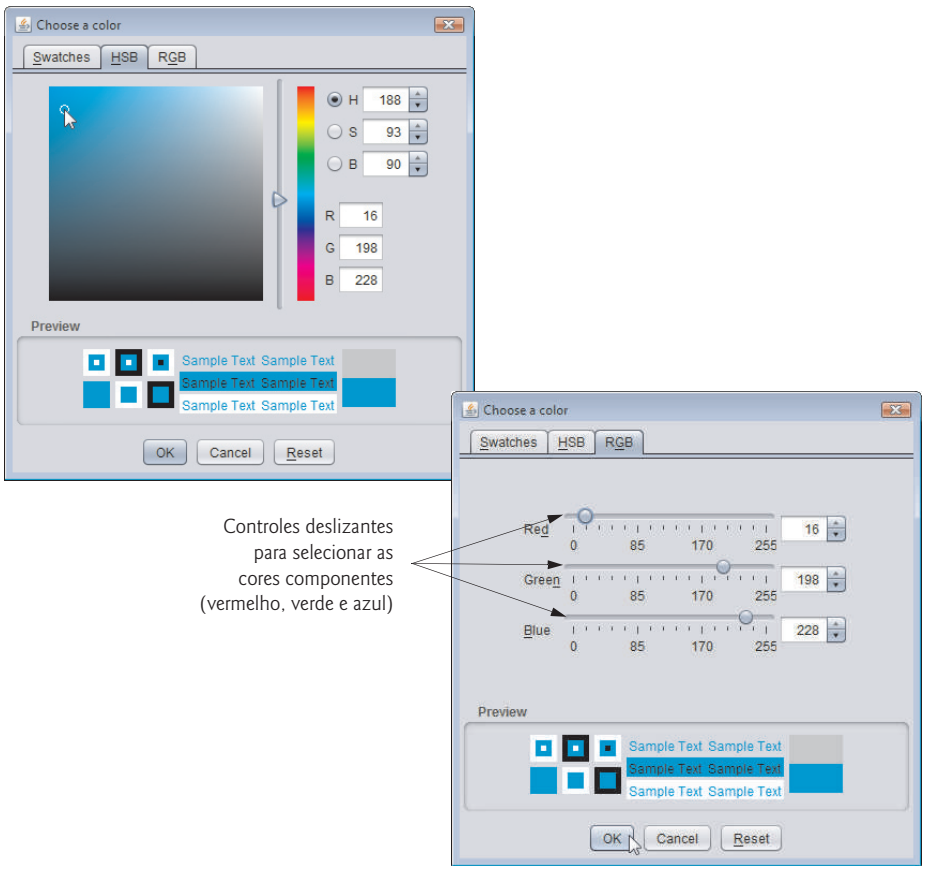


Figura 15.9 | As guias HSB e RGB da caixa de diálogo `JColorChooser`.

15.4 Manipulando fontes

Esta seção introduz métodos e constantes para manipular fontes. A maioria dos métodos de fonte e das constantes de fonte é parte da classe `Font`. Alguns métodos da classe `Font` e da classe `Graphics` estão resumidos na Figura 15.10.

Método ou constante	Descrição
Constantes, construtores e métodos de `Font`	
`public final static int PLAIN`	Uma constante representando um estilo de fonte simples.
`public final static int BOLD`	Uma constante representando um estilo de fonte negrito.
`public final static int ITALIC`	Uma constante representando um estilo de fonte itálico.
`public Font( String name, int style, int size )`	Cria um objeto `Font` com o nome, o estilo e o tamanho de fonte especificados.
`public int getStyle()`	Retorna um `int` indicando o estilo da fonte atual.
`public int getSize()`	Retorna um `int` indicando o tamanho da fonte atual.
`public String getName()`	Retorna o nome da fonte atual como uma string.
`public String getFamily()`	Retorna o nome da família de fontes como uma string.

Método ou constante	Descrição
`public boolean isPlain()`	Retorna `true` se a fonte for simples, caso contrário `false`.
`public boolean isBold()`	Retorna `true` se a fonte for negrito, caso contrário `false`.
`public boolean isItalic()`	Retorna `true` se a fonte for itálica, caso contrário `false`.
Métodos `Graphics` *para manipular* `Font`s	
`public Font getFont()`	Retorna uma referência de objeto `Font` representando a fonte atual.
`public void setFont( Font f )`	Configura a fonte atual como a fonte, o estilo e o tamanho especificados pela referência de objeto `Font` `f`.

Figura 15.10 | Métodos e constantes relacionados com `Font`.

O construtor da classe `Font` aceita três argumentos — **nome da fonte, estilo da fonte** e **tamanho da fonte**. O nome da fonte é qualquer fonte atualmente suportada pelo sistema em que o programa está executando, como as fontes padrão Java `Monospaced`, `SansSerif` e `Serif`. O estilo de fonte é **`Font.PLAIN`**, **`Font.ITALIC`** ou **`Font.BOLD`** (cada um é um campo `static` da classe `Font`). Os estilos de fonte podem ser utilizados em combinação (por exemplo, `Font.ITALIC + Font.BOLD`). O tamanho da fonte é medido em pontos. Um **ponto** é 1/72 de uma polegada. O método `Graphics` **`setFont`** configura a fonte atual de desenho — a fonte em que o texto será exibido — com seu argumento `Font`.

Dica de portabilidade 15.2

O número de fontes varia de um sistema para outro. O Java fornece cinco nomes de fonte — `Serif`, `Monospaced`, `SansSerif`, `Dialog` e `DialogInput` — que podem ser utilizados em todas as plataformas Java. O ambiente de tempo de execução Java (Java Runtime Environment — JRE) em cada plataforma mapeia esses nomes lógicos de fonte para fontes reais instaladas na plataforma. As fontes reais utilizadas podem variar entre plataformas.

O aplicativo das figuras 15.11 e 15.12 exibe texto em quatro fontes diferentes, com cada fonte em um tamanho diferente. A Figura 15.11 usa o construtor `Font` para inicializar objetos `Font` (nas linhas 16, 20, 24 e 29) que são passados para o método `Graphics` `setFont` a fim de alterar a fonte do desenho. Cada chamada ao construtor `Font` passa o nome de uma fonte (`Serif`, `Monospaced` ou `SansSerif`) como uma string, um estilo de fonte (`Font.PLAIN`, `Font.ITALIC` ou `Font.BOLD`) e um tamanho de fonte. Uma vez que o método `Graphics` `setFont` é invocado, todo texto exibido após a chamada aparecerá na nova fonte até que a fonte seja alterada. Informações de cada fonte são exibidas nas linhas 17, 21, 25, 30 e 31 com o método `drawString`. Note que a coordenada passada para `drawString` corresponde ao canto inferior esquerdo da linha de base da fonte. A linha 28 altera a cor do desenho para vermelho a fim de que a próxima string exibida apareça em vermelho. As linhas 30–31 exibem informações sobre o objeto `Font` final. O método **`getFont`** da classe `Graphics` retorna um objeto `Font` para representar a fonte atual. O método **`getName`** retorna o nome atual da fonte como uma string. O método **`getSize`** retorna o tamanho da fonte em pontos.

A Figura 15.12 contém o método `main`, que cria um `JFrame`. Adicionamos um objeto `FontJPanel` a esse `JFrame` (linha 15), que exibe as imagens gráficas criadas na Figura 15.11.

Observação de engenharia de software 15.2

Para alterar a fonte, você deve criar um novo objeto `Font`. Objetos `Font` são imutáveis — a classe `Font` não tem nenhum método set para alterar as características da fonte atual.

```
1  // Figura 15.11: FontJPanel.java
2  // exibe strings em diferentes fontes e cores.
3  import java.awt.Font;
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class FontJPanel extends JPanel
9  {
10    // exibe Strings em diferentes fontes e cores
11    public void paintComponent( Graphics g )
12    {
13       super.paintComponent( g ); // chama o paintComponent da superclasse
```

```
14
15          // fonte configurada como Serifa (Times), negrito, 12 pt e desenha uma string
16          g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
17          g.drawString( "Serif 12 point bold.", 20, 30 );
18
19          // configura font como Monospaced (Courier), itálico, 24 pt e desenha uma string
20          g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
21          g.drawString( "Monospaced 24 point italic.", 20, 50 );
22
23          // configura font como SansSerif (Helvetica), simples, 14 pt e desenha uma string
24          g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
25          g.drawString( "SansSerif 14 point plain.", 20, 70 );
26
27          // configura fonte como Serifa (Times), 18 pt negrito/itálico e desenha uma string
28          g.setColor( Color.RED );
29          g.setFont( new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
30          g.drawString(g.getFont().getName() + " " + g.getFont().getSize() +
31             " point bold italic.", 20, 90 );
32       } // fim do método paintComponent
33    } // fim da classe FontJPanel
```

Figura 15.11 | O método `Graphics` `setFont` altera a fonte de desenho.

```
1     // Figura 15.12: Fonts.java
2     // Utilizando fontes.
3     import javax.swing.JFrame;
4
5     public class Fonts
6     {
7        // executa o aplicativo
8        public static void main( String[] args )
9        {
10          // cria frame para FontJPanel
11          JFrame frame = new JFrame( "Using fonts" );
12          frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14          FontJPanel fontJPanel = new FontJPanel(); // cria FontJPanel
15          frame.add( fontJPanel ); // adiciona fontJPanel ao frame
16          frame.setSize( 420, 150 ); // configura o tamanho do frame
17          frame.setVisible( true ); // exibe o frame
18       } // fim de main
19    } // fim da classe Fonts
```

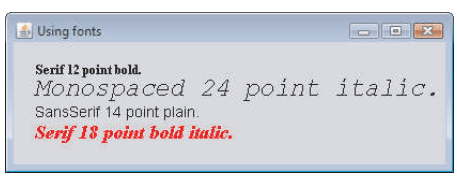


Figura 15.12 | Criando um `JFrame` para exibir fontes.

Métrica de fontes

Às vezes é necessário obter informações sobre a fonte atual do desenho, como seu nome, estilo e tamanho. Vários métodos `Font` utilizados para obter informações sobre fontes estão resumidos na Figura 15.10. O método **`getStyle`** retorna um valor inteiro que representa o estilo atual. O valor de inteiro retornado é `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` ou a combinação de `Font.ITALIC` e `Font.BOLD`. O método **`getFamily`** retorna o nome da família de fontes a qual a fonte atual pertence. O nome da família de fontes é específico à plataforma. Os métodos `Font` também estão disponíveis para testar o estilo da fonte atual, e estes também estão resumidos na Figura 15.10. Os métodos **`isPlain`**, **`isBold`** e **`isItalic`** retornam `true` se o estilo da fonte atual for simples, negrito ou itálico, respectivamente.

A Figura 15.13 ilustra alguns casos de **métrica de fonte** comum, que fornece informações precisas sobre uma fonte, como **altura**, **descendente** (quanto um caractere desce abaixo da linha de base), **ascendente** (quanto um caractere se eleva acima da linha de base) e **entrelinha** (a diferença entre a ascendente de uma linha de texto e a descendente da linha de texto embaixo dela — isto é, o espaçamento entre linhas).

Figura 15.13 | Medidas de fonte.

A classe **FontMetrics** declara vários métodos para obter a métrica de fonte. Esses métodos e o método `Graphics` **getFontMetrics** estão resumidos na Figura 15.14. O aplicativo das figuras 15.15 e 15.16 utiliza os métodos da Figura 15.14 para obter informações sobre a métrica da fonte para duas fontes.

Método	Descrição
Métodos `FontMetrics`	
`public int getAscent()`	Retorna a ascendente de uma fonte em pontos.
`public int getDescent()`	Retorna a descendente de uma fonte em pontos.
`public int getLeading()`	Retorna a entrelinha de uma fonte em pontos.
`public int getHeight()`	Retorna a altura de uma fonte em pontos.
Métodos `Graphics` *para obter* `FontMetrics` *de uma* `Font`	
`public FontMetrics getFontMetrics()`	Retorna o objeto `FontMetrics` para o argumento `Font` especificado.
`public FontMetrics getFontMetrics( Font f )`	Retorna o objeto `FontMetrics` para o argumento `Font` especificado.

Figura 15.14 | Os métodos `FontMetrics` e `Graphics` para obter medidas de fonte.

```
1  // Figura 15.15: MetricsJPanel.java
2  // Métodos FontMetrics e Graphics úteis para obter a métrica de fontes.
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class MetricsJPanel extends JPanel
9  {
10    // exibe a métrica de fontes.
11    public void paintComponent( Graphics g )
12    {
13       super.paintComponent( g ); // chama o paintComponent da superclasse
14
15       g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
16       FontMetrics metrics = g.getFontMetrics();
17       g.drawString( "Current font: " + g.getFont(), 10, 30 );
18       g.drawString( "Ascent: " + metrics.getAscent(), 10, 45 );
19       g.drawString( "Descent: " + metrics.getDescent(), 10, 60 );
20       g.drawString( "Height: " + metrics.getHeight(), 10, 75 );
21       g.drawString( "Leading: " + metrics.getLeading(), 10, 90 );
22
23       Font font = new Font( "Serif", Font.ITALIC, 14 );
24       metrics = g.getFontMetrics( font );
25       g.setFont( font );
26       g.drawString( "Current font: " + font, 10, 120 );
27       g.drawString( "Ascent: " + metrics.getAscent(), 10, 135 );
28       g.drawString( "Descent: " + metrics.getDescent(), 10, 150 );
29       g.drawString( "Height: " + metrics.getHeight(), 10, 165 );
30       g.drawString( "Leading: " + metrics.getLeading(), 10, 180 );
31    } // fim do método paintComponent
32 } // fim da classe MetricsJPanel
```

Figura 15.15 | Medidas de fonte.

```
1  // Figura 15.16: Metrics.java
2  // Exibindo a métrica de fonte.
3  import javax.swing.JFrame;
4
5  public class Metrics
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       // criar frame para MetricsJPanel
11       JFrame frame = new JFrame( "Demonstrating FontMetrics" );
12       frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14       MetricsJPanel metricsJPanel = new MetricsJPanel();
15       frame.add( metricsJPanel ); // adiciona metricsJPanel ao frame
16       frame.setSize( 510, 240 ); // configura o tamanho do frame
17       frame.setVisible( true ); // exibe o frame
18    } // fim de main
19 } // fim da classe Metrics
```

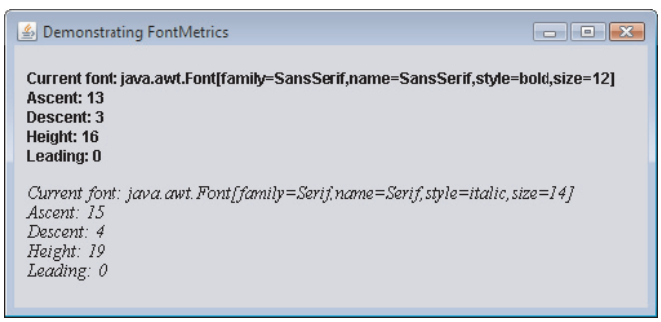


Figura 15.16 | Criando `JFrame` para exibir informações da métrica de fonte.

A linha 15 da Figura 15.15 cria e configura a fonte atual do desenho como uma fonte `SansSerif`, negrito, de 12 pontos. A linha 16 utiliza o método `Graphics` `getFontMetrics` para obter o objeto `FontMetrics` para a fonte atual. A linha 17 gera a saída da representação `String` da `Font` retornada por `g.getFont()`. As linhas 18–21 utilizam os métodos `FontMetric` para obter a ascendente, a descendente, a altura e a entrelinha da fonte.

A linha 23 cria uma nova fonte `Serif`, de 14 pontos em itálico. A linha 24 utiliza uma segunda versão do método `Graphics` `getFontMetrics`, que aceita um argumento `Font` e retorna um objeto `FontMetrics` correspondente. As linhas 27–30 obtêm a ascendente, descendente, altura e entrelinha da fonte. Observe que as medidas de fonte são ligeiramente diferentes para as duas fontes.

15.5 Desenhando linhas, retângulos e ovais

Esta seção apresenta os métodos `Graphics` para desenhar linhas, retângulos e ovais. Os métodos e seus parâmetros estão resumidos na Figura 15.17. Para cada método de desenho que requer um parâmetro `width` e `height`, o `width` e `height` devem ser valores não negativos. Caso contrário, a forma não será exibida.

Método	Descrição
`public void drawLine( int x1, int y1, int x2, int y2 )`	Desenha uma linha entre o ponto (`x1`, `y1`) e o ponto (`x2`, `y2`).
`public void drawRect( int x, int y, int width, int height )`	Desenha um retângulo com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo está localizado em (`x`, `y`). Somente o contorno do retângulo é desenhado utilizando a cor do objeto de `Graphics` — o corpo do retângulo não é preenchido com essa cor.
`public void fillRect( int x, int y, int width, int height )`	Desenha um retângulo preenchido na cor atual com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo está localizado em (`x`, `y`).

Método	Descrição
`public void clearRect( int x, int y, int width, int height )`	Desenha um retângulo preenchido com a `largura` e a `altura` especificadas na cor de fundo atual. O canto superior esquerdo do retângulo está localizado em (x, y). Esse método é útil se você quiser remover uma parte de uma imagem.
`public void drawRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight )`	Desenha um retângulo com cantos arredondados na cor atual com a `largura` e a `altura` especificadas. A `arcWidth` e a `arcHeight` determinam o arredondamento dos cantos (veja Figura 15.20). Somente o contorno da forma é desenhado.
`public void fillRoundRect( int x, int y, int width, int height, int arcWidth, int arcHeight )`	Desenha um retângulo preenchido na cor atual com cantos arredondados com a `largura` e a `altura` especificadas. A `arcWidth` e a `arcHeight` determinam o arredondamento dos cantos (veja Figura 15.20).
`public void draw3DRect( int x, int y, int width, int height, boolean b )`	Desenha um retângulo tridimensional na cor atual com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo está localizado em (x, y). O retângulo parece em alto relevo quando `b` é verdadeiro e em baixo relevo quando `b` é falso. Somente o contorno da forma é desenhado.
`public void fill3DRect( int x, int y, int width, int height, boolean b )`	Desenha um retângulo tridimensional preenchido na cor atual com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo está localizado em (x, y). O retângulo parece em alto relevo quando `b` é verdadeiro e em baixo relevo quando `b` é falso.
`public void drawOval( int x, int y, int width, int height )`	Desenha uma oval na cor atual com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo delimitado está localizado em (x, y). A oval toca todos os quatro lados do retângulo associado no centro de cada lado (veja Figura 15.21). Somente o contorno da forma é desenhado.
`public void fillOval( int x, int y, int width, int height )`	Desenha uma oval preenchida na cor atual com a `largura` e a `altura` especificadas. O canto superior esquerdo do retângulo delimitado está localizado em (x, y). A oval toca o centro dos quatro lados do retângulo delimitador (ver Figura 15.21).

Figura 15.17 | Métodos `Graphics` que desenham linhas, retângulos e ovais.

O aplicativo das figuras 15.18 e 15.19 demonstra o desenho de diversas linhas, retângulos, retângulos tridimensionais, retângulos arredondados e ovais. Na Figura 15.18, a linha 17 desenha uma linha vermelha, a linha 20 desenha um retângulo azul vazio e a linha 21 desenha um retângulo preenchido com azul. Os métodos **`fillRoundRect`** (linha 24) e **`drawRoundRect`** (linha 25) desenham retângulos com cantos arredondados. Seus primeiros dois argumentos especificam as coordenadas do canto superior esquerdo do **retângulo delimitador** — a área em que o retângulo arredondado será desenhado. Observe que as coordenadas do canto superior esquerdo não são a borda do retângulo arredondado, mas as coordenadas em que a borda estaria se o retângulo tivesse cantos quadrados. O terceiro e quarto argumentos especificam a largura e a altura do retângulo. Os dois últimos argumentos determinam os diâmetros horizontais e verticais do arco (isto é, a largura do arco e a altura do arco) utilizados para representar os cantos.

```
1  // Figura 15.18: LinesRectsOvalsJPanel.java
2  // Desenhando linhas, retângulos e ovais.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class LinesRectsOvalsJPanel extends JPanel
8  {
9     // exibe várias linhas, retângulos e elipses
10    public void paintComponent( Graphics g )
```

```
11      {
12         super.paintComponent( g ); // chama o método paint da superclasse
13
14         this.setBackground( Color.WHITE );
15
16         g.setColor( Color.RED );
17         g.drawLine( 5, 30, 380, 30 );
18
19         g.setColor( Color.BLUE );
20         g.drawRect( 5, 40, 90, 55 );
21         g.fillRect( 100, 40, 90, 55 );
22
23         g.setColor( Color.CYAN );
24         g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
25         g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
26
27         g.setColor( Color.GREEN );
28         g.draw3DRect( 5, 100, 90, 55, true );
29         g.fill3DRect( 100, 100, 90, 55, false );
30
31         g.setColor( Color.MAGENTA );
32         g.drawOval( 195, 100, 90, 55 );
33         g.fillOval( 290, 100, 90, 55 );
34      } // fim do método paintComponent
35   } // fim da classe LinesRectsOvalsJPanel
```

Figura 15.18 | Desenhando linhas, retângulos e ovais.

```
1    // Figura 15.19: LinesRectsOvals.java
2    // Desenhando linhas, retângulos e ovais.
3    import java.awt.Color;
4    import javax.swing.JFrame;
5
6    public class LinesRectsOvals
7    {
8       // executa o aplicativo
9       public static void main( String[] args )
10      {
11         // criar frame para LinesRectsOvalsJPanel
12         JFrame frame =
13            new JFrame( "Drawing lines, rectangles and ovals" );
14         frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
15
16         LinesRectsOvalsJPanel linesRectsOvalsJPanel =
17            new LinesRectsOvalsJPanel();
18         linesRectsOvalsJPanel.setBackground( Color.WHITE );
19         frame.add( linesRectsOvalsJPanel ); // adiciona painel ao frame
20         frame.setSize( 400, 210 ); // configura o tamanho do frame
21         frame.setVisible( true ); // exibe o frame
22      } // fim de main
23   } // fim da classe LinesRectsOvals
```

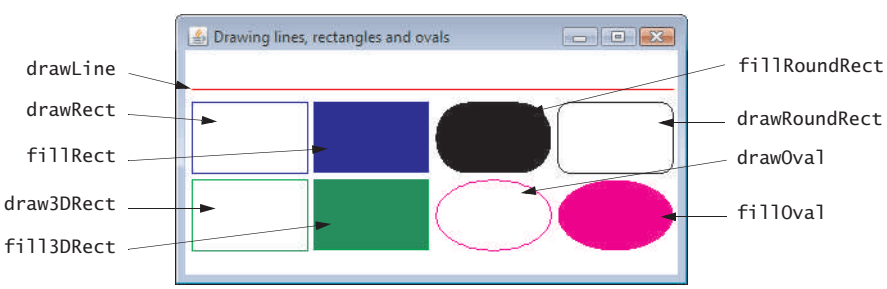


Figura 15.19 | Criando `JFrame` para exibir linhas, retângulos e ovais.

A Figura 15.20 rotula a largura e a altura de um arco e a largura e a altura de um retângulo arredondado. Utilizar o mesmo valor para a largura e a altura do arco produz o quarto de um círculo em cada um dos cantos. Se a largura do arco, altura do arco, largura e altura tiverem os mesmos valores, o resultado será um círculo. Se os valores para `width` (largura) e `height` (altura) forem os mesmos e os valores de `arcWidth` e `arcHeight` forem 0, o resultado será um quadrado.

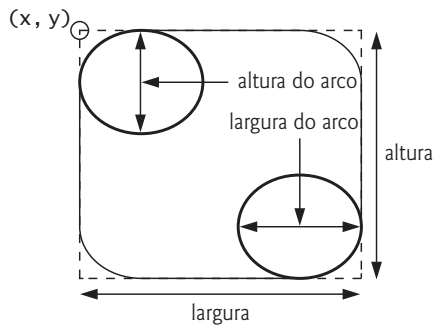


Figura 15.20 | A largura do arco e altura do arco para retângulos arredondados.

Os métodos **`draw3DRect`** (linha 28) e **`fill3DRect`** (linha 29) aceitam os mesmos argumentos. Os dois primeiros especificam o canto superior esquerdo do retângulo. Os próximos dois argumentos especificam a largura e altura do retângulo, respectivamente. O último argumento determina se o retângulo está em **alto relevo** (`true`) ou **baixo relevo** (`false`). O efeito tridimensional de `draw3DRect` aparece como duas bordas do retângulo na cor original e duas bordas em uma cor ligeiramente mais escura. O efeito tridimensional de `fill3DRect` aparece como duas bordas do retângulo na cor de desenho original e o preenche e as outras duas bordas em uma cor ligeiramente mais escura. Retângulos em alto relevo têm a cor de desenho original nas bordas superior e esquerda. Retângulos em baixo relevo têm a cor de desenho original nas bordas inferior e direita. O efeito tridimensional é difícil de ver em algumas cores.

Os métodos **`drawOval`** e **`fillOval`** (Figura 15.18, linhas 32–33) recebem os mesmos quatro argumentos. Os primeiros dois argumentos especificam a coordenada do canto superior esquerdo do retângulo delimitador que contém a oval. Os dois últimos argumentos especificam a largura e altura do retângulo delimitador, respectivamente. A Figura 15.21 mostra uma oval delimitada por um retângulo. Observe que a oval toca o centro de todos os quatro lados do retângulo delimitador. (O retângulo delimitador não é exibido na tela.)

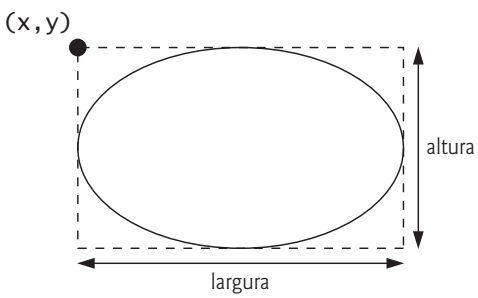


Figura 15.21 | Oval unida por um retângulo.

15.6 Desenhando arcos

Um **arco** é desenhado como uma parte de uma oval. Uma oval delimitada por um retângulo. Os arcos **varrem** (isto é, se movem ao longo de uma curva) a partir de um **ângulo inicial** até o número de graus especificado pelos seus **ângulos de arco**. Os ângulos de arco são medidos em graus. Os arcos *varrem* a partir de um *ângulo inicial* o número de graus especificado por seu *ângulo de arco*. O ângulo inicial indica em graus onde o arco começa. O conjunto esquerdo de eixos mostra uma varredura de arco de zero grau a aproximadamente 110 graus. Os arcos que varrem no sentido anti-horário são medidos em **graus positivos**. O conjunto de eixos à direita mostra um arco que varre de zero grau a aproximadamente –110 graus. Os arcos que varrem em sentido horário são medidos em **graus negativos**. Observe as caixas tracejadas em torno dos arcos na Figura 15.22. Ao desenhar um arco, especificamos um retângulo delimitador para uma oval. O arco varrerá parte da oval. Os métodos `Graphics` **`drawArc`** e **`fillArc`** para desenhar arcos são resumidos na Figura 15.23.

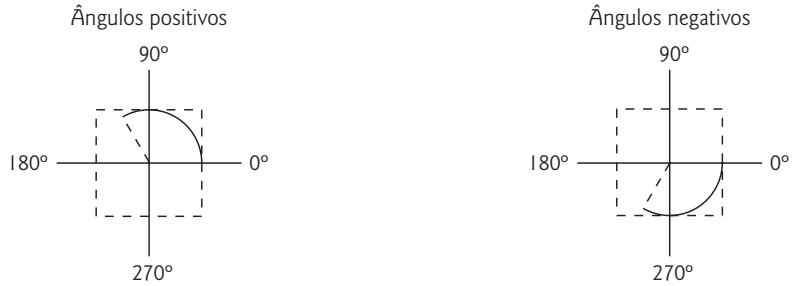


Figura 15.22 | Ângulos de arco positivo e negativo.

Método	Descrição
`public void drawArc( int x, int y, int width, int height, int startAngle, int arcAngle )`	Desenha um arco em relação ao canto superior esquerdo do retângulo delimitador e coordenadas `x` e `y` com a `largura` e `altura` especificadas. Métodos `Graphics` para desenhar arcos.
`public void fillArc( int x, int y, int width, int height, int startAngle, int arcAngle )`	Desenha um arco preenchido (isto é, um setor) em relação às coordenadas `x` e `y` do canto superior esquerdo do retângulo delimitador com a `largura` e `altura` especificadas. Métodos `Graphics` para desenhar arcos.

Figura 15.23 | Métodos `Graphics` para desenhar arcos.

As figuras 15.24 e 12.25 demonstram os métodos arc da Figura 15.23. O aplicativo desenha seis arcos (três não preenchidos e três preenchidos). Para ilustrar o retângulo delimitador que ajuda a determinar onde o arco aparece, os primeiros três arcos são exibidos dentro de um retângulo amarelo que tem os mesmos argumentos `x`, `y`, `largura` e `altura` que os arcos.

```
1  // Figura 15.24: ArcsJPanel.java
2  // Desenhando arcos.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class ArcsJPanel extends JPanel
8  {
9     // desenha retângulos e arcos
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g ); // chama o paintComponent da superclasse
13
14       // inicia em 0 e varre 360 graus
15       g.setColor( Color.RED );
16       g.drawRect( 15, 35, 80, 80 );
17       g.setColor( Color.BLACK );
18       g.drawArc( 15, 35, 80, 80, 0, 360 );
19
20       // inicia em 0 e varre 110 graus
21       g.setColor( Color.RED );
22       g.drawRect( 100, 35, 80, 80 );
23       g.setColor( Color.BLACK );
24       g.drawArc( 100, 35, 80, 80, 0, 110 );
25
```

```
26          // inicia em 0 e varre -270 graus
27          g.setColor( Color.RED );
28          g.drawRect( 185, 35, 80, 80 );
29          g.setColor( Color.BLACK );
30          g.drawArc( 185, 35, 80, 80, 0, -270 );
31
32          // inicia em 0 e varre 360 graus
33          g.fillArc( 15, 120, 80, 40, 0, 360 );
34
35          // inicia em 270 e varre -90 graus
36          g.fillArc( 100, 120, 80, 40, 270, -90 );
37
38          // inicia em 0 e varre -270 graus
39          g.fillArc( 185, 120, 80, 40, 0, -270 );
40       } // fim do método paintComponent
41    } // fim da classe ArcsJPanel
```

Figura 15.24 | Arcos exibidos com `drawArc` e `fillArc`.

```
1  // Figura 15.25: DrawArcs.java
2  // Desenhando arcos.
3  import javax.swing.JFrame;
4
5  public class DrawArcs
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       // cria frame para ArcsJPanel
11       JFrame frame = new JFrame( "Drawing Arcs" );
12       frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14       ArcsJPanel arcsJPanel = new ArcsJPanel(); // criar ArcsJPanel
15       frame.add( arcsJPanel ); // adiciona arcsJPanel ao frame
16       frame.setSize( 300, 210 ); // configura o tamanho do frame
17       frame.setVisible( true ); // exibe o frame
18    } // fim de main
19 } // fim da classe DrawArcs
```

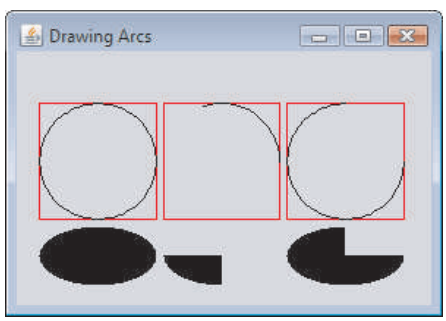


Figura 15.25 | Criando `JFrame` para exibir arcos.

15.7 Desenhando polígonos e polilinhas

Polígonos são formas de múltiplos lados compostas de segmentos de linhas retas. **Polilinhas** são sequências de pontos conectados. A Figura 15.26 discute métodos para desenhar polígonos e polilinhas. Observe que alguns métodos requerem um objeto **`Polygon`** (pacote `java.awt`). Os construtores da classe `Polygon` também são descritos na Figura 15.26. O aplicativo das figuras 15.27 e 15.28 desenha polígonos e polilinhas.

Método	Descrição
Métodos `Graphics` *para desenhar polígonos*	
`public void drawPolygon( int[] xPoints, int[] yPoints, int points )`	Desenha um polígono. A coordenada *x* de cada ponto é especificada no array `xPoints` e a coordenada *y* de cada ponto é especificada no array `yPoints`. O último argumento especifica o número de `points`. Esse método desenha um polígono. Se o último ponto for diferente do primeiro, o polígono é fechado por uma linha que conecta o último ponto ao primeiro.
`public void drawPolyline( int[] xPoints, int[] yPoints, int points )`	Desenha uma sequência de linhas conectadas. A coordenada *x* de cada ponto é especificada no array `xPoints` e a coordenada *y* de cada ponto é especificada no array `yPoints`. O último argumento especifica o número de `points`. Se o último ponto for diferente do primeiro, a polilinha não é fechada.
`public void drawPolygon( Polygon p )`	Desenha o polígono especificado.
`public void fillPolygon( int[] xPoints, int[] yPoints, int points )`	Desenha um polígono preenchido. A coordenada *x* de cada ponto é especificada no array `xPoints` e a coordenada *y* de cada ponto é especificada no array `yPoints`. O último argumento especifica o número de `points`. Esse método desenha um polígono. Se o último ponto for diferente do primeiro, o polígono é fechado por uma linha que conecta o último ponto ao primeiro.
`public void fillPolygon( Polygon p )`	Desenha o polígono preenchido especificado. O polígono é fechado.
Construtores e métodos `Polygon`	
`public Polygon()`	Constrói um novo objeto de polígono. O polígono não contém nenhum ponto.
`public void fillPolygon( int[] xPoints, int[] yPoints, int points )`	Constrói um novo objeto de polígono. O polígono tem `numberOfPoints` lados, com cada ponto consistindo em uma coordenada *x* de `xValues` e uma coordenada *y* de `yValues`.
`public void addPoint( int x, int y )`	Adiciona pares das coordenadas *x* e *y* ao `Polygon`.

Figura 15.26 | Métodos `Graphics` para polígonos e métodos `Polygon` da classe.

```
1  // Figura 15.27: PolygonsJPanel.java
2  // Desenhando polígonos.
3  import java.awt.Graphics;
4  import java.awt.Polygon;
5  import javax.swing.JPanel;
6
7  public class PolygonsJPanel extends JPanel
8  {
9     // desenha polígonos e polilinhas
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g ); // chama o paintComponent da superclasse
13
14       // desenha o polígono com objeto Polygon
15       int[] xValues = { 20, 40, 50, 30, 20, 15 };
16       int[] yValues = { 50, 50, 60, 80, 80, 60 };
17       Polygon polygon1 = new Polygon( xValues, yValues, 6 );
18       g.drawPolygon( polygon1 );
```

```
19
20          // desenha polilinhas com dois arrays
21          int[] xValues2 = { 70, 90, 100, 80, 70, 65, 60 };
22          int[] yValues2 = { 100, 100, 110, 110, 130, 110, 90 };
23          g.drawPolyline( xValues2, yValues2, 7 );
24
25          // preenche o polígono com dois arrays
26          int[] xValues3 = { 120, 140, 150, 190 };
27          int[] yValues3 = { 40, 70, 80, 60 };
28          g.fillPolygon( xValues3, yValues3, 4 );
29
30          // desenha o polígono preenchido com objeto Polygon
31          Polygon polygon2 = new Polygon();
32          polygon2.addPoint( 165, 135 );
33          polygon2.addPoint( 175, 150 );
34          polygon2.addPoint( 270, 200 );
35          polygon2.addPoint( 200, 220 );
36          polygon2.addPoint( 130, 180 );
37          g.fillPolygon( polygon2 );
38       } // fim do método paintComponent
39    } // fim da classe PolygonsJPanel
```

Figura 15.27 | Polígonos exibidos com `drawPolygon` e `fillPolygon`.

```
1  // Figura 15.28: DrawPolygons.java
2  // Desenhando polígonos.
3  import javax.swing.JFrame;
4
5  public class DrawPolygons
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       // cria frame para PolygonsJPanel
11       JFrame frame = new JFrame( "Drawing Polygons" );
12       frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14       PolygonsJPanel polygonsJPanel = new PolygonsJPanel();
15       frame.add( polygonsJPanel ); // adiciona polygonsJPanel ao frame
16       frame.setSize( 280, 270 ); // configura o tamanho do frame
17       frame.setVisible( true ); // exibe o frame
18    } // fim de main
19 } // fim da classe DrawPolygons
```

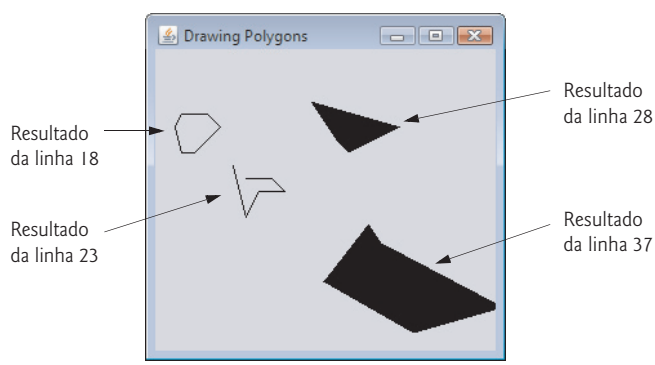


Figura 15.28 | Criando `JFrame` para exibir polígonos.

As linhas 15–16 da Figura 15.27 criam dois arrays `int` e os utilizam para especificar os pontos para `Polygon` `polygon1`. A chamada do construtor `Polygon` na linha 17 recebe o array `xValues`, que contém a coordenada *x* de cada ponto, o array `yValues`, que contém a coordenada *y* de cada ponto e 6 (o número de pontos no polígono). A linha 18 exibe `polygon1` passando-o como um argumento para o método `Graphics` **`drawPolygon`**.

As linhas 21–22 criam dois arrays `int` e os utilizam para especificar os pontos para uma série de linhas conectadas. O array `xValues2` contém a coordenada *x* de cada ponto e o array `yValues2`, a coordenada *y* de cada ponto. A linha 23 utiliza o método `Graphics` **`drawPolyline`** para exibir uma série de linhas conectadas especificadas com os argumentos `xValues2`, `yValues2` e `7` (o número de pontos).

As linhas 26–27 criam dois arrays `int` e os utilizam para especificar os pontos de um polígono. O array `xValues3` contém a coordenada *x* de cada ponto e o array `yValues3`, a coordenada *y* de cada ponto. A linha 28 exibe um polígono passando para o método `Graphics` **`fillPolygon`** os dois arrays (`xValues3` e `yValues3`) e o número de pontos a desenhar (`4`).

Erro comum de programação 15.1

Uma `ArrayIndexOutOfBoundsException` é lançada se o número de pontos especificado no terceiro argumento para o método `drawPolygon` ou o método `fillPolygon` for maior que o número de elementos nos arrays de coordenadas que especificam o polígono a exibir.

A linha 31 cria `Polygon polygon2` sem pontos. As linhas 32–36 utilizam o método `Polygon` addPoint para adicionar pares de coordenadas *x* e *y* a `Polygon`. A linha 37 exibe `Polygon polygon2` passando-o para o método `Graphics fillPolygon`.

15.8 Java 2D API

A **Java 2D API** fornece capacidades gráficas bidimensionais avançadas para programadores que requerem manipulações gráficas complexas e detalhadas. A API inclui recursos para processar arte a traço, texto e imagens nos pacotes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` e `java.awt.image.renderable`. As capacidades da API são muito amplas para abranger neste texto. Para uma visão geral dessas capacidades, consulte a demo Java 2D (discutida no Capítulo 23, "Applets e Java Web Start") ou visite `java.sun.com/javase/6/docs/technotes/guides/2d`. Nesta seção, apresentamos uma visão geral das várias capacidades de Java 2D.

Desenhar com a Java 2D API é realizado com uma referência **`Graphics2D`** (pacote `java.awt`). A `Graphics2D` é uma subclasse abstrata da classe `Graphics`, portanto ela contém todas as capacidades gráficas demonstradas anteriormente neste capítulo. De fato, o objeto real utilizado para desenhar em cada método `paintComponent` é uma instância de uma subclasse de `Graphics2D` que é passada para o método `paintComponent` e acessada via superclasse `Graphics`. Para acessar as capacidades de `Graphics2D`, devemos fazer a coerção da referência `Graphics` (`g`) passada a `paintComponent` para uma referência `Graphics2D` com uma instrução como

```
Graphics2D g2d = ( Graphics2D ) g;
```

Os dois exemplos a seguir utilizam essa técnica.

Linhas, retângulos, retângulos arredondados, arcos e elipses

Esse exemplo demonstra as várias formas de Java 2D no pacote `java.awt.geom`, incluindo **`Line2D.Double`**, **`Rectangle2D.Double`**, **`RoundRectangle2D.Double`**, **`Arc2D.Double`** e **`Ellipse2D.Double`**. Observe a sintaxe do nome de cada classe. Cada classe representa uma forma com dimensões especificadas como valores `double`. Há uma versão separada de cada uma representada com valores `float` (por exemplo, **`Ellipse2D.Float`**). Em cada caso, `Double` é uma classe `public static` aninhada da classe especificada à esquerda do ponto (por exemplo, `Ellipse2D`). Para utilizar a classe `static` aninhada, simplesmente qualificamos seu nome com o nome de classe externa.

Nas figuras 15.29 e 15.30, desenhamos formas de Java 2D e modificamos suas características de desenho, como mudar a espessura de linha, preencher formas com padrões e desenhar linhas tracejadas. Essas são apenas algumas das muitas capacidades fornecidas pelo Java 2D.

```
1  // Figura 15.29: ShapesJPanel.java
2  // Demonstrando algumas formas 2D Java.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.BasicStroke;
6  import java.awt.GradientPaint;
7  import java.awt.TexturePaint;
8  import java.awt.Rectangle;
9  import java.awt.Graphics2D;
10 import java.awt.geom.Ellipse2D;
11 import java.awt.geom.Rectangle2D;
12 import java.awt.geom.RoundRectangle2D;
13 import java.awt.geom.Arc2D;
14 import java.awt.geom.Line2D;
15 import java.awt.image.BufferedImage;
16 import javax.swing.JPanel;
17
18 public class ShapesJPanel extends JPanel
19 {
```

```
20       // desenha formas com Java 2D API
21       public void paintComponent( Graphics g )
22       {
23          super.paintComponent( g ); // chama o paintComponent da superclasse
24
25          Graphics2D g2d = ( Graphics2D ) g; // faz uma coerção em g para Graphics2D
26
27          // desenha elipse 2D preenchida com um gradiente azul-amarelo
28          g2d.setPaint( new GradientPaint( 5, 30, Color.BLUE, 35, 100,
29             Color.YELLOW, true ) );
30          g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
31
32          // desenha retângulo 2D em vermelho
33          g2d.setPaint( Color.RED );
34          g2d.setStroke( new BasicStroke( 10.0f ) );
35          g2d.draw( new Rectangle2D.Double( 80, 30, 65, 100 ) );
36
37          // desenha retâng. arred. 2D com um fundo armazenado em buffer
38          BufferedImage buffImage = new BufferedImage( 10, 10,
39             BufferedImage.TYPE_INT_RGB );
40
41          // obtém Graphics2D de buffImage e desenha nela
42          Graphics2D gg = buffImage.createGraphics();
43          gg.setColor( Color.YELLOW ); // desenha em amarelo
44          gg.fillRect( 0, 0, 10, 10 ); // desenha um retângulo preenchido
45          gg.setColor( Color.BLACK );  // desenha em preto
46          gg.drawRect( 1, 1, 6, 6 ); // desenha um retângulo
47          gg.setColor( Color.BLUE ); // desenha em azul
48          gg.fillRect( 1, 1, 3, 3 ); // desenha um retângulo preenchido
49          gg.setColor( Color.RED ); // desenha em vermelho
50          gg.fillRect( 4, 4, 3, 3 ); // desenha um retângulo preenchido
51
52          // pinta buffImage sobre o JFrame
53          g2d.setPaint( new TexturePaint( buffImage,
54             new Rectangle( 10, 10 ) ) );
55          g2d.fill(
56             new RoundRectangle2D.Double( 155, 30, 75, 100, 50, 50 ) );
57
58          // Desenha arco 2D em forma de torta em branco
59          g2d.setPaint( Color.WHITE );
60          g2d.setStroke( new BasicStroke( 6.0f ) );
61          g2d.draw(
62             new Arc2D.Double( 240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
63
64          // Desenha linhas 2D em verde e amarelo
65          g2d.setPaint( Color.GREEN );
66          g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
67
68          // desenha uma linha em 2D utilizando traço
69          float[] dashes = { 10 }; // especifica padrão de traço
70          g2d.setPaint( Color.YELLOW );
71          g2d.setStroke( new BasicStroke( 4, BasicStroke.CAP_ROUND,
72             BasicStroke.JOIN_ROUND, 10, dashes, 0 ) );
73          g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
74       } // fim do método paintComponent
75    } // fim da classe ShapesJPanel
```

Figura 15.29 | Formas 2D Java.

```
1    // Figura 15.30: Shapes.java
2    // Demonstrando algumas formas 2D Java.
3    import javax.swing.JFrame;
4
5    public class Shapes
6    {
7       // executa o aplicativo
```

```
8       public static void main( String[] args )
9       {
10         // cria frame para ShapesJPanel
11         JFrame frame = new JFrame( "Drawing 2D shapes" );
12         frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13
14         // cria ShapesJPanel
15         ShapesJPanel shapesJPanel = new ShapesJPanel();
16
17         frame.add( shapesJPanel ); // adiciona shapesJPanel ao frame
18         frame.setSize( 425, 200 ); // configura o tamanho do frame
19         frame.setVisible( true ); // exibe o frame
20      } // fim de main
21   } // fim da classe Shapes
```

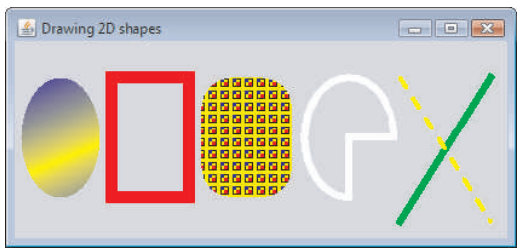


Figura 15.30 | Criando `JFrame` para exibir formas.

A linha 25 da Figura 15.29 faz coerção da referência `Graphics` recebida pelo `paintComponent` para uma referência `Graphics2D` e a atribui a `g2d` a fim de permitir acesso aos recursos do Java 2D.

Ovais e preenchimentos em degradê e objetos `Paint`

A primeira forma que desenhamos é uma oval preenchida com cores que mudam gradualmente. As linhas 28–29 invocam o método `Graphics2D` **`setPaint`** para configurar o objeto **`Paint`** que determina as cores exibidas pela forma. Um objeto `Paint` implementa a interface `java.awt.Paint`. Ele pode ser algo tão simples quanto um dos objetos `Color` pré-declarados introduzidos na Seção 15.3 (a classe `Color` implementa `Paint`) ou pode ser uma instância das classes `GradientPaint`, **`SystemColor`**, `TexturePaint`, `LinearGradientPaint` ou `RadialGradientPaint` da Java 2D API. Nesse caso, utilizamos um objeto `GradientPaint`.

A classe `GradientPaint` ajuda a desenhar uma forma em cores que mudam gradualmente — o que é chamado **gradiente** (ou *degradê*). O construtor `GradientPaint` utilizado aqui requer sete argumentos. Os dois primeiros especificam a coordenada inicial do gradiente (ou degradê). O terceiro especifica a `Color` inicial do gradiente. O quarto e o quinto argumentos especificam a coordenada final para o gradiente. O sexto especifica a `Color` de término do gradiente. O último argumento especifica se o gradiente é **cíclico** (`true`) ou **acíclico** (`false`). Os dois conjuntos de coordenadas determinam a direção do gradiente. Como a segunda coordenada (35, 100) está abaixo e à direita da primeira coordenada (5, 30), o gradiente segue para baixo e para a direita em um ângulo. Como esse gradiente é cíclico (`true`), a cor inicia com azul, gradualmente torna-se amarelo, então gradualmente retorna para azul. Se o gradiente fosse acíclico, a transição de cor seria da primeira cor especificada (por exemplo, azul) para a segunda cor (por exemplo, amarelo).

A linha 30 utiliza o método `Graphics2D` **`fill`** para desenhar um objeto **`Shape`** preenchido — um objeto que implementa a interface `Shape` (pacote `java.awt`). Nesse caso, exibimos um objeto `Ellipse2D.Double`. O construtor `Ellipse2D.Double` recebe quatro argumentos que especificam o retângulo delimitador para a elipse exibir.

Retângulos, `Strokes`

Em seguida, desenhamos um retângulo vermelho com uma borda grossa. A linha 33 invoca `setPaint` para configurar o objeto `Paint` como `Color.RED`. A linha 34 utiliza o método `Graphics2D` **`setStroke`** para configurar as características da borda do retângulo (ou as linhas para qualquer outra forma). O método `setStroke` requer como seu argumento um objeto que implementa a interface **`Stroke`** (pacote `java.awt`). Nesse caso, utilizamos uma instância da classe `BasicStroke`. A classe `BasicStroke` fornece diversos construtores para especificar a largura da linha, como as linhas terminam (chamadas **terminações de linha**), como as linhas se juntam (chamadas **junções de linha**) e os atributos de traço da linha (se for uma linha tracejada). O construtor aqui especifica que a linha deve ter 10 pixels de largura.

A linha 35 utiliza o método `Graphics2D` **`draw`** para desenhar um objeto `Shape` — nesse caso, um `Rectangle2D.Double`. O construtor `Rectangle2D.Double` recebe argumentos que especificam a coordenada *x* superior esquerda, a coordenada *y* superior esquerda, a largura e a altura.

Retângulos arredondados, objetos `BufferedImage` e `TexturePaint`

Em seguida, desenhamos um retângulo arredondado preenchido com um padrão criado em um objeto **`BufferedImage`** (pacote `java.awt.image`). As linhas 38–39 criam o objeto `BufferedImage`. A classe `BufferedImage` pode ser utilizada para criar imagens coloridas e na escala de cinza. Essa `BufferedImage` particular tem 10 pixels de largura e 10 pixels de altura (como especificado pelos dois primeiros argumentos do construtor). O terceiro argumento **`BufferedImage.TYPE_INT_RGB`** indica que a imagem é armazenada em cores utilizando o esquema de cores RGB.

Para criar o padrão de preenchimento do retângulo arredondado, devemos primeiro desenhar no `BufferedImage`. A linha 42 cria um objeto `Graphics2D` (chamando o método **`createGraphics`** `BufferedImage`) que pode ser utilizado para desenhar no `BufferedImage`. As linhas 43–50 utilizam os métodos `setColor`, `fillRect` e `drawRect` (discutidos anteriormente neste capítulo) para criar o padrão.

As linhas 53–54 configuram o objeto `Paint` como um novo objeto `TexturePaint` (pacote `java.awt`). Um objeto `TexturePaint` utiliza a imagem armazenada na sua `BufferedImage` associada (o primeiro argumento de construtor) como a textura de preenchimento para uma forma preenchida. O segundo argumento especifica a área `Rectangle` da `BufferedImage` que será replicada à textura. Nesse caso, `Rectangle` tem o mesmo tamanho que a `BufferedImage`. Mas uma parte menor da `BufferedImage` pode ser utilizada.

As linhas 55–56 utilizam o método `Graphics2D` `fill` para desenhar um objeto `Shape` preenchido — nesse caso, um `RoundRectangle2D.Double`. O construtor da classe `RoundRectangle2D.Double` recebe seis argumentos que especificam as dimensões do retângulo e a largura e a altura de arco utilizadas para determinar o arredondamento dos cantos.

Arcos

Em seguida desenhamos um arco em forma de torta com uma linha branca espessa. A linha 59 configura o objeto `Paint` como `Color.WHITE`. A linha 60 configura o objeto `Stroke` de um novo `BasicStroke` como uma linha de 6 pixels de largura. As linhas 61–62 usam o método `Graphics2D` `draw` para desenhar um objeto `Shape` — nesse caso, um `Arc2D.Double`. Os primeiros quatro argumentos do construtor `Arc2D.Double` especificam a coordenada *x* superior esquerda, a coordenada *y* superior esquerda, a largura e a altura do retângulo delimitador para o arco. O quinto argumento especifica o ângulo inicial. O sexto argumento especifica o ângulo do arco. O último argumento especifica como o arco é fechado. A constante **`Arc2D.PIE`** indica que o arco é fechado desenhando duas linhas — uma linha a partir do ponto inicial do arco para o centro do retângulo delimitador e uma linha a partir do centro do retângulo delimitador para o ponto terminal. A classe `Arc2D` fornece duas outras constantes `static` para especificar como o arco é fechado. A constante **`Arc2D.CHORD`** desenha uma linha do ponto inicial ao ponto final. A constante **`Arc2D.OPEN`** especifica que o arco não deve ser fechado.

Linhas

Por fim, desenhamos duas linhas utilizando objetos **`Line2D`** — uma sólida e uma tracejada. A linha 65 configura o objeto `Paint` como `Color.GREEN`. A linha 66 utiliza o método `Graphics2D` `draw` para desenhar um objeto `Shape` — nesse caso, uma instância da classe `Line2D.Double`. Os argumentos do construtor `Line2D.Double` especificam as coordenadas iniciais e coordenadas finais da linha.

A linha 69 declara o array `float` de um elemento contendo o valor 10. Esse array descreve os traços na linha tracejada. Nesse caso, cada traço terá 10 pixels de comprimento. Para criar traços de comprimentos diferentes em um padrão, simplesmente forneça o comprimento de cada traço como um elemento no array. A linha 70 configura o objeto `Paint` como `Color.YELLOW`. As linhas 71–72 configuram o objeto `Stroke` como um novo `BasicStroke`. A linha terá 4 pixels de largura e terá terminações arredondadas (**`BasicStroke.CAP_ROUND`**). Se as linhas se juntam (como em um vértice de um retângulo), as linhas de junção serão arredondadas (**`BasicStroke.JOIN_ROUND`**). O argumento `dashes` especifica o comprimento do traço da linha. O último argumento indica o índice inicial no array `dashes` para o primeiro traço no padrão. A linha 73 então desenha uma linha com o atual `Stroke`.

Criando suas próprias formas com caminhos gerais

Em seguida, apresentaremos um **caminho geral** — uma forma construída a partir de linhas retas e curvas complexas. Um caminho geral é representado com um objeto da classe **`GeneralPath`** (pacote `java.awt.geom`). O aplicativo das figuras 15.31 e 15.32 demonstra como desenhar um caminho geral na forma de uma estrela de cinco pontas.

```
1  // Figura 15.31: Shapes2JPanel.java
2  // Demonstrando um caminho geral.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.geom.GeneralPath;
7  import java.util.Random;
8  import javax.swing.JPanel;
9
10 public class Shapes2JPanel extends JPanel
11 {
```

```
12      // desenha caminhos gerais
13      public void paintComponent( Graphics g )
14      {
15         super.paintComponent( g ); // chama o paintComponent da superclasse
16         Random random = new Random(); // obtém o gerador de números aleatórios
17
18         int[] xPoints = { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
19         int[] yPoints = { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
20
21         Graphics2D g2d = ( Graphics2D ) g;
22         GeneralPath star = new GeneralPath(); // cria o objeto GeneralPath
23
24         // configura a coordenada inicial do General Path
25         star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );
26
27         // cria a estrela -- isso não desenha a estrela
28         for ( int count = 1; count < xPoints.length; count++ )
29            star.lineTo( xPoints[ count ], yPoints[ count ] );
30
31         star.closePath(); // fecha a forma
32
33         g2d.translate( 150, 150 ); // traduz a origem para (150, 150)
34
35         // gira em torno da origem e desenha estrelas em cores aleatórias
36         for ( int count = 1; count <= 20; count++ )
37         {
38            g2d.rotate( Math.PI / 10.0 ); // rotaciona o sistema de coordenadas
39
40            // configura cores aleatórias
41            g2d.setColor( new Color( random.nextInt( 256 ),
42               random.nextInt( 256 ), random.nextInt( 256 ) ) );
43
44            g2d.fill( star ); // desenha estrela preenchida
45         } // for final
46      } // fim do método paintComponent
47   } // fim da classe Shapes2JPanel
```

Figura 15.31 | Caminhos gerais Java 2D.

```
1    // Figura 15.32: Shapes2.java
2    // Demonstrando um caminho geral.
3    import java.awt.Color;
4    import javax.swing.JFrame;
5
6    public class Shapes2
7    {
8       // executa o aplicativo
9       public static void main( String[] args )
10      {
11         // cria frame para Shapes2JPanel
12         JFrame frame = new JFrame( "Drawing 2D Shapes" );
13         frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
14
15         Shapes2JPanel shapes2JPanel = new Shapes2JPanel();
16         frame.add( shapes2JPanel ); // adiciona shapes2JPanel ao frame
17         frame.setBackground( Color.WHITE ); // configura a cor de fundo do frame
18         frame.setSize( 315, 330 ); // configura o tamanho do frame
19         frame.setVisible( true ); // exibe o frame
20      } // fim de main
21   } // fim da classe Shapes2
```

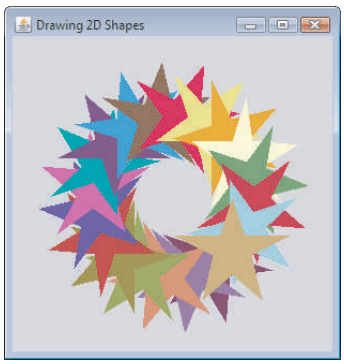


Figura 15.32 | Criando `JFrame` para exibir estrelas.

As linhas 18–19 declaram dois arrays `int` que representam as coordenadas *x* e *y* dos pontos na estrela. A linha 22 cria o objeto `GeneralPath star`. A linha 25 utiliza o método `GeneralPath` **`moveTo`** para especificar o primeiro ponto na `star`. A instrução `for` nas linhas 28–29 utilizam o método `GeneralPath` **`lineTo`** para desenhar uma linha para o próximo ponto em `star`. Cada nova chamada para `lineTo` desenha uma linha desde o ponto anterior até o ponto atual. A linha 31 utiliza o método `GeneralPath` **`closePath`** para desenhar uma linha desde o último ponto até o ponto especificado na última chamada a `moveTo`. Isso completa o caminho geral.

A linha 33 utiliza o método `Graphics2D` **`translate`** para mover a origem do desenho para a posição (150, 150). Todas as operações de desenho agora utilizam a posição (150, 150) como (0, 0).

A instrução `for` nas linhas 36–45 desenha a `star` 20 vezes rotacionando-a em torno do novo ponto de origem. A linha 38 utiliza o método `Graphics2D` **`rotate`** para rotacionar a próxima forma exibida. O argumento especifica o ângulo de rotação em radianos (com 360° = 2π radianos). A linha 44 utiliza o método `Graphics2D fill` para desenhar uma versão preenchida de `star`.

15.9 Conclusão

Neste capítulo, você aprendeu a utilizar as capacidades gráficas do Java para criar desenhos coloridos. Você aprendeu a especificar a posição de um objeto utilizando o sistema de coordenadas do Java e como desenhar em uma janela com o método `paintComponent`. Apresentamos a classe `Color` e você aprendeu como usar essa classe para especificar diferentes cores utilizando seus componentes RGB. Você utilizou o diálogo `JColorChooser` para permitir que usuários selecionem cores em um programa. Você então aprendeu a trabalhar com fontes ao desenhar texto em uma janela. Você aprendeu a criar um objeto `Font` a partir de um nome de fonte, estilo e tamanho e como acessar a métrica de uma fonte. Deste ponto, você aprendeu a desenhar várias formas em uma janela, como retângulos (regulares, arredondados e 3D), elipses e polígonos, bem como linhas e arcos. Você então utilizou a Java 2D API para criar formas mais complexas e para preenchê-las com gradientes ou padrões. Este capítulo concluiu com uma discussão sobre caminhos gerais utilizados para criar formas com linhas retas e curvas complexas. No próximo capítulo, discutiremos a classe `String` e seus métodos. Também introduzimos expressões regulares para correspondência de padrões em strings e demonstramos como validar a entrada de usuário com expressões regulares.

Resumo

Seção 15.1 Introdução

- O sistema de coordenadas do Java é um esquema para identificar cada ponto na tela.
- Um par de coordenadas tem uma coordenada *x* (horizontal) e uma coordenada *y* (vertical).
- As coordenadas são utilizadas para indicar onde as imagens gráficas devem ser exibidas em uma tela.
- Unidades coordenadas são medidas em pixels. Um pixel é a menor unidade de exibição de resolução do monitor.

Seção 15.2 Contextos gráficos e objetos gráficos

- Um contexto gráfico Java permite desenhar na tela.
- A classe `Graphics` contém métodos para desenhar strings, linhas, retângulos e outras formas. Os métodos também são incluídos para manipulação de fontes e manipulação de cores.
- Um objeto `Graphics` gerencia um contexto gráfico e desenha pixels na tela que representam texto e outros objetos gráficos (por exemplo, linhas, elipses, retângulos e outros polígonos).
- A classe `Graphics` é uma classe `abstract`. Cada implementação do Java tem uma subclasse `Graphics` que fornece capacidades de desenho. Essa implementação permanece oculta pela classe `Graphics`, que fornece a interface que permite utilizar imagens gráficas de uma maneira independente de plataforma.

- O método `paintComponent` pode ser utilizado para desenhar elementos gráficos em qualquer componente `JComponent`.
- O método `paintComponent` recebe um objeto `Graphics` que é passado para o método pelo sistema quando um componente Swing simples precisa ser repintado.
- Você raramente chama o método `paintComponent` diretamente, porque desenhar elementos gráficos é um processo baseado em eventos. Quando um aplicativo executa, o contêiner de aplicativo chama o método `paintComponent`. Para `paintComponent` ser chamado novamente, deve ocorrer um evento.
- Quando um `JComponent` é exibido, seu método `paintComponent` é chamado.
- Chamar o método `repaint` em um componente atualiza o elemento gráfico desenhado nesse componente.

Seção 15.3 Controle de cor

- A classe `Color` declara métodos e constantes para manipular cores em um programa Java.
- Cada cor é criada a partir de um componente vermelho, um verde e um azul. Juntos, esses componentes são chamados valores RGB. Os componentes RGB especificam a quantidade de vermelho, verde e azul em uma cor, respectivamente. Quanto maior o valor de RGB, maior a quantidade dessa cor particular.
- Os métodos `Color`, `getGreen` e `getBlue` retornam valores `int` de 0 a 255 representando a quantidade de vermelho, verde e azul, respectivamente.
- O método `Graphics` `getColor` retorna um objeto `Color` representando a cor atual do desenho.
- O método `Graphics` `setColor` configura a cor atual do desenho.
- O método `Graphics` `fillRect` desenha um retângulo preenchido pela cor atual do objeto `Graphics`.
- O método `Graphics` `drawString` desenha uma `String` na cor atual.
- O componente GUI de `JColorChooser` permite que os usuários do aplicativo selecionem cores.
- O método `static` `showDialog` `JColorChooser` cria um objeto `JColorChooser`, anexa-o a uma caixa de diálogo e exibe a caixa de diálogo. Quando ele está na tela, o usuário só pode interagir com a caixa de diálogo. Esse tipo de diálogo é chamado diálogo modal.

Seção 15.4 Manipulando fontes

- A classe `Font` contém métodos e constantes para manipular fontes.
- A construtor da classe `Font` aceita três argumentos — nome da fonte, estilo da fonte e tamanho da fonte.
- O estilo de fonte de uma `Font` pode ser `Font.PLAIN`, `Font.ITALIC` ou `Font.BOLD` (cada um é um campo `static` da classe `Font`). Os estilos de fonte podem ser utilizados em combinação (por exemplo, `Font.ITALIC + Font.BOLD`).
- O tamanho da fonte é medido em pontos. Um ponto tem 1/72 de uma polegada.
- O método `Graphics` `setFont` configura a fonte de desenho na qual o texto será exibido.
- O método `Font` `getStyle` retorna um valor de inteiro representando o estilo atual de `Font`.
- O método `Font` `getSize` retorna o tamanho de fonte em pontos.
- O método `Font` `getName` retorna o nome atual da fonte como uma string.
- O método `Font` `getFamily` retorna o nome da família de fontes a que a fonte atual pertence. O nome da família de fontes é específico da plataforma.
- A classe `FontMetrics` contém métodos para obter informações de fonte.
- Métricas de fonte incluem altura, descendente e entrelinha.

Seção 15.5 Desenhando linhas, retângulos e ovais

- Os métodos `Graphics` `fillRoundRect` e `drawRoundRect` desenham retângulos com cantos arredondados.
- Os métodos `Graphics` `draw3DRect` e `fill3DRect` desenham retângulos tridimensionais.
- Os métodos `Graphics` `drawOval` e `fillOval` desenham ovais.

Seção 15.6 Desenhando arcos

- Um arco é desenhado como uma parte de uma oval.
- Arcos estendem-se de um ângulo inicial pelo número de graus especificado pelo ângulo do arco.
- Os métodos `Graphics` `drawArc` e `fillArc` são utilizados para desenhar arcos.

Seção 15.7 Desenhando polígonos e polilinhas

- A classe `Polygon` contém métodos para criar polígonos.
- Polígonos são formas de múltiplos lados compostas de segmentos de linhas retas.
- As polilinhas são uma sequência de pontos conectados.
- O método `Graphics` `drawPolyline` exibe uma série de linhas conectadas.
- Os métodos `Graphics` `drawPolygon` e `fillPolygon` são utilizados para desenhar polígonos.
- O método `Polygon` `addPoint` da classe `Polygon` adiciona os pares das coordenadas *x* e *y* a `Polygon`.

Seção 15.8 Java 2D API

- A API 2D do Java fornece capacidades gráficas bidimensionais avançadas.
- A classe `Graphics2D` — subclasse de `Graphics` — é utilizada para desenhar com a Java 2D API.
- As classes da Java 2D API para desenhar formas incluem `Line2D.Double`, `Rectangle2D.Double`, `RoundRectangle2D.Double`, `Arc2D.Double` e `Ellipse2D.Double`.
- A classe `GradientPaint` ajuda a desenhar uma forma em cores que mudam gradualmente — o que é chamado gradiente (ou degradê).
- O método `Graphics2D` `fill` desenha um objeto preenchido de qualquer tipo que implementa a interface `Shape`.
- A classe `BasicStroke` ajuda a especificar as características do desenho de linhas.
- O método `Graphics2D` `draw` é utilizado para desenhar um objeto `Shape`.
- As classes `GradientPaint` e `TexturePaint` ajudam a especificar as características para preencher formas com cores ou padrões.
- Um caminho geral é uma forma construída de linhas retas e curvas complexas e é representado com um objeto de classe `GeneralPath`.
- O método `GeneralPath` `moveTo` especifica o primeiro ponto em um caminho geral.
- O método `GeneralPath` `lineTo` desenha uma linha para o próximo ponto no caminho. Cada nova chamada a `lineTo` desenha uma linha desde o ponto anterior até o ponto atual.
- O método `GeneralPath` `closePath` desenha uma linha do último ponto ao ponto especificado na última chamada a `moveTo`. Isso completa o caminho geral.
- O método `Graphics2D` `translate` é utilizado para mover a origem do desenho para uma nova posição.
- O método `Graphics2D` `rotate` é utilizado para rotacionar a próxima forma exibida.

Terminologia

`addPoint`, método da classe `Polygon`, 503
altura de uma fonte, 493
amostras de cor, 491
ângulo inicial de um arco, 498
`Arc2D.Double`, classe, 503
arco, 498
arco, ângulo, 498
ascendente de uma fonte, 493
`BasicStroke`, classe, 484
`BOLD`, constante da classe `Font`, 491
brilho, 491
`BufferedImage`, classe, 506
caminho geral, 506
`CAP_ROUND`, constante da classe `BasicStroke`, 506
`CHORD`, constante da classe `Arc2D`, 506
`closePath`, método da classe `GeneralPath`, 508
`Color`, classe, 484
contexto de imagens gráficas, 484
coordenada horizontal, 484
coordenada vertical, 484
`createGraphics`, método da classe `BufferedImage`, 506
descendente de uma fonte, 493
`draw`, método da classe `Graphics2D`, 505
`draw3DRect`, método da classe `Graphics`, 498
`drawArc`, método da classe `Graphics`, 498
`drawPolygon`, método da classe `Graphics`, 502
`drawPolyline`, método da classe `Graphics`, 503
`drawRoundRect`, método da classe `Graphics`, 496
`drawString`, método da classe `Graphics`, 488
eixo *x*, 484
eixo *y*, 484
`Ellipse2D.Double`, classe, 503
`Ellipse2D.Float`, classe, 503
entrelinha de uma fonte, 493
estilo de fonte, 492
`fill`, método da classe `Graphics2D`, 505
`fill3DRect`, método da classe `Graphics`, 498
`fillArc`, método da classe `Graphics`, 498
`fillOval`, método da classe `Graphics`, 498
`fillPolygon`, método da classe `Graphics`, 503
`fillRect`, método da classe `Graphics`, 487
`fillRoundRect`, método da classe `Graphics`, 496
`Font`, classe, 484
`FontMetrics`, classe, 484
`GeneralPath`, classe, 506
`getBlue`, método da classe `Color`, 487
`getColor`, método da classe `Color`, 487
`getFamily`, método da classe `Font`, 493
`getFontMetrics`, método da classe `FontMetrics`, 494
`getGreen`, método da classe `Color`, 487
`getName`, método da classe `Font`, 492
`getRed`, método da classe `Color`, 487
`getSize`, método da classe `Font`, 492
`getStyle`, método da classe `Font`, 493
gradiente, 505
gradiente acíclico, 505
gradiente cíclico, 505
`GradientPaint`, classe, 484
`Graphics`, classe, 484
`Graphics2D`, classe, 484
graus negativos, 498
graus positivos, 498
`isBold`, método da classe `Font`, 493
`isItalic`, método da classe `Font`, 493
`isPlain`, método da classe `Font`, 493
`ITALIC`, constante da classe `Font`, 491
Java2D API, 503
`JColorChooser`, classe, 489
`JOIN_ROUND`, constante da classe `BasicStroke`, 506
junções de linha, 505
`Line2D`, classe, 506
`Line2D.Double`, classe, 503
`lineTo`, método da classe `GeneralPath`, 508
maiúsculas de fim (de uma linha), 505
métrica de fonte, 493
`moveTo`, método da classe `GeneralPath`, 508
nome de fonte, 492
`OPEN`, constante da classe `Arc2D`, 506
`Paint`, objeto, 505
`PIE`, constante da classe `Arc2D`, 506
pixel (*picture element*), 484
`PLAIN`, constante da classe `Font`, 491
polígono, 500
polilinha, 500
`Polygon`, classe, 484
ponto, 492
`Rectangle2D.Double`, classe, 503
`repaint`, método da classe `JComponent`, 486
retângulo delimitador, 496
retângulo em alto relevo, 498
retângulo em baixo relevo, 498
`rotate`, método da classe `Graphics2D`, 508
`RoundRectangle2D.Double`, classe, 503
saturação, 491
`setColor`, método da classe `Color`, 487
`setColor`, método da classe `Graphics`, 487
`setFont`, método da classe `Graphics`, 492
`setPaint`, método da classe `Graphics2D`, 505
`setStroke`, método da classe `Graphics2D`, 505
`Shape`, objeto, 505
`showDialog`, método da classe `JColorChooser`, 490
sistema de coordenadas, 484
`Stroke`, objeto, 505
`SystemColor`, classe, 505

tamanho de fonte, 492
`TexturePaint`, classe, 484
thread, 486
thread de despacho do evento (*Event-Dispatch Thread* — EDT), 486
tom, 491
`translate`, método da classe `Graphics2D`, 508
`TYPE_INT_RGB`, constante da classe `BufferedImage`, 506
varredura de um arco, 498
varrer, 498
x, coordenada, 484
y, coordenada, 484

Exercícios de autorrevisão

15.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Em Java 2D, o método _____ da classe _____ configura as características de uma linha utilizada para desenhar uma forma.

b) A classe _____ ajuda a especificar o preenchimento para uma forma de tal modo que o preenchimento gradualmente muda de uma cor para outra.

c) O método _____ da classe `Graphics` desenha uma linha entre dois pontos.

d) O RGB é acrônimo para _____ , _____ e _____ .

e) Os tamanhos da fonte são medidos em unidades chamadas _____ .

f) A classe _____ ajuda a especificar o preenchimento para uma forma que utiliza um padrão desenhado em uma `BufferedImage` .

15.2 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Os primeiros dois argumentos do método `Graphics drawOval` especificam a coordenada do centro da oval.

b) No sistema de coordenadas do Java, as coordenadas *x* aumentam da esquerda para a direita e as coordenadas *y* de cima para baixo.

c) O método `Graphics fillPolygon` desenha um polígono preenchido na cor atual.

d) O método `Graphics drawArc` permite ângulos negativos.

e) O método `Graphics getSize` retorna o tamanho da fonte atual em centímetros.

f) A coordenada de pixel (0, 0) localiza-se no centro exato do monitor.

15.3 Localize o(s) erro(s) em cada um dos seguintes itens e explique como corrigi-lo(s). Assuma que `g` é um objeto `Graphics`.

a)
```
g.setFont( "SansSerif" );
```

b)
```
g.erase( x, y, w, h );     // remove o retângulo em (x, y)
```

c)
```
Font f = new Font( "Serif", Font.BOLDITALIC, 12 );
```

d)
```
g.setColor( 255, 255, 0 );  // muda a cor para amarelo
```

Respostas dos exercícios de autorrevisão

15.1 a) `setStroke`, `Graphics2D`. b) `GradientPaint`. c) `drawLine`. d) Red, Green, Blue. e) pontos. f) `TexturePaint`.

15.2 a) Falso. Os primeiros dois argumentos especificam o canto superior esquerdo do retângulo delimitador.

b) Verdadeiro.

c) Verdadeiro.

d) Verdadeiro.

e) Falso. Os tamanhos da fonte são medidos em pontos.

f) Falso. A coordenada (0,0) corresponde ao canto superior esquerdo de um componente GUI em que o desenho ocorre.

15.3 a) O método `setFont` recebe um objeto `Font` como um argumento — não uma `String`.

b) A classe `Graphics` não tem um método `erase`. O método `clearRect` deve ser utilizado.

c) `Font.BOLDITALIC` não é um estilo válido de fonte. Para obter uma fonte em negrito e itálico, utilize `Font.BOLD + Font.ITALIC`.

d) O método `setColor` recebe um objeto `Color` como um argumento, não três inteiros.

Exercícios

15.4 Preencha as lacunas em cada uma das seguintes afirmações:

a) A classe _____ da Java 2D API é utilizada para desenhar elipses.

b) Os métodos `draw` e `fill` da classe `Graphics2D` requerem um objeto do tipo _____ como seu argumento.

c) As três constantes que especificam o estilo de fonte são _____ , _____ e _____ .

d) O método `Graphics2D` _____ configura a cor de pintura para formas Java 2D.

15.5 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) O método `Graphics drawPolygon` conecta automaticamente os pontos finais do polígono.

b) O método `Graphics drawLine` desenha uma linha entre dois pontos.

c) O método `Graphics fillArc` utiliza graus para especificar o ângulo.

d) No sistema de coordenadas Java, os valores no eixo *y* aumentam da esquerda para a direita.

e) `Graphics` herda diretamente da classe `Object`.

f) `Graphics` é uma classe `abstract`.

g) A classe `Font` herda diretamente de classe `Graphics`.

15.6 **(*Círculos concêntricos utilizando o método `drawArc`*)** Escreva um aplicativo que desenha uma série de oito círculos concêntricos. Os círculos devem ser separados por 10 pixels. Utilize o método `Graphics drawArc`.

15.7 **(*Círculos concêntricos utilizando a classe `Ellipse2D.Double`*)** Modifique sua solução do Exercício 15.6 para desenhar as ovais utilizando a classe `Ellipse2D.Double` e o método `draw` da classe `Graphics2D`.

15.8 **(*Linhas aleatórias utilizando a classe `Line2D.Double`*)** Modifique sua solução do Exercício 15.7 para desenhar linhas aleatórias com cores aleatórias e espessuras aleatórias de linha. Utilize a classe `Line2D.Double` e o método `draw` da classe `Graphics2D` para desenhar as linhas.

15.9 **(*Triângulos aleatórios*)** Escreva um aplicativo que exiba triângulos gerados aleatoriamente em cores diferentes. Cada triângulo deve ser preenchido com uma cor diferente. Utilize a classe `GeneralPath` e o método `fill` da classe `Graphics2D` para desenhar os triângulos.

15.10 **(*Caracteres aleatórios*)** Escreva um aplicativo que desenhe aleatoriamente caracteres em diferentes fontes, tamanhos e cores.

15.11 **(*Grade utilizando o método `drawLine`*)** Escreva um aplicativo que desenhe uma grade de 8 por 8. Utilize o método `Graphics drawLine`.

15.12 **(*Grade utilizando a classe `Line2D.Double`*)** Modifique sua solução do Exercício 15.11 para desenhar a grade utilizando instâncias da classe `Line2D.Double` e o método `draw` da classe `Graphics2D`.

15.13 **(*Grade utilizando o método `drawRect`*)** Escreva um aplicativo que desenhe uma grade de 10 por 10. Utilize o método `Graphics drawRect`.

15.14 **(*Grade utilizando a classe `Rectangle2D.Double`*)** Modifique sua solução do Exercício 15.13 para desenhar a grade usando a classe `Rectangle2D.Double` e o método `draw` da classe `Graphics2D`.

15.15 **(*Desenhando tetraedros*)** Escreva um aplicativo que desenhe um tetraedro (uma pirâmide). Utilize a classe `GeneralPath` e o método `draw` da classe `Graphics2D`.

15.16 **(*Desenhando cubos*)** Escreva um aplicativo que desenhe um cubo. Utilize a classe `GeneralPath` e o método `draw` da classe `Graphics2D`.

15.17 **(*Círculos utilizando a classe `Ellipse2D.Double`*)** Escreva um aplicativo que solicite ao usuário para inserir o raio de um círculo como um número de ponto flutuante e desenhe o círculo, bem como os valores do diâmetro do círculo, circunferência e área. Utilize o valor 3,14159 para π. [*Nota:* você também pode utilizar a constante `Math.PI` predefinida para o valor de π. Essa constante é mais precisa que o valor 3,14159. A classe `Math` é declarada no pacote `java.lang`, então você não precisa `importá`-la.] Utilize as seguintes fórmulas (*r* é o raio):

diâmetro = $2r$
circunferência = $2\pi r$
área = πr^2

Também se deve solicitar ao usuário um conjunto de coordenadas além do raio. Então desenhe o círculo e exiba seu diâmetro, circunferência e área, utilizando um objeto `Ellipse2D.Double` para representar o círculo e o método `draw` da classe `Graphics2D` para exibi-lo.

15.18 **(*Protetor de tela*)** Escreva um aplicativo que simule um protetor de tela. O aplicativo deve desenhar linhas aleatoriamente utilizando o método `drawLine` da classe `Graphics`. Depois de desenhar 100 linhas, o aplicativo deve se autorredefinir e começar a desenhar as linhas novamente. Para permitir que o programa desenhe continuamente, coloque uma chamada a `repaint` como a última linha no método `paintComponent`. Você notou qualquer problema com isso em seu sistema?

15.19 **(*Protetor de tela utilizando `Timer`*)** O pacote `javax.swing` contém uma classe chamada `Timer` que é capaz de chamar o método `actionPerformed` da interface `ActionListener` em um intervalo fixo de tempo (especificado em milissegundos). Modifique sua solução do Exercício 15.18 para remover a chamada a `repaint` a partir do método `paintComponent`. Declare sua classe para implementar `ActionListener`. (O método `actionPerformed` deve simplesmente chamar `repaint`.) Declare uma variável de instância do tipo `Timer` chamada `timer` em sua classe. No construtor da sua classe, escreva as seguintes instruções:

```
timer = new Timer( 1000, this );
timer.start( );
```

Isso cria uma instância de classe `Timer` que chamará o método `actionPerformed` do objeto `this` a cada `1000` milissegundos (isto é, um segundo).

15.20 **(*Protetor de tela para um número aleatório de linhas*)** Modifique sua solução do Exercício 15.19 para permitir que o usuário insira o número de linhas aleatórias que deve ser desenhado antes de o aplicativo apagar seu próprio desenho e começar a desenhar linhas novamente. Utilize um `JTextField` para obter o valor. O usuário deve ser capaz de digitar um novo número no `JTextField` em qualquer momento durante a execução do programa. Utilize uma classe interna para realizar o tratamento de evento para o `JTextField`.

15.21 **(*Protetor de tela com formas*)** Modifique sua solução do Exercício 15.19 para que ele utilize a geração de números aleatórios a fim de escolher diferentes formas a exibir. Utilize os métodos da classe `Graphics`.

15.22 **(*Protetor de tela utilizando a Java 2D API*)** Modifique sua solução do Exercício 15.21 para utilizar as classes e as capacidades de desenho da Java 2D API. Desenhe formas como retângulos e elipses, com gradientes gerados aleatoriamente. Utilize a classe `GradientPaint` para gerar o gradiente.

15.23 (***Gráficos de tartaruga***) Modifique sua solução do Exercício 7.21 — *Gráficos de tartaruga* — para adicionar uma interface gráfica com o usuário utilizando `JTextFields` e `JButtons`. Desenhe linhas em vez de asteriscos (*). Quando o programa gráfico de tartaruga especificar um movimento, traduza o número de posições em um número de pixels na tela multiplicando o número de posições por 10 (ou qualquer valor que você escolher). Implemente o desenho com os recursos da Java 2D API.

15.24 (***Passeio do Cavalo***) Crie uma versão gráfica do problema do Passeio do Cavalo (exercícios 7.22, 7.23 e 7.26). À medida que cada movimento é feito, a célula apropriada do tabuleiro deve ser atualizada com o número adequado do movimento. Se o resultado do programa é um *tour completo* ou um *tour fechado*, o programa deve exibir uma mensagem apropriada. Se quiser, utilize a classe `Timer` (veja o Exercício 15.19) para ajudar a animar o Passeio do Cavalo.

15.25 (***A tartaruga e a lebre***) Crie uma versão gráfica da simulação da *tartaruga e lebre* (Exercício 7.28). Simule a montanha desenhando um arco que se estende do canto inferior esquerdo da janela ao canto superior direito. A lebre e a tartaruga devem correr subindo a montanha. Implemente a saída gráfica para imprimir a tartaruga e a lebre no arco a cada movimento. [*Dica:* estenda o percurso da corrida de 70 para 300 a fim de permitir uma área gráfica maior.]

15.26 (***Desenhando espirais***) Escreva um aplicativo que utilize o método `Graphics` `drawPolyline` para desenhar uma espiral semelhante àquela mostrada na Figura 15.33.

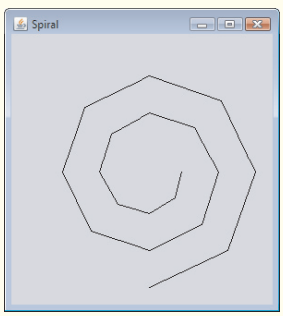


Figura 15.33 | Desenho de espiral utilizando o método `drawPolyline`.

15.27 (***Gráfico de torta***) Escreva um programa que insira quatro números e disponha os números em um gráfico de torta. Utilize a classe `Arc2D.Double` e o método `fill` da classe `Graphics2D` para executar o desenho. Desenhe cada pedaço da torta em uma cor separada.

15.28 (***Selecionando formas***) Escreva um aplicativo que permita ao usuário selecionar uma forma a partir de uma `JComboBox` e a desenhe 20 vezes com posições e dimensões aleatórias no método `paintComponent`. O primeiro item na `JComboBox` deve ser a forma padrão que é exibida na primeira vez em que `paintComponent` é chamado.

15.29 (***Cores aleatórias***) Modifique o Exercício 15.28 para desenhar cada uma das 20 formas com dimensões aleatórias em uma cor selecionada aleatoriamente. Utilize todos os 13 objetos `Color` predefinidos em um array de `Colors`.

15.30 (***Diálogo `JColorChooser`***) Modifique o Exercício 15.28 para permitir ao usuário selecionar a cor em que as formas devem ser desenhadas a partir de um diálogo `JColorChooser`.

(*Opcional*) *Estudo de caso de GUIs e imagens gráficas: adicionando Java2D*

15.31 O Java2D introduz várias novas capacidades para criar imagens gráficas únicas e impressionantes. Adicionaremos um pequeno subconjunto desses recursos ao aplicativo de desenho que você criou no Exercício 14.17. Nessa versão, você permitirá que o usuário especifique gradientes para preencher formas e alterar as características dos traços e desenhar linhas e contornos das formas. O usuário será capaz de escolher as cores que compõem o gradiente e configurar a largura e comprimento do traço da linha tracejada.

Primeiro, você deve atualizar a hierarquia `MyShape` para suportar a funcionalidade do Java2D. Faça as seguintes alterações na classe `MyShape`:

a) Modifique o tipo de parâmetro do método `draw` `abstract` de `Graphics` para `Graphics2D`.

b) Altere todas as variáveis do tipo `Color` para o tipo `Paint` a fim de permitir suporte a gradientes. [*Nota:* lembre-se de que a classe `Color` implementa a interface `Paint`.]

c) Adicione uma variável de instância do tipo `Stroke` à classe `MyShape` e um parâmetro `Stroke` ao construtor para inicializar a nova variável de instância. O traço padrão deve ser uma instância da classe `BasicStroke`.

As classes `MyLine`, `MyBoundedShape`, `MyOval` e `MyRectangle` devem adicionar um parâmetro `Stroke` aos seus construtores. Nos métodos `draw`, cada forma deve configurar `Paint` e `Stroke` antes de desenhar ou preencher uma forma. Como `Graphics2D` é uma subclasse de `Graphics`, podemos continuar a utilizar os métodos `Graphics` `drawLine`, `drawOval`, `fillOval` etc. para desenhar as formas. Quando esses métodos são chamados, eles desenharão a forma apropriada utilizando as configurações `Paint` e `Stroke` especificadas.

Em seguida, você atualizará a `DrawPanel` para tratar os recursos Java2D. Altere todas as variáveis `Color` para variáveis `Paint`. Declare uma variável de instância `currentStroke` do tipo `Stroke` e forneça um método *set* para ela. Atualize as chamadas aos construtores individuais da forma para incluir os argumentos `Paint` e `Stroke`. No método `paintComponent`, faça uma coerção da referência `Graphics` ao tipo `Graphics2D` e use a referência `Graphics2D` em cada chamada ao método `draw` `MyShape`.

Em seguida, torne os novos recursos Java2D acessíveis na GUI. Crie um `JPanel` de componentes GUI para configurar as opções Java2D. Adicione esses componentes à parte superior de `DrawFrame` abaixo do painel que contém atualmente os controles padrão da forma (veja a Figura 15.34). Esses componentes GUI devem incluir:

a) Uma caixa de seleção para especificar pintura com um gradiente.

b) Dois `JButtons` que mostram um diálogo `JColorChooser` para permitir que o usuário escolha a primeira e segunda cores no gradiente. (Estes substituirão o `JComboBox` utilizado para escolher a cor no Exercício 14.17.)

c) Um campo de texto para inserir a largura de `Stroke`.

d) Um campo de texto para inserir o comprimento do traço da linha tracejada de `Stroke`.

e) Uma caixa de seleção para selecionar se desenha uma linha tracejada ou sólida.

Se o usuário optar por desenhar com um gradiente, configure `Paint` no `DrawPanel` para ser um gradiente das duas cores escolhidas pelo usuário. A expressão

```
new GradientPaint( 0, 0, color1, 50, 50, color2, true ))
```

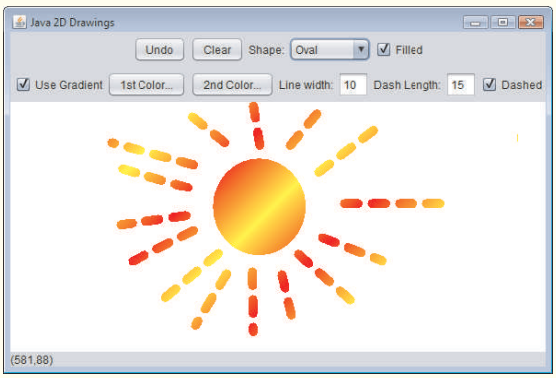


Figura 15.34 | Desenhando com Java2D.

cria um `GradientPaint` que, a cada 50 pixels, vai diagonalmente da parte superior esquerda para a parte inferior direita. As variáveis `color1` e `color2` representam as cores escolhidas pelo usuário. Se o usuário optar por não utilizar um gradiente, simplesmente configure `Paint` em `DrawPanel` como a primeira `Color` escolhida pelo usuário.

Para traços, se o usuário escolher uma linha sólida, crie o `Stroke` com a expressão

```
new BasicStroke( width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND )
```

onde a variável `largura` é a largura especificada pelo usuário no campo de texto de largura da linha. Se o usuário escolher uma linha tracejada, crie o `Stroke` com a expressão

```
new BasicStroke( width, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, 10, dashes, 0 )
```

onde `largura` é novamente a largura no campo de largura da linha e `dashes` é um array com um elemento cujo valor é o comprimento especificado no campo de comprimento do traço da linha tracejada. Os objetos `Panel` e `Stroke` devem ser passados para o construtor do objeto forma quando a forma é criada em `DrawPanel`.

Fazendo a diferença

15.32 (***Exibição de texto grande para pessoas com problemas de visão***) A acessibilidade dos computadores e da Internet para todas as pessoas, independentemente de deficiências, está se tornando mais importante à medida que essas ferramentas desempenham papéis cada vez mais importantes nas nossas vidas pessoais e profissionais. De acordo com uma estimativa recente da Organização Mundial da Saúde (`www.who.int/mediacentre/factsheets/fs282/en/`), 124 milhões de pessoas no mundo todo têm problemas de visão. Para aprender mais sobre os problemas de visão, verifique a simulação dos problemas de visão baseada em GUI em `www.webaim.org/simulations/lowvision.php`. Pessoas com problemas de visão talvez prefiram escolher uma fonte e/ou um tamanho de fonte maior ao ler documentos eletrônicos e páginas Web. O Java tem cinco fontes "lógicas" predefinidas que seguramente estão disponíveis em qualquer implementação Java, incluindo `Serif`, `SansSerif` e `Monospaced`. Escreva um aplicativo GUI que forneça uma `JTextArea` na qual o usuário pode digitar o texto. Permita que o usuário selecione `Serif`, `SansSerif` ou `Monospaced` a partir de um `JComboBox`. Forneça uma `JCheckBox` **Bold**, que, se marcada, torna o texto negritado. Inclua os `JButtons` **Increase Font Size** e **Decrease Font Size** que permitem ao usuário aumentar ou diminuir o tamanho da fonte, respectivamente, por um ponto em um dado momento. Comece com um tamanho de fonte de 18 pontos. Para os objetivos deste exercício, configure o tamanho da fonte nos `JComboBox`, `JButtons` e `JCheckBox` como 20 pontos para que uma pessoa com problemas de visão consiga ler o texto neles.

The chief defect of Henry King
Was chewing little bits of string.
[O principal defeito do rei Henrique
era mastigar pedaços de cordão.]
— Hilaire Belloc

Um texto vigoroso é conciso. Uma frase não deve conter nenhuma palavra desnecessária; um parágrafo, nenhuma frase desnecessária.
— William Strunk, Jr.

Escrevi uma carta mais longa que o normal, porque me falta tempo para fazê-la mais curta.
— Blaise Pascal

16

Strings, caracteres e expressões regulares

Objetivos

Neste capítulo, você aprenderá:

- A criar e manipular imutáveis objetos de string de caractere da classe `String`.
- A criar e manipular objetos mutáveis de string de caractere da classe `StringBuilder`.
- A criar e manipular objetos da classe `Character`.
- A dividir um objeto `String` em tokens utilizando o método `String split`.
- A utilizar expressões regulares para validar dados `String` inseridos em um aplicativo.

Sumário

16.1 Introdução
16.2 Fundamentos de caracteres e strings
16.3 Classe `String`
- 16.3.1 Construtores `String`
- 16.3.2 Métodos `String` `length`, `charAt` e `getChars`
- 16.3.3 Comparando `String`s
- 16.3.4 Localizando caracteres e substrings em strings
- 16.3.5 Extraindo substrings de strings
- 16.3.6 Concatenando strings
- 16.3.7 Métodos de `String` diversos
- 16.3.8 Método `String` `valueOf`

16.4 Classe `StringBuilder`
- 16.4.1 Construtores `StringBuilder`
- 16.4.2 Métodos `StringBuilder` `length`, `capacity`, `setLength` e `ensureCapacity`
- 16.4.3 Métodos `StringBuilder` `charAt`, `setCharAt`, `getChars` e `reverse`
- 16.4.4 Métodos `StringBuilder` `append`
- 16.4.5 Métodos de inserção e exclusão de `StringBuilder`

16.5 Classe `Character`
16.6 Tokenização de `String`s
16.7 Expressões regulares, classe `Pattern` e classe `Matcher`
16.8 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Seção especial: exercícios de manipulação avançada de string | Seção especial: projetos de manipulação de string desafiadores | Fazendo a diferença

16.1 Introdução

Este capítulo introduz as capacidades de processamento de string e caractere do Java. As técnicas discutidas aqui são apropriadas para validar a entrada de programa, exibir informações para usuários e outras manipulações baseadas em texto. Elas também são apropriadas para desenvolver editores de texto, processadores de texto, software de layout de página, sistemas computadorizados de composição e outros tipos de software de processamento de textos. Já apresentamos várias capacidades de processamento de string nos capítulos anteriores. Este capítulo discute detalhadamente as capacidades da classe `String` bem como as classes `StringBuilder` e `Character` do pacote `java.lang`. Essas classes fornecem os princípios básicos para manipulação de string e caractere no Java.

O capítulo também discute as expressões regulares que fornecem aos aplicativos a capacidade de validar entrada. As funcionalidades encontram-se na classe `String` com as classes `Matcher` e `Pattern` localizadas no pacote `java.util.regex`.

16.2 Fundamentos de caracteres e strings

Os caracteres são os blocos de construção fundamentais dos programas-fonte do Java. Cada programa é composto de uma sequência de caracteres que — quando agrupados entre si significativamente — é interpretada pelo computador como uma série de instruções utilizadas para realizar uma tarefa. Um programa pode conter **literais de caractere**. Um literal de caractere é um valor inteiro representado como caractere entre aspas simples. Por exemplo, `'z'` representa o valor inteiro de `z` e `'\n'` representa o valor inteiro de nova linha. O valor de um literal de caractere é o valor inteiro do caractere no **conjunto de caracteres Unicode**. O Apêndice B apresenta os equivalentes inteiros dos caracteres no conjunto de caracteres ASCII, que é um subconjunto de Unicode (discutido no Apêndice L). Para informações detalhadas sobre o Unicode, visite `www.unicode.org`.

A partir da discussão da Seção 2.2, lembre-se de que uma string é uma sequência de caracteres tratada como uma única unidade. Uma string pode incluir letras, dígitos e vários **caracteres especiais**, como `+`, `-`, `*`, `/` e `$`. Uma string é um objeto de classe `String`. Os **literais de string** (armazenados na memória como objetos `String`) são escritos como uma sequência de caracteres entre aspas duplas, como em:

```
"John Q. Doe"                (a name)
"9999 Main Street"           (a street address)
"Waltham, Massachusetts"     (a city and state)
"(201) 555-1212"             (a telephone number)
```

Uma string pode ser atribuída a uma referência `String`. A declaração

```
String color = "blue";
```

inicializa a variável `String` `color` para referenciar um objeto `String` que contém a string `"blue"`.

Dica de desempenho 16.1

O Java trata todos os literais string com o mesmo conteúdo de um único objeto `String` que tem muitas referências a ele. Isso economiza memória.

16.3 Classe `String`

A classe `String` é utilizada para representar strings em Java. As próximas várias subseções discutem muitas das capacidades da classe `String`.

16.3.1 Construtores de `String`

A classe `String` fornece construtores para inicializar objetos `String` de uma variedade de maneiras. Quatro dos construtores são demonstrados no método `main` da Figura 16.1.

```
1  // Figura 16.1: StringConstructors.java
2  // construtores da classe String.
3
4  public class StringConstructors
5  {
6     public static void main( String[] args )
7     {
8        char[] charArray = { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
9        String s = new String( "hello" );
10
11       // utiliza os construtores String
12       String s1 = new String();
13       String s2 = new String( s );
14       String s3 = new String( charArray );
15       String s4 = new String( charArray, 6, 3 );
16
17       System.out.printf(
18          "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
19          s1, s2, s3, s4 ); // exibe strings
20    } // fim de main
21 } // fim da classe StringConstructors
```

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

Figura 16.1 | Construtores de classe `string`.

A linha 12 instancia um novo objeto `String` utilizando o construtor sem argumento da classe `String` e atribui sua referência a `s1`. O novo objeto `String` não contém caracteres (isto é, a **string vazia**, que também pode ser representada como `""`) e tem um tamanho de 0. A linha 13 instancia um novo objeto `String` utilizando o construtor da classe `String` que aceita um objeto `String` como um argumento e destina a sua referência a `s2`. O novo objeto `String` contém a mesma sequência de caracteres do objeto `String` `s` que é passado como um argumento ao construtor.

Observação de engenharia de software 16.1

*Não é necessário copiar um objeto `String` existente. Os objetos `String` são **imutáveis** — seu conteúdo de caractere não pode ser alterado depois que eles são criados, porque a classe `String` não fornece nenhum método que permita que o conteúdo de um objeto `String` seja modificado.*

A linha 14 instancia um novo objeto `String` e atribui sua referência à classe `s3` utilizando o construtor de `String` que aceita um array `char` como um argumento. O novo objeto `String` contém uma cópia dos caracteres no array.

A linha 15 instancia um novo objeto `String` e atribui sua referência à classe `s4` utilizando o construtor de `String` que aceita um array de `chars` e dois inteiros como argumentos. O segundo argumento especifica a posição inicial (o deslocamento) a partir do qual os caracteres no array são acessados. Lembre-se de que o primeiro caractere está na posição 0. O terceiro argumento especifica o número de

caracteres (a contagem) a acessar no array. O novo objeto `String` é formado dos caracteres acessados. Se o deslocamento ou a contagem especificada como um argumento resultar no acesso a um elemento fora dos limites do array de caracteres, uma `StringIndexOutOfBoundsException` é lançada.

Erro comum de programação 16.1

Acessar um caractere fora dos limites de uma `String` (isto é, um índice menor que 0 ou um índice maior que ou igual ao tamanho da `String`) resulta em uma `StringIndexOutOfBoundsException`.

16.3.2 Métodos `String` `length`, `charAt` e `getChars`

Os métodos **`length`**, **`charAt`** e **`getChars`** de `String` retornam o comprimento de uma `String`, obtêm o caractere em uma localização específica em uma `String` e recuperam um conjunto de caracteres de uma `String` como um array `char`, respectivamente. A Figura 16.2 demonstra cada um desses métodos.

```
1  // Figura 16.2: StringMiscellaneous.java
2  // Esse aplicativo demonstra os métodos da classe String
3  // length, charAt e getChars.
4
5  public class StringMiscellaneous
6  {
7     public static void main( String[] args )
8     {
9        String s1 = "hello there";
10       char[] charArray = new char[ 5 ];
11
12       System.out.printf( "s1: %s", s1 );
13
14       // testa o método length
15       System.out.printf( "\nLength of s1: %d", s1.length() );
16
17       // faz loop pelos caracteres em s1 com charAt e os exibe na ordem inversa
18       System.out.print( "\nThe string reversed is: " );
19
20       for ( int count = s1.length() - 1; count >= 0; count-- )
21          System.out.printf( "%c ", s1.charAt( count ) );
22
23       // copia caracteres a partir de string para charArray
24       s1.getChars( 0, 5, charArray, 0 );
25       System.out.print( "\nThe character array is: " );
26
27       for ( char character : charArray )
28          System.out.print( character );
29
30       System.out.println();
31    } // fim de main
32 } // fim da classe StringMiscellaneous
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

Figura 16.2 | Os métodos de manipulação de caractere de classe `String`.

A linha 15 utiliza o método `String` `length` para determinar o número de caracteres em string `s1`. Como os arrays, as strings conhecem seu próprio comprimento. Entretanto, diferentemente dos arrays, você não pode acessar o comprimento de uma `String` via um campo `length` — em vez disso você deve chamar o método `length` de `String`.

As linhas 20–21 imprimem os caracteres da `String` `s1` em ordem inversa (e separados por espaços). O método `String` `charAt` (linha 21) retorna o caractere em uma posição específica na `String`. O método `charAt` recebe um argumento inteiro que é utilizado como o índice e retorna o caractere nessa posição. Como os arrays, o primeiro elemento de uma `String` está na posição 0.

A linha 24 utiliza o método `String` `getChars` para copiar os caracteres de uma `String` em um array de caracteres. O primeiro argumento é o índice inicial na `String` a partir da qual os caracteres devem ser copiados. O segundo argumento é o índice que está um além do

último caractere que será copiado da `String`. O terceiro argumento é o array de caracteres em que eles devem ser copiados. O último argumento é o índice inicial no qual os caracteres copiados são colocados no array de caracteres-alvo. Em seguida, as linhas 27–28 imprimem o conteúdo do array `char` um caractere por vez.

16.3.3 Comparando `Strings`

O Capítulo 19 discute a classificação e pesquisa de arrays. Frequentemente, as informações a serem classificadas ou pesquisadas consistem em `String`s que devem ser comparadas para colocá-las em ordem ou determinar se uma string aparece em um array (ou em outra coleção). A classe `String` fornece vários métodos para comparar strings, como os demonstrados nos dois exemplos a seguir.

Para entender o que significa uma string ser "maior que" ou "menor que" outra, considere o processo de alfabetar uma série de sobrenomes. Sem dúvida, você colocaria "Jones" antes de "Smith", pois a primeira letra de "Jones" vem antes da primeira letra de "Smith" no alfabeto. Mas o alfabeto é mais do que uma mera lista de 26 letras — é um conjunto ordenado de caracteres. Cada letra ocorre em uma posição específica dentro do conjunto. O "z" é mais do que apenas uma letra do alfabeto — ele é especificamente a vigésima sexta letra do alfabeto.

Como o computador sabe que uma letra vem antes de outra? Todos os caracteres são representados no computador como códigos numéricos (ver o Apêndice B). Quando o computador compara strings, ele realmente compara os códigos numéricos dos caracteres nas strings.

A Figura 16.3 demonstra os métodos `String` `equals`, `equalsIgnoreCase`, `compareTo` e **`regionMatches`** e a utilização do operador de igualdade == para comparar objetos `String`.

```
1  // Figura 16.3: StringCompare.java
2  // Métodos String equals, equalsIgnoreCase, compareTo e regionMatches.
3
4  public class StringCompare
5  {
6     public static void main( String[] args )
7     {
8        String s1 = new String( "hello" ); // s1 é uma cópia de "hello"
9        String s2 = "goodbye";
10       String s3 = "Happy Birthday";
11       String s4 = "happy birthday";
12
13       System.out.printf(
14          "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );
15
16       // teste para igualdade
17       if (s1.equals( "hello" ) )  // true
18          System.out.println( "s1 equals \"hello\"" );
19       else
20          System.out.println( "s1 does not equal \"hello\"" );
21
22       // testa quanto à igualdade com ==
23       if (s1 == "hello" )  // false; eles não são os mesmos objetos
24          System.out.println( "s1 is the same object as \"hello\"" );
25       else
26          System.out.println( "s1 is not the same object as \"hello\"" );
27
28       // testa quanto à igualdade (ignora maiúsculas e minúsculas)
29       if (s3.equalsIgnoreCase( s4 ) )  // true
30          System.out.printf( "%s equals %s with case ignored\n", s3, s4 );
31       else
32          System.out.println( "s3 does not equal s4" );
33
34       // testa compareTo
35       System.out.printf(
36          "\ns1.compareTo( s2 ) is %d", s1.compareTo( s2 ) );
37       System.out.printf(
38          "\ns2.compareTo( s1 ) is %d", s2.compareTo( s1 ) );
39       System.out.printf(
40          "\ns1.compareTo( s1 ) is %d", s1.compareTo( s1 ) );
41       System.out.printf(
42          "\ns3.compareTo( s4 ) is %d", s3.compareTo( s4 ) );
43       System.out.printf(
44          "\ns4.compareTo( s3 ) is %d\n\n", s4.compareTo( s3 ) );
45
46       // testa regionMatches (distingue maiúsculas e minúsculas)
47       if (s3.regionMatches( 0, s4, 0, 5 ) )
48          System.out.println( "First 5 characters of s3 and s4 match" );
49       else
```

```
50             System.out.println(
51                "First 5 characters of s3 and s4 do not match" );
52
53          // testa regionMatches (ignora maiúsculas e minúsculas)
54          if (s3.regionMatches( true, 0, s4, 0, 5 ) )
55             System.out.println(
56                "First 5 characters of s3 and s4 match with case ignored" );
57          else
58             System.out.println(
59                "First 5 characters of s3 and s4 do not match" );
60       } // fim de main
61    } // fim da classe StringCompare
```

```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1 equals "hello"
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo( s2 ) is 1
s2.compareTo( s1 ) is -1
s1.compareTo( s1 ) is 0
s3.compareTo( s4 ) is -32
s4.compareTo( s3 ) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

Figura 16.3 | Métodos `string` `equals`, `equalsIgnoreCase`, `compareTo` e `regionMatches`.

A condição na linha 17 utiliza o método `equals` para comparar `String` `s1` e o literal da `String` `"hello"` quanto à igualdade. O método `equals` (um método da classe `Object` sobrescrito em `String`) testa dois objetos quaisquer quanto à igualdade — as strings contidas nos dois objetos são idênticas. O método retorna `true` se os conteúdos dos objetos forem iguais e `false`, caso contrário. A condição anterior é `true` porque `String` `s1` foi inicializada com o literal de string `"hello"`. O método `equals` utiliza uma **comparação lexicográfica** — compara os valores inteiros Unicode (ver Apêndice L, para informações adicionais) que representam cada caractere em cada `String`. Portanto, se a `String` `"hello"` é comparada com a string `"HELLO"`, o resultado é `false`, pois a representação de inteiro de uma letra minúscula é diferente daquela da letra maiúscula correspondente.

A condição na linha 23 utiliza o operador de igualdade == para comparar a igualdade entre `String` `s1` e o literal da `String` `"hello"`. Quando valores de tipo de dados primitivo são comparados com ==, o resultado é `true` se ambos os valores forem idênticos. Quando referências são comparadas com ==, o resultado é `true` se ambas as referências referenciam o mesmo objeto na memória. Para comparar o conteúdo real (ou informações de estado) de objetos quanto à igualdade, um método deve ser invocado. No caso de `String`s, esse método é `equals`. A condição anterior é avaliada como `false` na linha 23 porque a referência `s1` foi inicializada com a instrução

```
s1 = new String( "hello" );
```

que cria um novo objeto `String` com uma cópia de literal da string `"hello"` e atribui o novo objeto à variável `s1`. Se `s1` tivesse sido inicializada com a instrução

```
s1 = "hello";
```

que atribui diretamente o literal de string `"hello"` à variável `s1`, a condição seria `true`. Lembre-se de que o Java trata todos os objetos de literal string com o mesmo conteúdo como um objeto `String` ao qual há muitas referências. Portanto, as linhas 8, 17 e 23 todas referenciam o mesmo objeto `String` `"hello"` na memória.

Erro comum de programação 16.2

Comparar referências com == pode levar a erros de lógica, porque == compara as referências a fim de determinar se elas referenciam o mesmo objeto, não se dois objetos têm o mesmo conteúdo. Quando dois objetos idênticos (mas separados) são comparados com ==, o resultado será `false`. Ao comparar objetos para determinar se eles têm o mesmo conteúdo, utilize o método `equals`.

Se estiver classificando `String`s, você pode compará-las quanto à igualdade com o método `equals-IgnoreCase`, que ignora se as letras em cada `String` são maiúsculas ou minúsculas ao realizar a comparação. Assim, `"hello"` e `"HELLO"` são consideradas iguais. A linha

29 utiliza o método String equalsIgnoreCase para comparar String s3 — Happy Birthday — quanto à igualdade com String s4 — happy birthday. O resultado dessa comparação é true porque a comparação ignora distinção entre maiúsculas e minúsculas.

As linhas 35–44 utilizam o método compareTo para comparar Strings. O método compareTo é declarado na interface Comparable e implementado na classe String. A linha 36 compara String s1 com String s2. O método compareTo retorna 0 se as Strings forem iguais, um número negativo se a String que invoca compareTo for menor que a String que é passada como um argumento e um número positivo se a String que invoca compareTo for maior que a String que é passada como um argumento. O método compareTo utiliza uma comparação lexicográfica — ele compara os valores numéricos de caracteres correspondentes em cada String. (Para obter mais informações sobre o valor exato retornado pelo método compareTo, consulte java.sun.com/javase/6/docs/api/java/lang/String.html.)

A condição na linha 47 utiliza o método String regionMatches para comparar a igualdade entre partes de duas Strings. O primeiro argumento é o índice inicial na String que invoca o método. O segundo argumento é uma comparação de String. O terceiro argumento é o índice inicial na comparação de String. O último argumento é o número de caracteres a comparar entre as duas Strings. O método retorna true apenas se o número especificado de caracteres for lexicograficamente igual.

Por fim, a condição na linha 54 utiliza uma versão de cinco argumentos do método String regionMatches para comparar a igualdade de partes de duas Strings. Quando o primeiro argumento é true, o método ignora maiúsculas e minúsculas dos caracteres sendo comparados. Os argumentos restantes são idênticos àqueles descritos para o método de quatro argumentos regionMatches.

O próximo exemplo (Figura 16.4) demonstra os métodos String **startsWith** e **endsWith**. O método main cria o array strings que contém "started", "starting", "ended" e "ending". O restante do método main consiste em três instruções for que testam os elementos do array para determinar se eles iniciam ou terminam com um conjunto particular de caracteres.

```
1  // Figura 16.4: StringStartEnd.java
2  // métodos String StartsWith e endsWith.
3
4  public class StringStartEnd
5  {
6     public static void main( String[] args )
7     {
8        String[] strings = { "started", "starting", "ended", "ending" };
9
10       // testa o método startsWith
11       for ( String string : strings )
12       {
13          if (string.startsWith( "st" ) )
14             System.out.printf( "\"%s\" starts with \"st\"\n", string );
15       } // for final
16
17       System.out.println();
18
19       // testa o método startsWith iniciando da posição 2 de string
20       for ( String string : strings )
21       {
22          if (string.startsWith( "art", 2 ) )
23             System.out.printf(
24                "\"%s\" starts with \"art\" at position 2\n", string );
25       } // for final
26
27       System.out.println();
28
29       // testa o método endsWith
30       for ( String string : strings )
31       {
32          if (string.endsWith( "ed" ) )
33             System.out.printf( "\"%s\" ends with \"ed\"\n", string );
34       } // for final
35    } // fim de main
36 } // fim da classe StringStartEnd
```

```
"started" starts with "st"
"starting" starts with "st"

"started" starts with "art" at position 2
"starting" starts with "art" at position 2

"started" ends with "ed"
"ended" ends with "ed"
```

Figura 16.4 | Métodos String startsWith e endsWith.

As linhas 11–15 utilizam a versão do método `startsWith` que aceita um argumento `String`. A condição na instrução `if` (linha 13) determina se cada `String` no array inicia com os caracteres `"st"`. Se iniciar, o método retorna `true` e o aplicativo imprime essa `String`. Caso contrário, o método retorna `false` e nada acontece.

As linhas 20–25 utilizam o método `startsWith` que aceita uma `String` e um número inteiro como argumentos. O inteiro especifica o índice em que a comparação deve iniciar na `String`. A condição na instrução `if` (linha 22) determina se cada `String` no array tem os caracteres `"art"` começando com o terceiro caractere em cada `String`. Se tiver, o método retorna `true` e o aplicativo imprime a `String`.

A terceira instrução `for` (linhas 30–34) utiliza o método `endsWith`, que aceita um argumento `String`. A condição na linha 32 determina se cada `String` no array termina com os caracteres `"ed"`. Se tiver, o método retorna `true` e o aplicativo imprime a `String`.

16.3.4 Localizando caracteres e substrings em strings

Costuma ser útil pesquisar uma string para um caractere ou conjunto de caracteres. Por exemplo, se estiver criando seu próprio processador de texto, você poderia querer fornecer a capacidade de pesquisar por documentos. A Figura 16.5 demonstra as muitas versões dos métodos `String` **`indexOf`** e **`lastIndexOf`** que procuram um caractere especificado ou substring em uma `String`.

```
1  // Figura 16.5: StringIndexMethods.java
2  // Métodos de pesquisa de String indexOf e lastIndexOf.
3
4  public class StringIndexMethods
5  {
6     public static void main( String[] args )
7     {
8        String letters = "abcdefghijklmabcdefghijklm";
9
10       // testa indexOf para localizar um caractere em uma string
11       System.out.printf(
12          "'c' is located at index %d\n", letters.indexOf( 'c' ) );
13       System.out.printf(
14          "'a' is located at index %d\n", letters.indexOf( 'a', 1 ) );
15       System.out.printf(
16          "'$' is located at index %d\n\n", letters.indexOf( '$' ) );
17
18       // testa lastIndexOf para localizar um caractere em uma string
19       System.out.printf( "Last 'c' is located at index %d\n",
20          letters.lastIndexOf( 'c' ) );
21       System.out.printf( "Last 'a' is located at index %d\n",
22          letters.lastIndexOf( 'a', 25 ) );
23       System.out.printf( "Last '$' is located at index %d\n\n",
24          letters.lastIndexOf( '$' ) );
25
26       // testa indexOf para localizar uma substring em uma string
27       System.out.printf( "\"def\" is located at index %d\n",
28          letters.indexOf( "def" ) );
29       System.out.printf( "\"def\" is located at index %d\n",
30          letters.indexOf( "def", 7 ) );
31       System.out.printf( "\"hello\" is located at index %d\n\n",
32          letters.indexOf( "hello" ) );
33
34       // testa lastIndexOf para localizar uma substring em uma string
35       System.out.printf( "Last \"def\" is located at index %d\n",
36          letters.lastIndexOf( "def" ) );
37       System.out.printf( "Last \"def\" is located at index %d\n",
38          letters.lastIndexOf( "def", 25 ) );
39       System.out.printf( "Last \"hello\" is located at index %d\n",
40          letters.lastIndexOf( "hello" ) );
41    } // fim de main
42 } // fim da classe StringIndexMethods
```

```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1

Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1
```

```
"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

Figura 16.5 | Métodos de pesquisa de string `indexOf` e `lastIndexOf`.

Todas as pesquisas nesse exemplo são realizadas em `String letters` (inicializadas com `"abcdefghijklmabcdefghijklm"`). As linhas 11–16 utilizam o método `indexOf` para localizar a primeira ocorrência de um caractere em uma `String`. Se o método localizar o caractere, ele retorna o índice do caractere na `String` — caso contrário, retorna –1. Há duas versões de `indexOf` que procuram caracteres em uma `String`. A expressão na linha 12 utiliza a versão do método `indexOf` que aceita uma representação de inteiro do caractere a localizar. A expressão na linha 14 utiliza outra versão do método `indexOf`, que aceita dois argumentos inteiros — o caractere e o índice inicial em que a pesquisa da `String` deve iniciar.

As linhas 19–24 utilizam o método `lastIndexOf` para localizar a última ocorrência de um caractere em uma `String`. O método procura do fim da `String` em direção ao começo. Se localizar o caractere, ele retorna o índice do caractere na `String` — caso contrário, ele retorna –1. Há duas versões de `lastIndexOf` que pesquisam por caracteres em uma `String`. A expressão na linha 20 utiliza a versão que aceita a representação de inteiro do caractere. A expressão na linha 22 utiliza a versão que aceita dois argumentos inteiros — a representação de inteiro do caractere e o índice a partir do qual iniciar a pesquisa de trás para frente.

As linhas 27–40 demonstram versões dos métodos `indexOf` e `lastIndexOf` que aceitam uma `String` como o primeiro argumento. Essas versões trabalham identicamente àquelas descritas anteriormente exceto que procuram sequências de caracteres (ou substrings) que são especificadas por seus argumentos `String`. Se a substring for localizada, esses métodos retornam o índice na `String` do primeiro caractere na substring.

16.3.5 Extraindo substrings de strings

A classe `String` fornece dois métodos `substring` para permitir que um novo objeto `String` seja criado copiando parte de um objeto `String` existente. Cada método retorna um novo objeto `String`. Ambos os métodos são demonstrados na Figura 16.6.

```
1  // Figura 16.6: SubString.java
2  // métodos substring da classe String.
3
4  public class SubString
5  {
6     public static void main( String[] args )
7     {
8        String letters = "abcdefghijklmabcdefghijklm";
9
10       // testa métodos substring
11       System.out.printf( "Substring from index 20 to end is \"%s\"\n",
12          letters.substring( 20 ) );
13       System.out.printf( "%s \"%s\"\n",
14          "Substring from index 3 up to, but not including 6 is",
15          letters.substring( 3, 6 ) );
16    } // fim de main
17 } // fim da classe Substring
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including 6 is "def"
```

Figura 16.6 | Métodos `substring` da classe `String`.

A expressão `letters.substring(20)` na linha 12 utiliza o método `substring` que aceita um argumento inteiro. O argumento especifica o índice inicial na `String letters` original a partir da qual os caracteres devem ser copiados. A substring retornada contém uma cópia dos caracteres desde o índice inicial até o final da `String`. Especificar um índice fora dos limites da `String` causa uma **`StringIndexOutOfBoundsException`**.

A linha 15 utiliza o método `substring` que aceita dois argumentos do tipo inteiro — o índice inicial a partir do qual copiar caracteres na `String` original e o índice um além do último caractere que será copiado (isto é, copiar até, mas não incluindo, esse índice na `String`). A substring retornada contém uma cópia dos caracteres especificados da `String` original. Um índice fora dos limites da `String` causa uma `StringIndexOutOfBoundsException`.

16.3.6 Concatenando strings

O método `String` **`concat`** (Figura 16.7) concatena dois objetos `String` e retorna um novo objeto `String` que contém os caracteres de ambas as `String`s originais. A expressão `s1.concat(s2)` na linha 13 forma uma `String` acrescentando os caracteres em `s2` aos caracteres em `s1`. As `String`s originais que referenciam `s1` e `s2` não são modificadas.

```
1  // Figura 16.7: StringConcatenation.java
2  // Método string concat.
3
4  public class StringConcatenation
5  {
6     public static void main( String[] args )
7     {
8        String s1 = "Happy ";
9        String s2 = "Birthday";
10
11       System.out.printf( "s1 = %s\ns2 = %s\n\n",s1, s2 );
12       System.out.printf(
13          "Result of s1.concat( s2 ) = %s\n", s1.concat( s2 ) );
14       System.out.printf( "s1 after concatenation = %s\n", s1 );
15    } // fim de main
16 } // fim da classe StringConcatenation
```

```
s1 = Happy
s2 = Birthday

Result of s1.concat( s2 ) = Happy Birthday
s1 after concatenation = Happy
```

Figura 16.7 | Método `String` `concat`.

16.3.7 Métodos de `String` diversos

A classe `String` fornece vários métodos que retornam cópias modificadas de `String`s ou que retornam arrays de caracteres. Esses métodos são demonstrados no aplicativo na Figura 16.8.

```
1  // Figura 16.8: StringMiscellaneous2.java
2  // Métodos String replace, toLowerCase, toUpperCase, trim e toCharArray.
3
4  public class StringMiscellaneous2
5  {
6     public static void main( String[] args )
7     {
8        String s1 = "hello";
9        String s2 = "GOODBYE";
10       String s3 = "   spaces   ";
11
12       System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\n\n", s1, s2, s3 );
13
14       // testa o método replace
15       System.out.printf(
16          "Replace 'l' with 'L' in s1: %s\n\n", s1.replace( 'l', 'L' ) );
17
18       // testa o toLowerCase e toUpperCase
19       System.out.printf( "s1.toUpperCase() = %s\n", s1.toUpperCase() );
20       System.out.printf( "s2.toLowerCase() = %s\n\n", s2.toLowerCase() );
21
22       // testa o método trim
23       System.out.printf( "s3 after trim = \"%s\"\n\n", s3.trim() );
24
25       // testa o método toCharArray
26       char[] charArray = s1.toCharArray();
27       System.out.print( "s1 as a character array = " );
28
```

```
29          for ( char character : charArray )
30             System.out.print( character );
31
32          System.out.println();
33       } // fim de main
34    } // fim da classe StringMiscellaneous2
```

```
s1 = hello
s2 = GOODBYE
s3 =    spaces

Replace 'l' with 'L' in s1: heLLo

s1.toUpperCase() = HELLO
s2.toLowerCase() = goodbye

s3 after trim = "spaces"

s1 as a character array = hello
```

Figura 16.8 | Métodos `String` `replace`, `toLowerCase`, `toUpperCase`, `trim` e `toCharArray`.

A linha 16 utiliza o método `String` `replace` para retornar um novo objeto `String` em que cada ocorrência na `s1` do caractere `'l'` (letras minúsculas el) é substituída pelo caractere `'L'`. O método `replace` deixa a `String` original inalterada. Se não houver nenhuma ocorrência do primeiro argumento na `String`, o método `replace` retorna a `String` original. Uma versão sobrecarregada do método `replace` permite substituir substrings em vez de caracteres individuais.

A linha 19 utiliza o método `String` **`toUpperCase`** para gerar uma nova `String` com letras maiúsculas na qual letras minúsculas correspondentes existem em `s1`. O método retorna um novo objeto `String` contendo a `String` convertida e deixa a `String` original inalterada. Se não houver caractere para converter, o método `toUpperCase` retorna a `String` original.

A linha 20 utiliza o método `String` **`toLowerCase`** para retornar um novo objeto `String` com letras minúsculas no qual há letras maiúsculas correspondentes em `s2`. A `String` original permanece inalterada. Se não houver caracteres na `String` original para serem convertidos, `toLowerCase` retorna a `String` original.

A linha 23 utiliza o método `String` **`trim`** para gerar um novo objeto `String` que remove todos os caracteres de espaço em branco que aparecem no início ou no fim da `String` em que `trim` opera. O método retorna um novo objeto `String` contendo a `String` sem espaço em branco inicial ou final. A `String` original permanece inalterada.

A linha 26 utiliza o método `String` **`toCharArray`** para criar um novo array de caractere que contém uma cópia dos caracteres em `s1`. As linhas 29–30 geram saída de cada `char` no array.

16.3.8 Método `String` `valueOf`

Como vimos, cada objeto em Java tem um método `toString` que permite que um programa obtenha a representação de string do objeto. Infelizmente, essa técnica não pode ser utilizada com tipos primitivos porque eles não têm métodos. A classe `String` fornece os métodos `static` que aceitam um argumento de qualquer tipo e o convertem em um objeto `String`. A Figura 16.9 demonstra os métodos **`valueOf`** da classe `String`.

```
1  // Figura 16.9: StringValueOf.java
2  // Métodos valueOf de String.
3
4  public class StringValueOf
5  {
6     public static void main( String[] args )
7     {
8        char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
9        boolean booleanValue = true;
10       char characterValue = 'Z';
11       int integerValue = 7;
12       long longValue = 10000000000L; // sufixo L indica tipo long
13       float floatValue = 2.5f; // f indica que 2.5 é um tipo float
14       double doubleValue = 33.333; // sem sufixo, tipo double é padrão
15       Object objectRef = "hello"; // atribui string a uma referência Object
16
17       System.out.printf(
18          "char array = %s\n", String.valueOf( charArray ) );
```

```
19          System.out.printf( "part of char array = %s\n",
20             String.valueOf( charArray, 3, 3 ) );
21          System.out.printf(
22             "boolean = %s\n", String.valueOf( booleanValue ) );
23          System.out.printf(
24             "char = %s\n", String.valueOf( characterValue ) );
25          System.out.printf( "int = %s\n", String.valueOf( integerValue ) );
26          System.out.printf( "long = %s\n", String.valueOf( longValue ) );
27          System.out.printf( "float = %s\n", String.valueOf( floatValue ) );
28          System.out.printf(
29             "double = %s\n", String.valueOf( doubleValue ) );
30          System.out.printf( "Object = %s", String.valueOf( objectRef ) );
31       } // fim de main
32    } // fim da classe da StringValueOf
```

```
char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello
```

Figura 16.9 | Métodos `String valueOf`.

A expressão `String.valueOf(charArray)` na linha 18 utiliza o array de caractere `charArray` para criar um novo objeto `String`. A expressão `String.valueOf(charArray, 3, 3)` na linha 20 utiliza uma parte do array de caracteres `charArray` para criar um novo objeto `String`. O segundo argumento especifica o índice inicial a partir do qual os caracteres são utilizados. O terceiro argumento especifica o número de caracteres a ser utilizado.

Há sete outras versões do método `valueOf`, que aceitam argumentos do tipo `boolean`, `char`, `int`, `long`, `float`, `double` e `Object`, respectivamente. Esses são demonstrados nas linhas 21–30. Observe que a versão de `valueOf` que aceita um `Object` como um argumento pode fazer isso porque todos os `Objects` podem ser convertidos em `Strings` com o método `toString`.

[*Nota*: As linhas 12–13 utilizam os valores literais `10000000000L` e `2.5f` como os valores iniciais da variável `long longValue` e variável `float floatValue`, respectivamente. Por padrão, o Java trata os literais de inteiro como o tipo `int` e os literais de ponto flutuante como o tipo `double`. Acrescentar a letra `L` ao literal `10000000000` e letra `f` ao literal `2.5` indica para o compilador que `10000000000` deve ser tratado como um `long` e que `2.5` deve ser tratado como um `float`. Um `L` maiúsculo ou `l` minúsculo pode ser utilizado para denotar uma variável de tipo `long` e um `F` maiúsculo ou `f` minúsculo pode ser utilizado para denotar uma variável de tipo `float`].

16.4 Classe `StringBuilder`

Discutimos agora os recursos da classe **`StringBuilder`** para criar e manipular informações de string dinâmica — isto é, strings modificáveis. Cada `StringBuilder` é capaz de armazenar um número de caracteres especificado pela sua capacidade. Se a capacidade de um `StringBuilder` for excedida, a capacidade se expande para acomodar os caracteres adicionais.

Dica de desempenho 16.2

O Java pode realizar certas otimizações que envolvem objetos `String` (como referenciar um objeto `String` a partir de múltiplas variáveis) porque ele sabe que esses objetos não se alterarão. `Strings` (não `StringBuilders`) devem ser utilizadas se os dados não se alterarem.

Dica de desempenho 16.3

Em programas que frequentemente realizam a concatenação de strings, ou outras modificações de strings, em geral, é mais eficiente implementar as modificações com a classe `StringBuilder`.

Observação de engenharia de software 16.2

*`StringBuilders` não são seguros para thread. Se múltiplos threads exigirem acesso às mesmas informações de string dinâmicas, utilize a classe **`StringBuffer`** no seu código. As classes `StringBuilder` e `StringBuffer` fornecem capacidades idênticas, mas a classe `StringBuffer` é segura para threads. Para obter mais detalhes sobre threading, consulte o Capítulo 26.*

16.4.1 Construtores StringBuilder

A classe StringBuilder fornece quatro construtores. Demonstramos três deles na Figura 16.10. A linha 8 utiliza o construtor sem argumento StringBuilder para criar um StringBuilder sem caracteres e uma capacidade inicial de 16 caracteres (o padrão para um StringBuilder). A linha 9 utiliza o construtor StringBuilder que aceita um argumento inteiro para criar um StringBuilder sem caracteres e a capacidade inicial especificada pelo argumento inteiro (isto é, 10). A linha 10 utiliza o construtor StringBuilder que aceita um argumento String para criar um StringBuilder contendo o caractere no argumento String. A capacidade inicial é o número de caracteres no argumento String mais 16.

As linhas 12–14 utilizam o método toString da classe StringBuilder para gerar saída de StringBuilders com o método printf. Na Seção 16.4.4, discutimos como o Java utiliza objetos StringBuilder para implementar os operadores + e += para concatenação de strings.

```
1  // Figura 16.10: StringBuilderConstructors.java
2  // Construtores StringBuilder.
3
4  public class StringBuilderConstructors
5  {
6     public static void main( String[] args )
7     {
8        StringBuilder buffer1 = new StringBuilder();
9        StringBuilder buffer2 = new StringBuilder( 10 );
10       StringBuilder buffer3 = new StringBuilder( "hello" );
11
12       System.out.printf( "buffer1 = \"%s\"\n", buffer1.toString() );
13       System.out.printf( "buffer2 = \"%s\"\n", buffer2.toString() );
14       System.out.printf( "buffer3 = \"%s\"\n", buffer3.toString() );
15    } // fim de main
16 } // fim da classe StringBuilderConstructors
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

Figura 16.10 | Construtores StringBuilder.

16.4.2 Métodos StringBuilder length, capacity, setLength e ensureCapacity

A classe StringBuilder fornece os métodos **length** e **capacity** para retornar o número de caracteres atualmente em um StringBuilder e o número de caracteres que pode ser armazenado em um StringBuilder sem alocar mais memória, respectivamente. O método **ensureCapacity** garante que um StringBuilder tenha pelo menos a capacidade especificada. O método setLength aumenta ou diminui o comprimento de uma StringBuilder. A Figura 16.11 demonstra esses métodos.

```
1  // Figura 16.11: StringBuilderCapLen.java
2  // Métodos StringBuilder length, setLength, capacity e ensureCapacity.
3
4  public class StringBuilderCapLen
5  {
6     public static void main( String[] args )
7     {
8        StringBuilder buffer = new StringBuilder( "Hello, how are you?" );
9
10       System.out.printf( "buffer = %s\nlength = %d\ncapacity = %d\n\n",
11          buffer.toString(),buffer.length(),buffer.capacity() );
12
13       buffer.ensureCapacity( 75 );
14       System.out.printf( "New capacity = %d\n\n", buffer.capacity() );
15
16       buffer.setLength( 10 );
17       System.out.printf( "New length = %d\nbuffer = %s\n",
18          buffer.length(), buffer.toString() );
19    } // fim de main
20 } // fim da classe StringBuilderCapLen
```

```
buffer = Hello, how are you?
length = 19
capacity = 35

New capacity = 75

New length = 10
buffer = Hello, how
```

Figura 16.11 | Métodos `StringBuilder` `length`, `setLength`, `capacity` e `ensureCapacity`.

O aplicativo contém um `StringBuilder` chamado `buffer`. A linha 8 utiliza o construtor `StringBuilder` que aceita um argumento `String` para inicializar o `StringBuilder` com `"Hello, how are you?"`. As linhas 10–11 imprimem o conteúdo, o comprimento e a capacidade do `StringBuilder`. Note na janela de saída que a capacidade do `StringBuilder` é inicialmente 35. Lembre-se de que o construtor `StringBuilder` que aceita um argumento `String` inicializa a capacidade com o comprimento da string passada como um argumento mais 16.

A linha 13 utiliza o método `ensureCapacity` para expandir a capacidade do `StringBuilder` a um mínimo de 75 caracteres. Na verdade, se a capacidade original for menor que o argumento, o método assegura uma capacidade que é o maior do número especificado como um argumento e duas vezes a capacidade original mais 2. A capacidade atual do `StringBuilder` permanece inalterada se ela for maior do que a capacidade especificada.

Dica de desempenho 16.4

Aumentar a capacidade de um `StringBuilder` dinamicamente pode exigir um tempo relativamente longo. Executar um grande número dessas operações pode degradar o desempenho de um aplicativo. Se o tamanho de um `StringBuilder` vai aumentar significativamente, possivelmente múltiplas vezes, configurar sua capacidade alta no início aumentará o desempenho.

A linha 16 utiliza o método `setLength` para configurar o comprimento do `StringBuilder` como 10. Se o comprimento especificado é menor que o número atual de caracteres no `StringBuilder`, o buffer é truncado para o comprimento especificado (isto é, os caracteres na `StringBuilder` depois do comprimento especificado são descartados). Se o comprimento especificado for maior que o número de caracteres atualmente no `StringBuilder`, caracteres nulos (caracteres com a representação numérica 0) são acrescentados até que o número total de caracteres em `StringBuilder` seja igual ao comprimento especificado.

16.4.3 Métodos `StringBuilder`, `charAt`, `setCharAt`, `getChars` e `reverse`

A classe `StringBuilder` fornece os métodos **`charAt`**, **`setCharAt`**, **`getChars`** e **`reverse`** para manipular os caracteres em um `StringBuilder`(Figura 16.12). O método `charAt` (linha 12) aceita um argumento inteiro e retorna o caractere no `StringBuilder` nesse índice. O método `getChars` (linha 15) copia caracteres de um `StringBuilder` no array de caractere passado como um argumento. Esse método aceita quatro argumentos — o índice inicial a partir do qual caracteres devem ser copiados na `StringBuilder`, o índice um além do último caractere que será copiado a partir do `StringBuilder`, o array de caracteres em que os caracteres serão copiados e a localização inicial no array de caracteres em que o primeiro caractere deve ser colocado. O método `setCharAt` (linhas 21 e 22) aceita um argumento inteiro e um argumento caractere e configura o caractere na posição especificada no `StringBuilder` como o argumento caractere. O método `reverse` (linha 25) inverte o conteúdo do `StringBuilder`.

Erro comum de programação 16.3

Tentar acessar um caractere que está além dos limites de uma `StringBuilder` (isto é, com um índice menor que 0 ou um índice maior que ou igual ao comprimento da `StringBuilder`) resulta em uma `StringIndexOutOfBoundsException`.

```
1  // Figura 16.12: StringBuilderChars.java
2  // Métodos StringBuilder charAt, setCharAt, getChars e reverse.
3
4  public class StringBuilderChars
5  {
6     public static void main( String[] args )
7     {
8        StringBuilder buffer = new StringBuilder( "hello there" );
9
10       System.out.printf( "buffer = %s\n", buffer.toString() );
11       System.out.printf( "Character at 0: %s\nCharacter at 4: %s\n\n",
```

```
            buffer.charAt( 0 ), buffer.charAt( 4 ) );

         char[] charArray = new char[ buffer.length() ];
         buffer.getChars( 0, buffer.length(), charArray, 0 );
         System.out.print( "The characters are: " );

         for ( char character : charArray )
            System.out.print( character );

         buffer.setCharAt( 0, 'H' );
         buffer.setCharAt( 6, 'T' );
         System.out.printf( "\n\nbuffer = %s", buffer.toString() );

         buffer.reverse();
         System.out.printf( "\n\nbuffer = %s\n", buffer.toString() );
      } // fim de main
   } // fim da classe StringBuilderChars
```

```
buffer = hello there
Character at 0: h
Character at 4: o

The characters are: hello there

buffer = Hello There

buffer = erehT olleH
```

Figura 16.12 | Os métodos `StringBuilder` `charAt`, `setCharAt`, `getChars` e `reverse`.

16.4.4 Métodos `StringBuilder` append

A classe `StringBuilder` fornece os métodos **append** sobrecarregados (Figura 16.13) para permitir que valores de vários tipos sejam acrescentados no fim de um `StringBuilder`. Versões são fornecidas para cada um dos tipos primitivos, e para arrays de caractere, `Strings`, `Objects` e mais. (Lembre-se de que o método `toString` produz uma representação de string de qualquer `Object`.) Cada método recebe seu argumento, converte-o em uma string e a acrescenta a `StringBuilder`.

```
// Figura 16.13: StringBuilderAppend.java
// Métodos append de StringBuilder

public class StringBuilderAppend
{
   public static void main( String[] args )
   {
      Object objectRef = "hello";
      String string = "goodbye";
      char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
      boolean booleanValue = true;
      char characterValue = 'Z';
      int integerValue = 7;
      long longValue = 10000000000L;
      float floatValue = 2.5f;
      double doubleValue = 33.333;

      StringBuilder lastBuffer = new StringBuilder( "last buffer" );
      StringBuilder buffer = new StringBuilder();

      buffer.append( objectRef );
      buffer.append( "\n" );
      buffer.append( string );
      buffer.append( "\n" );
      buffer.append( charArray );
      buffer.append( "\n" );
      buffer.append( charArray, 0, 3 );
      buffer.append( "\n" );
      buffer.append( booleanValue );
      buffer.append( "\n" );
      buffer.append( characterValue );
```

```
32          buffer.append( "\n" );
33          buffer.append( integerValue );
34          buffer.append( "\n" );
35          buffer.append( longValue );
36          buffer.append( "\n" );
37          buffer.append( floatValue );
38          buffer.append( "\n" );
39          buffer.append( doubleValue );
40          buffer.append( "\n" );
41          buffer.append( lastBuffer );
42
43          System.out.printf( "buffer contains %s\n", buffer.toString() );
44       } // fim de main
45    } // termina StringBuilderAppend
```

```
buffer contains hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

Figura 16.13 | Métodos `StringBuilder` `append`.

De fato, um compilador pode utilizar os métodos `StringBuilder` (ou `StringBuffer`) e `append` para implementar os operadores de concatenação de `String` + e += . Por exemplo, considerando as declarações

```
String string1 = "hello";
String string2 = "BC";
int value = 22;
```

a instrução

```
String s = string1 + string2 + value;
```

concatena `"hello"`, `"BC"` e `22`. A concatenação pode ser realizada como mostrado a seguir:

```
String s = new StringBuilder().append( "hello" ).append( "BC" ).
   append( 22 ).toString();
```

Em primeiro lugar, a instrução anterior cria um `StringBuilder` vazio, depois acrescenta-lhe as strings `"hello"` e `"BC"` e o número inteiro `22`. Em seguida, o método `toString` de `StringBuilder` converte o objeto `StringBuilder` em uma `String` a ser atribuída à `String` `s`. A instrução

```
s += "!";
```

pode ser realizada desta maneira:

```
s = new StringBuilder().append( s ).append( "!" ).toString();
```

Primeiro, a instrução anterior cria um `StringBuilder` vazio, depois acrescenta-lhe o conteúdo atual de `s` seguido por `"!"`. Em seguida, o método `toString` de `StringBuilder` converte o `StringBuilder` em uma representação de string e o resultado é atribuído à `s`.

16.4.5 Métodos de inserção e exclusão de `StringBuilder`

A classe `StringBuilder` fornece os métodos **`insert`** sobrecarregados para inserir valores de vários tipos em qualquer posição em um `StringBuilder`. As versões são oferecidas para os tipos primitivos e para os arrays de caracteres, `Strings`, `Objects` e `CharSequences`. Cada método aceita seu segundo argumento, converte-o em uma `String` e o insere no índice especificado pelo primeiro argumento. Se o primeiro argumento menor que `0` ou maior que o comprimento `StringBuilder`, uma `StringIndexOutOfBoundsException` ocorre. A classe `StringBuilder` também fornece métodos **`delete`** e **`deleteCharAt`** para excluir caracteres em qualquer posição em um `StringBuilder`. O método `delete` — aceita dois argumentos — o índice inicial e o índice um além do fim dos caracteres a excluir. Todos os caracteres que começam no índice inicial, mas não incluindo o índice final, são excluídos. O método `deleteCharAt` aceita um argumento — o índice do caractere a excluir. Os índices inválidos fazem com que ambos os métodos lancem uma `StringIndexOutOfBoundsException`. A Figura 16.14 demonstra os métodos `insert`, `delete` e `deleteCharAt`.

```
1  // Figura 16.14: StringBuilderInsertDelete.java
2  // Métodos StringBuilder insert, delete e deleteCharAt.
3
4  public class StringBuilderInsertDelete
5  {
6     public static void main( String[] args )
7     {
8        Object objectRef = "hello";
9        String string = "goodbye";
10       char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
11       boolean booleanValue = true;
12       char characterValue = 'K';
13       int integerValue = 7;
14       long longValue = 10000000;
15       float floatValue = 2.5f; // o sufixo f indica que 2.5 é um tipo float
16       double doubleValue = 33.333;
17
18       StringBuilder buffer = new StringBuilder();
19
20       buffer.insert( 0, objectRef );
21       buffer.insert( 0, "  " ); // cada um desses contém dois espaços
22       buffer.insert( 0, string );
23       buffer.insert( 0, "  " );
24       buffer.insert( 0, charArray );
25       buffer.insert( 0, "  " );
26       buffer.insert( 0, charArray, 3, 3 );
27       buffer.insert( 0, "  " );
28       buffer.insert( 0, booleanValue );
29       buffer.insert( 0, "  " );
30       buffer.insert( 0, characterValue );
31       buffer.insert( 0, "  " );
32       buffer.insert( 0, integerValue );
33       buffer.insert( 0, "  " );
34       buffer.insert( 0, longValue );
35       buffer.insert( 0, "  " );
36       buffer.insert( 0, floatValue );
37       buffer.insert( 0, "  " );
38       buffer.insert( 0, doubleValue );
39
40       System.out.printf(
41          "buffer after inserts:\n%s\n\n", buffer.toString() );
42
43       buffer.deleteCharAt( 10 ); // exclui 5 em 2.5
44       buffer.delete( 2, 6 ); // exclui .333 in 33.333
45
46       System.out.printf(
47          "buffer after deletes:\n%s\n", buffer.toString() );
48    } // fim de main
49 } // fim da classe StringBuilderInsertDelete
```

```
buffer after inserts:
33.333  2.5  10000000  7  K  true  def  abcdef  goodbye  hello

buffer after deletes:
33  2.  10000000  7  K  true  def  abcdef  goodbye  hello
```

Figura 16.14 | Os métodos `StringBuilder` `insert`, `delete` e `deleteCharAt`.

16.5 Classe Character

O Java fornece oito classes do tipo wrapper — `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` e `Long` — que permite que valores de tipo primitivo sejam tratados como objetos. Nesta seção, apresentamos a classe `Character` — a classe do tipo wrapper para o tipo primitivo `char`.

A maioria dos métodos `Character` são métodos `static` projetados por uma questão de conveniência no processamento de valores `char` individuais. Esses métodos aceitam pelo menos um argumento caractere e realizam um teste ou uma manipulação do caractere. A classe `Character` também contém um construtor que recebe um argumento `char` para inicializar um objeto `Character`. A maioria dos

métodos da classe `Character` é apresentada nos próximos três exemplos. Para mais informações sobre a classe `Character` (e todas as classes empacotadoras de tipo), veja o pacote `java.lang` na documentação do Java API.

A Figura 16.15 demonstra métodos `static` que testam caracteres para determinar se eles são um tipo específico de caractere e os métodos `static` que realizam as conversões de caracteres em letras maiúsculas/minúsculas. Você pode inserir qualquer caractere e aplicar os métodos ao caractere.

```
1  // Figura 16.15: StaticCharMethods.java
2  // Métodos Character static para testar caracteres e converter letras maiúsculas e minúsculas.
3  import java.util.Scanner;
4
5  public class StaticCharMethods
6  {
7     public static void main( String[] args )
8     {
9        Scanner scanner = new Scanner( System.in ); // cria scanner
10       System.out.println( "Enter a character and press Enter" );
11       String input = scanner.next();
12       char c = input.charAt( 0 ); // obtém caractere de entrada
13
14       // exibe informações de caractere de
15       System.out.printf( "is defined: %b\n", Character.isDefined( c ) );
16       System.out.printf( "is digit: %b\n", Character.isDigit( c ) );
17       System.out.printf( "is first character in a Java identifier: %b\n",
18          Character.isJavaIdentifierStart( c ) );
19       System.out.printf( "is part of a Java identifier: %b\n",
20          Character.isJavaIdentifierPart( c ) );
21       System.out.printf( "is letter: %b\n", Character.isLetter( c ) );
22       System.out.printf(
23          "is letter or digit: %b\n", Character.isLetterOrDigit( c ) );
24       System.out.printf(
25          "is lower case: %b\n", Character.isLowerCase( c ) );
26       System.out.printf(
27          "is upper case: %b\n", Character.isUpperCase( c ) );
28       System.out.printf(
29          "to upper case: %s\n", Character.toUpperCase( c ) );
30       System.out.printf(
31          "to lower case: %s\n", Character.toLowerCase( c ) );
32    } // fim de main
33 } // StaticCharMethods fim da classe
```

```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

```
Enter a character and press Enter
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

Figura 16.15 | Métodos `Character` `static` para testar caracteres e converter caracteres maiúsculos e minúsculos.

A linha 15 utiliza o método `Character` **`isDefined`** para determinar se o caractere `c` está definido no conjunto de caracteres Unicode. Se estiver, o método retorna `true`; caso contrário, retorna `false`. A linha 16 utiliza o método `Character` **`isDigit`** para determinar se o caractere `c` é um dígito Unicode definido. Se for, o método retorna `true` e caso contrário, `false`.

A linha 18 utiliza o método `Character` **`isJavaIdentifierStart`** para determinar se `c` é um caractere que pode ser o primeiro caractere de um identificador em Java — isto é, uma letra, um sublinhado (_) ou um sinal de cifrão ($). Se for, o método retorna `true` e caso contrário, `false`. A linha 20 utiliza o método `Character` **`isJavaIdentifierPart`** para determinar se o caractere `c` é um caractere que pode ser usado em um identificador em Java — isto é, um dígito, uma letra, um sublinhado (_) ou um sinal de cifrão ($). Se for, o método retorna `true` e caso contrário, `false`.

A linha 21 utiliza o método `Character` **`isLetter`** para determinar se o caractere `c` é uma letra. Se for, o método retorna `true` e caso contrário, `false`. A linha 23 utiliza o método `Character` **`isLetterOrDigit`** para determinar se o caractere `c` é uma letra ou um dígito. Se for, o método retorna `true` e caso contrário, `false`.

A linha 25 utiliza o método `Character` **`isLowerCase`** para determinar se o caractere `c` é uma letra minúscula. Se for, o método retorna `true` e caso contrário, `false`. A linha 27 utiliza o método `Character` **`isUpperCase`** para determinar se o caractere `c` é uma letra maiúscula. Se for, o método retorna `true` e caso contrário, `false`.

A linha 29 utiliza o método `Character` **`toUpperCase`** para converter o caractere `c` em seu equivalente em letras maiúsculas. O método retorna o caractere convertido se o caractere tiver um equivalente em letras maiúsculas; caso contrário, o método retorna seu argumento original. A linha 31 utiliza o método `Character` **`toLowerCase`** para converter o caractere `c` em seu equivalente em letras minúsculas. O método retorna o caractere convertido se o caractere tem um equivalente em letras minúsculas e, caso contrário, o método retorna seu argumento original.

A Figura 16.16 demonstra os métodos static `Character` **`digit`** e **`forDigit`**, que convertem os caracteres em dígitos e dígitos em caracteres, respectivamente, em diferentes sistemas de números. Sistemas comuns de número incluem decimal (base 10), octal (base 8), hexadecimal (base 16) e binário (base 2). A base de um número também é conhecida como sua **radical**. Para mais informações sobre conversões entre sistemas de números, veja o Apêndice H.

```
1  // Figura 16.16: StaticCharMethods2.java
2  // Métodos de conversão static da classe Character.
3  import java.util.Scanner;
4
5  public class StaticCharMethods2
6  {
7     // executa o aplicativo
8     public static void main( String[] args )
9     {
10       Scanner scanner = new Scanner( System.in );
11
12       // obtém radical
13       System.out.println( "Please enter a radix:" );
14       int radix = scanner.nextInt();
15
16       // obtém escolha de usuário
17       System.out.printf( "Please choose one:\n1 -- %s\n2 -- %s\n",
18          "Convert digit to character", "Convert character to digit" );
19       int choice = scanner.nextInt();
20
21       // processa solicitação
22       switch ( choice )
23       {
```

```
24            case 1: // converte dígito em caractere
25               System.out.println( "Enter a digit:" );
26               int digit = scanner.nextInt();
27               System.out.printf( "Convert digit to character: %s\n",
28                  Character.forDigit( digit, radix ) );
29               break;
30
31            case 2: // converte caractere em dígito
32               System.out.println( "Enter a character:" );
33               char character = scanner.next().charAt( 0 );
34               System.out.printf( "Convert character to digit: %s\n",
35                  Character.digit( character, radix ) );
36               break;
37         } // fim do switch
38      } // fim de main
39   } // fim da classe StaticCharMethods2
```

```
Please enter a radix:
16
Please choose one:
1 -- Convert digit to character
2 -- Convert character to digit
2
Enter a character:
A
Convert character to digit: 10
```

```
Please enter a radix:
16
Please choose one:
1 -- Convert digit to character
2 -- Convert character to digit
1
Enter a digit:
13
Convert digit to character: d
```

Figura 16.16 | Métodos de conversão `static` da classe `Character`.

A linha 28 método utiliza `forDigit` para converter o inteiro `digit` em um caractere no sistema de números especificado pelo inteiro `radix` (a base do número). Por exemplo, o inteiro decimal `13` na base 16 (o `radix`) tem o valor de caractere `'d'`. Letras minúsculas e maiúsculas representam o mesmo valor em sistemas numéricos. A linha 35 utiliza o método `digit` para converter a variável `character` em um inteiro no sistema numérico especificado pelo inteiro `radix` (a base do número). Por exemplo, o caractere `'A'` é a representação base 16 (o `radix`) do valor 10 de base 10. O radical deve estar entre 2 e 36, inclusive.

A Figura 16.17 demonstra o construtor e vários métodos não `static` da classe `Character` — **`charValue`**, `toString` e `equals`. As linhas 7–8 instanciam dois objetos `Character` atribuindo as constantes de caractere `'A'` e `'a'`, respectivamente, às variáveis `Character`. O Java converte automaticamente esses literais `char` em objetos `Character` — um processo conhecido como autoboxing que discutimos mais detalhadamente na Seção 20.4. A linha 11 utiliza o método `Character` `charValue` para retornar o valor `char` armazenado no objeto `Character` `c1`. A linha 11 retorna uma representação de string do objeto `Character` `c2` utilizando o método `toString`. A condição na linha 13 utiliza o método `equals` para determinar se o objeto `c1` tem o mesmo conteúdo que o objeto `c2` (isto é, os caracteres dentro de cada objeto são iguais).

```
1  // Figura 16.17: OtherCharMethods.java
2  // Métodos não static da classe Character.
3  public class OtherCharMethods
4  {
5     public static void main( String[] args )
6     {
7        Character c1 = 'A';
8        Character c2 = 'a';
9
```

```
10          System.out.printf(
11             "c1 = %s\nc2 = %s\n\n", c1.charValue(),c2.toString() );
12
13          if (c1.equals( c2 ) )
14             System.out.println( "c1 and c2 are equal\n" );
15          else
16             System.out.println( "c1 and c2 are not equal\n" );
17       } // fim de main
18    } // fim da classe OtherCharMethods
```

```
c1 = A
c2 = a

c1 and c2 are not equal
```

Figura 16.17 | Método não `static` da classe `Character`.

16.6 Tokenização de Strings

Quando você lê uma frase, sua mente a divide em **tokens** — palavras individuais e sinais de pontuação que lhe transmitem significado. Os compiladores também realizam "tokenização". Eles dividem instruções em pedaços individuais, como palavras-chave, identificadores, operadores e outros elementos de linguagem de programação. Agora estudaremos o método **`split`** da classe `String`, que divide uma `String` em seus tokens componentes. Os tokens são separados entre si por **delimitadores**, em geral caracteres de espaçamento como espaço, tabulação, nova linha e retorno de carro. Outros caracteres também podem ser utilizados como delimitadores para separar tokens. O aplicativo na Figura 16.18 demonstra o método `split` de `String`.

```
1  // Figura 16.18: TokenTest.java
2  // Objeto StringTokenizer utilizado para tokenizar strings.
3  import java.util.Scanner;
4  import java.util.StringTokenizer;
5
6  public class TokenTest
7  {
8     // executa o aplicativo
9     public static void main( String[] args )
10    {
11       // obtém a frase
12       Scanner scanner = new Scanner( System.in );
13       System.out.println( "Enter a sentence and press Enter" );
14       String sentence = scanner.nextLine();
15
16       // processa a frase do usuário
17       String[] tokens = sentence.split( " " );
18       System.out.printf( "Number of elements: %d\nThe tokens are:\n",
19          tokens.length );
20
21       for ( String token : tokens )
22          System.out.println( token );
23    } // fim de main
24 } // fim da classe TokenTest
```

```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of elements: 7
The tokens are:
This
is
a
sentence
with
seven
tokens
```

Figura 16.18 | O objeto `StringTokenizer` utilizado para tokenizar strings.

Quando o usuário pressiona a tecla *Enter*, a frase de entrada é armazenada na variável `sentence`. A linha 17 invoca o método `String` `split` com o argumento `String` `" "`, que retorna um array de `String`s. O caractere de espaço no argumento `String` é o delimitador que o método `split` utiliza para localizar os tokens na `String`. Como você aprenderá na próxima seção, o argumento para o método split pode ser uma expressão regular para tokenização mais complexa. A linha 19 exibe o tamanho do array `tokens` — isto é, o número de tokens em `sentence`. As linhas 21–22 geram saída de cada token em uma linha separada.

16.7 Expressões regulares, classe `Pattern` e classe `Matcher`

Uma **expressão regular** é uma `String` especialmente formatada que descreve um padrão de pesquisa para correspondência de caracteres em outras `String`s. Elas são úteis para validar a entrada e assegurar que os dados estão em determinado formato. Por exemplo, um CEP deve consistir em cinco dígitos e um sobrenome deve conter somente letras, espaços, apóstrofos e hífens. Um aplicativo de expressões regulares serve para facilitar a construção de um compilador. Frequentemente, uma expressão regular grande e complexa é utilizada para validar a sintaxe de um programa. Se o código do programa não localizar expressão regular, o compilador sabe que há um erro de sintaxe dentro do código.

A classe `String` fornece vários métodos para realizar operações de expressão regular, das quais a mais simples é a operação de correspondência. O método `String` **`matches`** recebe uma `String` que especifica a expressão regular e localiza o conteúdo do objeto `String` em que ele é chamado na expressão regular. O método retorna um `boolean` indicando se a correspondência foi ou não bem-sucedida.

Uma expressão regular consiste em caracteres literais e símbolos especiais. A Figura 16.19 especifica algumas **classes de caractere predefinidas** que podem ser utilizadas com expressões regulares. Uma classe de caracteres é uma sequência de escape que representa um grupo de caracteres. Um dígito é qualquer caractere numérico. Um **caractere de palavra** é qualquer letra (em letras maiúsculas ou minúsculas), qualquer dígito ou o caractere sublinhado. Um caractere de espaço em branco é um espaço, uma tabulação, um retorno de carro, um caractere de nova linha ou um avanço de formulário. Cada classe de caracteres localiza a um único caractere na `String` que estamos tentando localizar com a expressão regular.

Caractere	Correspondências	Caractere	Correspondências
`\d`	qualquer dígito	`\D`	qualquer não dígito
`\w`	qualquer caractere de palavra	`\W`	qualquer caractere não palavra
`\s`	qualquer caractere de espaço em branco	`\S`	qualquer caractere não espaço em branco

Figura 16.19 | Classes predefinidas de caractere.

As expressões regulares não estão limitadas a essas classes predefinidas de caractere. As expressões empregam vários operadores e outras formas de notação para localizar padrões complexos. Examinamos várias dessas técnicas no aplicativo das figuras 16.20 e 16.21 que valida a entrada de usuário por meio de expressões regulares. [*Nota*: Esse aplicativo não é projetado para localizar todas as possíveis entradas de usuário válidas].

```
1  // Figura 16.20: ValidateInput.java
2  // Valida informações de usuário utilizando expressões regulares.
3
4  public class ValidateInput
5  {
6     // valida o primeiro nome
7     public static boolean validateFirstName( String firstName )
8     {
9        return firstName.matches( "[A-Z][a-zA-Z]*" );
10    } // fim do método validateFirstName
11
12    // valida o sobrenome
13    public static boolean validateLastName( String lastName )
14    {
15       return lastName.matches( "[a-zA-z]+([ '-][a-zA-Z]+)*" );
16    } // fim do método validateLastName
17
18    // valida o endereço
19    public static boolean validateAddress( String address )
20    {
21       return address.matches(
```

```
22          "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
23       } // fim do método validateAddress
24
25       // valida cidade
26       public static boolean validateCity( String city )
27       {
28          return city.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
29       } // fim do método validateCity
30
31       // valida estado
32       public static boolean validateState( String state )
33       {
34          return state.matches( "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" ) ;
35       } // fim do método validateState
36
37       // valida CEP
38       public static boolean validateZip( String zip )
39       {
40          return zip.matches( "\\d{5}" );
41       } // fim do método validateZip
42
43       // valida telefone
44       public static boolean validatePhone( String phone )
45       {
46          return phone.matches( "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
47       } // fim do método validatePhone
48    } // fim da classe ValidateInput
```

Figura 16.20 | Validando informações de usuário com expressões regulares.

```
1  // Figura 16.21: Validate.java
2  // Valida informações de usuário utilizando expressões regulares.
3  import java.util.Scanner;
4
5  public class Validate
6  {
7     public static void main( String[] args )
8     {
9        // obtém entrada de usuário
10       Scanner scanner = new Scanner( System.in );
11       System.out.println( "Please enter first name:" );
12       String firstName = scanner.nextLine();
13       System.out.println( "Please enter last name:" );
14       String lastName = scanner.nextLine();
15       System.out.println( "Please enter address:" );
16       String address = scanner.nextLine();
17       System.out.println( "Please enter city:" );
18       String city = scanner.nextLine();
19       System.out.println( "Please enter state:" );
20       String state = scanner.nextLine();
21       System.out.println( "Please enter zip:" );
22       String zip = scanner.nextLine();
23       System.out.println( "Please enter phone:" );
24       String phone = scanner.nextLine();
25
26       // valida entrada de usuário e exibe mensagem de erro
27       System.out.println( "\nValidate Result:" );
28
29       if ( !ValidateInput.validateFirstName( firstName ) )
30          System.out.println( "Invalid first name" );
31       else if ( !ValidateInput.validateLastName( lastName ) )
32          System.out.println( "Invalid last name" );
33       else if ( !ValidateInput.validateAddress( address ) )
34          System.out.println( "Invalid address" );
35       else if ( !ValidateInput.validateCity( city ) )
36          System.out.println( "Invalid city" );
```

```
37          else if ( !ValidateInput.validateState( state ) )
38             System.out.println( "Invalid state" );
39          else if ( !ValidateInput.validateZip( zip ) )
40             System.out.println( "Invalid zip code" );
41          else if ( !ValidateInput.validatePhone( phone ) )
42             System.out.println( "Invalid phone number" );
43          else
44             System.out.println( "Valid input.  Thank you." );
45       } // fim de main
46    } // fim da classe Validate
```

```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
123
Please enter phone:
123-456-7890

Validate Result:
Invalid zip code
```

```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS

Please enter zip:
12345
Please enter phone:
123-456-7890

Validate Result:
Valid input.  Thank you.
```

Figura 16.21 | Insere e valida os dados de usuário utilizando a classe `ValidateInput`.

A Figura 16.20 valida a entrada de usuário. A linha 9 valida o primeiro nome. Para localizar um conjunto de caracteres que não tem uma classe de caracteres predefinida, utilize os colchetes, `[]`. Por exemplo, o padrão `"[aeiou]"` localiza um único caractere que é uma vogal. Os intervalos de caractere são representados colocando um traço (`-`) entre dois caracteres. No exemplo, `"[A-Z]"` identifica uma única letra maiúscula. Se o primeiro caractere entre colchetes for `"^"`, a expressão aceitará qualquer caractere diferente desses indicados. Entretanto, é importante observar que `"[^Z]"` não é o mesmo que `"[A-Y]"`, que localiza todas as letras maiúsculas no intervalo A–Y — `"[^Z]"` localiza qualquer caractere diferente da letra Z maiúscula, incluindo as letras minúsculas e não letras como o caractere de nova linha. Os intervalos nas classes de caractere são determinados pelos valores inteiros das letras. Neste exemplo, `"[A-Za-z]"` localiza todas as letras maiúsculas e minúsculas. O intervalo `"[A-z]"` localiza todas as letras e também aqueles caracteres (como [e \) com um valor inteiro entre o Z maiúsculo e o *a* minúsculo (para obter informações adicionais sobre valores inteiros de caracteres, consulte o Apêndice B). Como as classes predefinidas de caractere, as classes de caractere delimitadas por colchetes localizam um único caractere no objeto de pesquisa.

Na linha 9, o asterisco depois da segunda classe de caracteres indica que pode haver correspondência com qualquer número de letras. Em geral, quando o operador de expressão regular "*" aparece em uma expressão regular, o aplicativo tenta localizar a zero ou mais ocorrências da subexpressão imediatamente anterior a "*". O operador "+" tenta identificar uma ou mais ocorrências da subexpressão imediatamente anterior "+". Assim tanto "A*" como "A+" localizará "AAA" ou "A", mas apenas "A*" identificará uma string vazia.

Se o método validateFirstName retorna true (linha 29 da Figura 16.21), o aplicativo tenta validar o sobrenome (linha 31) chamando validateLastName (linhas 13–16 da Figura 16.20). A expressão regular para validar o sobrenome localiza qualquer número de letras separadas por espaços, apóstrofos ou hífens.

A linha 33 da Figura 16.21 chama o método validateAddress (linhas 19–23 da Figura 16.20) para validar o endereço. A primeira classe de caracteres localiza qualquer dígito uma ou mais vezes (\\d+). Observe que os dois caracteres \ são utilizados, pois \ normalmente inicia uma sequência de escape em uma string. Assim \\ Em seguida, pesquisamos um ou mais caracteres de espaço em branco (\\s+). O caractere "|" localiza a expressão à sua esquerda ou direita. Por exemplo, "Hi (John|Jane)" localiza tanto a "Hi John" como "Hi Jane". Os parênteses são utilizados para agrupar partes da expressão regular. Neste exemplo, o lado esquerdo de | localiza uma única palavra e, o direito, a duas palavras separadas por qualquer quantidade de espaço em branco. Assim o endereço deve conter um número seguido por uma ou duas palavras. Portanto, "10 Broadway" e "10 Main Street" são ambos endereços válidos nesse exemplo. Os métodos city (linhas 26–29 da Figura 16.20) e state (linhas 32–35 da Figura 16.20) também procuram qualquer palavra de pelo menos um caractere ou, alternativamente, quaisquer duas palavras de pelo menos um caractere se as palavras forem separadas por um único espaço, assim ambos Waltham e West Newton devem corresponder.

Quantificadores

Os sinais de asterisco (*) e de adição (+) são formalmente chamados de **quantificadores**. A Figura 16.22 lista todos os quantificadores. Já discutimos como os quantificadores de asterisco (*) e sinal de adição (+) funcionam. Todos os quantificadores afetam apenas a subexpressão imediatamente anterior ao quantificador. O ponto de interrogação quantificador (?) localiza zero ou uma ocorrência da expressão que ele quantifica. Um conjunto de chaves contendo um número ({*n*}) localiza exatamente *n* ocorrências da expressão que ele quantifica. Demonstramos esse quantificador para validar o CEP da Figura 16.20 na linha 40. Incluir uma vírgula depois do número incluído entre chaves localiza chaves pelo menos *n* ocorrências da expressão quantificada. O conjunto de chaves que contém dois números ({*n*,*m*}), localiza entre *n* e *m* ocorrências da expressão que ele qualifica. Os quantificadores podem ser aplicados a padrões entre parênteses para criar expressões regulares mais complexas.

Quantificador	Correspondências
*	Localiza a zero ou mais ocorrências do padrão.
+	Localiza a uma ou mais ocorrências do padrão.
?	Localiza a zero ou uma ocorrência do padrão.
{*n*}	Localiza exatamente *n* ocorrências.
{*n*,}	Localiza pelo menos *n* ocorrências.
{*n*, *m*}	Localiza entre *n* e *m* (inclusive) ocorrências.

Figura 16.22 | Quantificadores utilizados em expressões regulares.

Todos os quantificadores são **gananciosos**. Isso significa que eles identificarão quantas ocorrências eles puderem contanto que a correspondência ainda seja bem-sucedida. Entretanto, se qualquer um desses quantificadores for seguido por um ponto de interrogação (?), o quantificador se tornará **relutante** (às vezes chamado de **preguiçoso**). Então ele reconhecerá o mínimo de ocorrências possível contanto que a correspondência ainda seja bem-sucedida.

O CEP (linha 40 na Figura 16.20) localiza cinco vezes um dígito. Essa expressão regular utiliza a classe de caracteres de dígito e um quantificador com o dígito 5 entre chaves. O número de telefone (linha 46 na Figura 16.20) combina com três dígitos (o primeiro não pode ser zero) seguidos por um traço seguido por mais três dígitos (novamente o primeiro não pode ser zero) seguido por mais quatro dígitos.

O método String matches verifica se uma string inteira se adapta a uma expressão regular. Por exemplo, queremos aceitar "Smith" como um sobrenome, mas não "9@Smith#". Se ao menos uma substring corresponder à expressão regular, o método matches retorna false.

Substituindo substrings e dividindo strings

Às vezes é útil substituir partes de uma string ou dividir uma string em partes. Para esse propósito, a classe `String` fornece os métodos **`replaceAll`**, **`replaceFirst`** e **`split`**. Esses métodos são demonstrados na Figura 16.23.

```
1  // Figura 16.23: RegexSubstitution.Java
2  // Métodos string ReplaceFirst, replaceAll e split.
3  import java.util.Arrays;
4
5  public class RegexSubstitution
6  {
7     public static void main( String[] args )
8     {
9        String firstString = "This sentence ends in 5 stars *****";
10       String secondString = "1, 2, 3, 4, 5, 6, 7, 8";
11
12       System.out.printf( "Original String 1: %s\n", firstString );
13
14       // substitui '*' por '^'
15       firstString = firstString.replaceAll( "\\*", "^" );
16
17       System.out.printf( "^ substituted for *: %s\n", firstString );
18
19       // substitui asteriscos por circunflexos
20       firstString = firstString.replaceAll( "stars", "carets" );
21
22       System.out.printf(
23          "\"carets\" substituted for \"stars\": %s\n", firstString );
24
25       // substitui palavras por 'palavra'
26       System.out.printf( "Every word replaced by \"word\": %s\n\n",
27          firstString.replaceAll( "\\w+", "word" ) );
28
29       System.out.printf( "Original String 2: %s\n", secondString );
30
31       // substitui primeiros três dígitos pelo 'dígito'
32       for ( int i = 0; i < 3; i++ )
33          secondString = secondString.replaceFirst( "\\d", "digit" );
34
35       System.out.printf(
36          "First 3 digits replaced by \"digit\" : %s\n", secondString );
37
38       System.out.print( "String split at commas: " );
39       String[] results = secondString.split( ",\\s*" ); // divide em vírgulas
40       System.out.println( Arrays.toString( results ) );
41    } // fim de main
42 } // fim da classe RegexSubstitution
```

```
Original String 1: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original String 2: 1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" : digit, digit, digit, 4, 5, 6, 7, 8
String split at commas: ["digit", "digit", "digit", "4", "5", "6", "7", "8"]
```

Figura 16.23 | Métodos `String` `replaceFirst`, `replaceAll` e `split`.

O método `replaceAll` substitui texto em uma `String` com texto novo (o segundo argumento) onde quer que a string original localize uma expressão regular (o primeiro argumento). A linha 15 substitui cada instância de `"*"` em `firstString` por `"^"`. Observe que a expressão regular (`"\\*"`) precede o caractere `*` com duas barras invertidas. Normalmente, `*` é um quantificador que indica que uma expressão regular deve corresponder a qualquer número de ocorrências de um padrão anterior. Entretanto, na linha 15, queremos combinar todas as ocorrências do caractere literal `*` — para fazer isso, devemos escapar o caractere `*` pelo caractere `\`. Escapar um caractere de expressão regular especial com `\` instrui o mecanismo de correspondência a localizar o caractere real. Visto que a expressão é armazenada em uma `String` Java e `\` é um caractere especial em `String`s Java, devemos incluir um `\` adicional. Então a `String` `"\\*"` do Java representa

o padrão de expressão regular * que identifica um único caractere * na string de pesquisa. Na linha 20, cada correspondência da expressão regular "stars" em firstString é substituída por "carets". A linha 27 utiliza replaceAll para substituir todas as palavras na string por "word".

O método replaceFirst (linha 33) substitui a primeira ocorrência de uma correspondência de padrão. Strings do Java são imutáveis, portanto o método replaceFirst retorna uma nova String em que os caracteres apropriados foram substituídos. Essa linha aceita a String original e a substitui pela String retornada por replaceFirst. Iterando três vezes substituímos as três primeiras instâncias de um dígito (\d) em secondString pelo texto "digit".

O método split divide uma String em várias substrings. A original é dividida em qualquer localização que corresponde a uma expressão regular especificada. O método split retorna um array de Strings que contém as substrings entre as correspondências da expressão regular. Na linha 39, utilizamos o método split para tokenizar uma String de inteiros separados por vírgulas. O argumento é a expressão regular que corresponde ao delimitador. Nesse caso, utilizamos a expressão regular "\,\\s*" para separar as substrings onde quer que ocorra uma vírgula. Correspondendo quaisquer caracteres espaço em branco, eliminamos os espaços extras das substrings resultantes. Observe que as vírgulas e os caracteres de espaço em branco não são retornados como parte das substrings. Novamente, observe que a String "\,\\s*" do Java representa a expressão regular \,\s*. A linha 40 utiliza o método Arrays toString para exibir os conteúdos do array results em colchetes e separado por vírgulas.

Classes Pattern e Matcher

Além das capacidades de expressão regular da classe String, o Java fornece outras classes no pacote java.util.regex que ajudam os desenvolvedores a manipular expressões regulares. A classe **Pattern** representa uma expressão regular. A classe **Matcher** contém tanto um padrão de expressão regular como uma CharSequence na qual procurar o padrão.

CharSequence (pacote java.lang) é uma interface que permite acesso de leitura a uma sequência de caracteres. A interface exige que os métodos charAt, length, subSequence e toString sejam declarados. Tanto String como StringBuilder implementam a interface CharSequence, portanto uma instância de qualquer dessas classes pode ser utilizada com a classe Matcher.

Erro comum de programação 16.4

Uma expressão regular pode ser testada contra um objeto de qualquer classe que implemente a interface CharSequence, mas a expressão regular deve ser uma String. Tentar criar uma expressão regular como um StringBuilder é um erro.

Se uma expressão regular vai ser utilizada apenas uma vez, o método static Pattern **matches** pode ser utilizado. Esse método aceita uma String que especifica a expressão regular e um CharSequence em que realiza a correspondência. Esse método retorna um boolean que indica se o objeto de pesquisa (o segundo argumento) corresponde à expressão regular.

Se uma expressão regular vai ser utilizada mais de uma vez (em um loop por exemplo), é mais eficiente utilizar o método static Pattern **compile** para criar um objeto Pattern específico para essa expressão regular. Esse método recebe uma String que representa o padrão e retorna um novo objeto Pattern, que então pode ser utilizado para chamar o método **matcher**. Esse método recebe um CharSequence para procurar e retornar um objeto Matcher.

Matcher fornece método **matches**, que realiza a mesma tarefa que o método Pattern matches, mas não recebe nenhum argumento — o padrão de pesquisa e o objeto de pesquisa são encapsulados no objeto Matcher. A classe Matcher fornece outros métodos, incluindo **find**, **lookingAt**, **replaceFirst** e **replaceAll**.

A Figura 16.24 apresenta um exemplo simples que emprega expressões regulares. Esse programa identifica aniversários em uma expressão regular. A expressão identifica somente aniversários que não ocorram em abril e que pertençam às pessoas cujos nomes iniciam com "J".

As linhas 11–12 criam um Pattern invocando o método static Pattern compile. O caractere de ponto "." na expressão regular (linha 12) procura qualquer caractere único exceto um caractere de nova linha.

```
1  // Figura 16.24: RegexMatches.java
2  // as classes Pattern e Matcher.
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class RegexMatches
7  {
8     public static void main( String[] args )
9     {
10       // cria expressão regular
11       Pattern expression =
12          Pattern.compile( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
13
14       String string1 = "Jane's Birthday is 05-12-75\n" +
15          "Dave's Birthday is 11-04-68\n" +
```

```
16            "John's Birthday is 04-28-73\n" +
17            "Joe's Birthday is 12-17-77";
18
19         // corresponde expressão regular à string e imprime as correspondências
20         Matcher matcher = expression.matcher( string1 );
21
22         while ( matcher.find() )
23            System.out.println( matcher.group() );
24      } // fim de main
25   } // fim da classe RegexMatches
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Figura 16.24 | Classes `Pattern` e `Matcher`.

A linha 20 cria o objeto `Matcher` para a expressão regular compilada e a sequência de correspondência (`string1`). As linhas 22–23 utilizam um loop `while` para iterar pela `String`. A linha 22 utiliza o método `Matcher` `find` para tentar corresponder uma parte do objeto de pesquisa ao padrão de pesquisa. Cada chamada para esse método inicia no ponto em que a última chamada terminou, então múltiplas correspondências podem ser localizadas. O método `Matcher` `lookingAt` executa da mesma maneira, exceto que sempre desde o início do objeto de pesquisa e sempre localizará a primeira correspondência se houver uma.

Erro comum de programação 16.5

O método `matches` (da classe `String`, `Pattern` ou `Matcher`) retornará `true` somente se o objeto de pesquisa inteiro corresponder à expressão regular. Os métodos `find` e `lookingAt` (da classe `Matcher`) retornarão `true` se uma parte do objeto de pesquisa corresponder à expressão regular.

A linha 23 utiliza o método `Matcher` **group**, que retorna a `String` do objeto de pesquisa que corresponde ao padrão de pesquisa. A `String` que é retornada é aquela que correspondeu da última vez por uma chamada a `find` ou `lookingAt`. A saída na Figura 16.24 mostra as duas correspondências que foram encontradas em `string1`.

Para obter mais informações sobre expressões regulares, visite o nosso Regular Expressions Resource Center em `www.deitel.com/regularexpressions/`.

16.8 Conclusão

Neste capítulo, você aprendeu mais sobre os métodos `String` para selecionar partes de `String`s e manipular `String`s. Você também aprendeu sobre a classe `Character` e alguns métodos que ela declara para tratar `char`s. O capítulo também discutiu as capacidades da classe `StringBuilder` de criar `String`s. O fim do capítulo discutiu as expressões regulares, as quais fornecem uma capacidade poderosa de pesquisar e corresponder partes de `String`s que se ajustam a um padrão particular. No próximo capítulo, você aprenderá sobre o processamento de arquivo, incluindo como os dados persistentes são armazenados e recuperados.

Resumo

Seção 16.2 Fundamentos de caracteres e strings

- O valor de um literal de caractere é seu valor de tipo inteiro no Unicode. As strings podem incluir letras, dígitos e caracteres especiais como +, –, *, / e $. Uma string no Java é um objeto da classe `String`. Os literais de `String` são frequentemente referidos como objetos e escritos entre aspas duplas em um programa.

Seção 16.3 Classe `String`

- Objetos `String` são imutáveis — depois que são criados, seu conteúdo de caracteres não pode ser alterado.
- O método `string` `length` retorna o número de caracteres em uma `String`.
- O método `String` `charAt` retorna o caractere em uma posição específica.
- O método `string` `equals` testa a igualdade. O método retorna `true` se o conteúdo das `String`s forem iguais, `false`, caso contrário. O método `equals` utiliza uma comparação lexicográfica para `String`s.
- Quando valores de tipo de dados primitivo são comparados com ==, o resultado é `true` se ambos os valores forem idênticos. Quando referências são comparadas com ==, o resultado será `true` se ambas referenciam o mesmo objeto.
- O Java trata todos os literais de string com o mesmo conteúdo como um único objeto `String`.

- O método `String` `equalsIgnoreCase` realiza uma comparação de string que não diferencia maiúsculas e minúsculas.
- O método `string` `compareTo` utiliza uma comparação lexicográfica e retorna 0 se as `Strings` forem iguais, um número negativo se a string que chama `compareTo` for menor do que o argumento `String` e um número positivo se a string que chama `compareTo` for maior do que o argumento `String`.
- O método `String` `regionMatches` compara partes de duas strings quanto à igualdade.
- O método `String` `startsWith` determina se uma string inicia com os caracteres especificados. O método `String` `endsWith` determina se uma string termina com os caracteres especificados.
- O método `String` `indexOf` localiza a primeira ocorrência de um caractere ou uma substring em uma string. O método `String` `lastIndexOf` localiza a última ocorrência de um caractere ou uma substring em uma string.
- O método `String` `substring` copia e retorna parte de um objeto string existente.
- O método `String` `concat` concatena dois objetos string e retorna um novo objeto string.
- O método `String` `replace` retorna um novo objeto string que substitui cada ocorrência em uma `String` do seu primeiro argumento de caractere pelo seu segundo argumento de caractere.
- `String` `toUpperCase` retorna uma nova string com letras maiúsculas nas posições em que a string original tinha letras minúsculas. O método `String` `toLowerCase` retorna uma nova string com letras minúsculas nas posições em que a string original tinha letras maiúsculas.
- O método `String` `trim` retorna um novo objeto string em que todos os caracteres de espaço em branco (por exemplo, espaços, nova linha e tabulações) foram removidos do início ao fim de uma string.
- O método `String` `toCharArray` retorna um array `char` contendo uma cópia dos caracteres da string.
- O método `static` `valueOf` da classe `String` retorna seu argumento convertido em uma string.

Seção 16.4 Classe `StringBuilder`

- A classe `StringBuilder` fornece construtores que permitem que `StringBuilders` sejam inicializados sem caracteres e tenham uma capacidade inicial de 16 caracteres, sem caracteres e uma capacidade inicial especificada no argumento de inteiro; ou com uma cópia dos caracteres do argumento `String` e uma capacidade inicial que é o número de caracteres no argumento de `String` mais 16.
- O método `StringBuilder` `length` retorna o número de caracteres atualmente armazenado em um `StringBuilder`. O método `StringBuilder` `capacity` retorna o número de caracteres que pode ser armazenado em um `StringBuilder` sem alocar mais memória.
- O método `StringBuilder` `ensureCapacity` assegura que um `StringBuilder` tem pelo menos a capacidade especificada. O método `StringBuilder` `setLength` aumenta ou diminui o comprimento de um `StringBuilder`.
- O método `StringBuilder` `charAt` retorna o caractere no índice especificado. O método `StringBuilder` `setCharAt` configura o caractere na posição especificada. O método `StringBuilder` `getChars` copia caracteres do `StringBuilder` no array de caractere passado como um argumento.
- Os métodos `append` de `StringBuilder` sobrecarregados adicionam o array de caracteres de tipo primitivo, valores `String`, `Object` ou `CharSequence` ao fim de um `StringBuilder`.
- Os métodos `insert` sobrecarregados de `StringBuilder` inserem tipo primitivo, array de caracteres e valores `String`, `Object` ou `CharSequence` em qualquer posição em um `StringBuilder`.

Seção 16.5 Classe `Character`

- O método `Character` `isDefined` determina se um caractere está no conjunto de caracteres Unicode.
- O método `Character` `isDigit` determina se um caractere é um dígito Unicode definido.
- O método `caractere` `isJavaIdentifierStart` determina se um caractere pode ser utilizado como o primeiro caractere de um identificador Java. O método `Character` `isJavaIdentifierPart` determina se um caractere pode ser utilizado em um identificador.
- O método `caractere` `isLetter` determina se um caractere é uma letra. O método `Character` `isLetterOrDigit` determina se um caractere é uma letra ou um dígito.
- O método `Character` `isLowerCase` determina se um caractere é uma letra minúscula. O método `Character` `isUpperCase` determina se um caractere é uma letra maiúscula.
- O método `Character` `toUpperCase` converte um caractere em seu equivalente em letras maiúsculas. O método `Character` `toLowerCase` converte um caractere em seu equivalente em letras minúsculas.
- O método `Character` `digit` converte seu argumento caractere em um inteiro no sistema de números especificado por seu argumento inteiro `radix`. O método `Character` `forDigit` converte seu argumento inteiro `digit` em um caractere no sistema de números especificado por seu argumento inteiro `radix`.
- O método `caractere` `charValue` retorna o `char` armazenado em um objeto `Character`. O método `Character` `toString` retorna uma representação `String` de um `Character`.

Seção 16.6 Tokenização de `Strings`

- O método `split` da classe `String` tokeniza uma `String` com base no delimitador especificado como um argumento e retorna um array de `Strings` que contém o token.

Seção 16.7 Expressões regulares, classe `Pattern` e classe `Matcher`

- Expressões regulares são sequências de caracteres e símbolos que definem um conjunto de strings. Elas são úteis para validar entrada e assegurar que os dados estão em um formato particular.
- O método `String` `matches` recebe uma string que especifica a expressão regular e corresponde ao conteúdo do objeto `String` em que é chamado com a expressão regular. O método retorna um `boolean` indicando se a correspondência foi ou não bem-sucedida.
- Uma classe de caracteres é uma sequência de escape que representa um grupo de caracteres. Cada classe de caracteres localiza um único caractere na string a que estamos tentando localizar com a expressão regular.
- Um caractere de palavra (`\w`) é qualquer letra (em maiúsculas ou minúsculas), qualquer dígito ou o caractere sublinhado.
- Um caractere espaço em branco (`\s`) é um espaço, uma tabulação, um retorno de carro, uma nova linha ou um avanço de formulário.
- Um dígito (`\d`) é qualquer caractere numérico.
- Para localizar um conjunto de caracteres que não tem uma classe de caracteres predefinida, utilize os colchetes, `[]`. Os intervalos podem ser representados colocando-se um traço (`-`) entre dois caracteres. Se o primeiro caractere entre colchetes for `"^"`, a expressão aceitará qualquer caractere diferente desses indicados.
- Quando o operador de expressão regular `"*"` aparece em uma expressão regular, o programa tenta combinar zero ou mais ocorrências da subexpressão que imediatamente precede o `"*"`.
- O operador `"+"` tenta localizar uma ou mais ocorrências da subexpressão que o precede.
- O caractere `"|"` permite uma correspondência da expressão a sua esquerda ou direita.
- Parênteses ( ) são utilizados para agrupar partes da expressão regular.
- Os sinais de asterisco (`*`) e de adição (`+`) são formalmente chamados de quantificadores.
- Um quantificador afeta apenas a subexpressão que imediatamente o precede.
- O ponto de interrogação quantificador (`?`) localiza zero ou uma ocorrência da expressão que ele quantifica.
- Um conjunto de chaves contendo um número (`{`*n*`}`) localiza exatamente *n* ocorrências da expressão que ele quantifica. Incluir uma vírgula depois do número entre chaves corresponde pelo menos a *n* ocorrências.
- Um conjunto de chaves que contém dois números (`{`*n*`,`*m*`}`), localiza entre *n* e *m* ocorrências da expressão que ele qualifica.
- Os quantificadores são gulosos — eles localizarão quantas ocorrências puderem contanto que a correspondência seja bem-sucedida. Se um quantificador for seguido por um ponto de interrogação (`?`), o quantificador torna-se relutante, identificando o menor número de ocorrências possível contanto que a correspondência seja bem-sucedida.
- O método `String` `replaceAll` substitui o texto em uma string pelo novo texto (o segundo argumento) onde quer que a string original coincida com uma expressão regular (o primeiro argumento).
- Escapar um caractere especial de expressão regular com uma `\` instrui o mecanismo de correspondência de expressão regular a localizar o caractere real, em oposição ao que ele representa em uma expressão regular.
- O método `string` `replaceFirst` substitui a primeira ocorrência de uma correspondência de padrão e retorna uma nova string na qual os caracteres apropriados foram substituídos.
- O método `string` `split` divide uma string em uma substring em qualquer localização que corresponda a uma expressão regular especificada e retorna um array das substrings.
- A classe `Pattern` representa uma expressão regular.
- A classe `Matcher` contém um padrão de expressão regular e um `CharSequence` na qual pesquisar.
- `CharSequence` é uma interface que permite acesso de leitura a uma sequência de caracteres. Tanto a classe `String` como `StringBuilder` implementam essa interface, portanto, podem ser utilizadas com a classe `Matcher`.
- Se uma expressão regular vai ser utilizada apenas uma vez, o método `Pattern` `matches` estático aceita uma string que especifica a expressão regular e uma `CharSequence` na qual realizar a correspondência. Esse método retorna um `boolean` que indica se o objeto de pesquisa corresponde à expressão regular.
- Se uma expressão regular vai ser utilizada mais de uma vez, é mais eficiente utilizar o método `Pattern` `compile` para criar um objeto `Pattern` específico para essa expressão regular. Esse método recebe uma string que representa o padrão e retorna um novo objeto `Pattern`.
- O método `Pattern` `matcher` recebe um `CharSequence` para procurar e retornar um objeto `Matcher`. O método `Matcher` `matches` realiza a mesma tarefa que método `Pattern` `matches`, mas sem argumentos.
- O método `Matcher` `find` tenta localizar uma parte objeto de pesquisa para o padrão de pesquisa. Cada chamada para esse método inicia no ponto em que a última chamada terminou, assim múltiplas correspondências podem ser localizadas.
- O método `Matcher` `lookingAt` executa o mesmo que o `find`, exceto que sempre inicia desde o início do objeto de pesquisa e sempre localizará a primeira correspondência se houver alguma.
- O método `Matcher` `group` retorna a string do objeto de pesquisa que reconhece o padrão de pesquisa. A string retornada é aquela que correspondeu da última vez por uma chamada a `find` ou `lookingAt`.

Terminologia

append, método da classe StringBuilder, 529
capacity, método da classe StringBuilder, 527
caractere especial, 516
caractere palavra (expressões regulares), 536
charAt, método da classe String, 518
charAt, método da classe StringBuilder, 528
CharSequence, interface, 541
charValue, método da classe Character, 534
classe de caractere predefinida (expressões regulares), 536
comparação lexicográfica, 520
compile, método da classe Pattern, 541
concat, método da classe String, 524
delete, método da classe StringBuilder, 530
deleteCharAt, método da classe StringBuilder, 530
delimitador para tokens, 535
digit, método da classe Character, 533
endsWith, método da classe String, 521
ensureCapacity, método da classe StringBuilder, 527
expressão regular, 536
find, método da classe Matcher, 541
forDigit, método da classe Character, 533
getChars, método da classe String, 518
getChars, método da classe StringBuilder, 528
group, método da classe Matcher, 542
indexOf, método da classe String, 522
insert, método da classe StringBuilder, 530
isDefined, método da classe Character, 533
isDigit, método da classe Character, 533
isJavaIdentifierPart, método da classe Character, 533
isJavaIdentifierStart, método da classe Character, 533
isLetter, método da classe Character, 533
isLetterOrDigit, método da classe Character, 533
isLowerCase, método da classe Character, 533
isUpperCase, método da classe Character, 533
lastIndexOf, método da classe String, 522
length, método da classe String, 518
length, método da classe StringBuilder, 527
literal de caractere, 516
literal de string, 516
lookingAt, método da classe Matcher, 541
Matcher, classe, 541
matcher, método da classe Pattern, 541
matches, método da classe Matcher, 541
matches, método da classe Pattern, 541
matches, método da classe String, 536
Pattern, classe, 541
quantificadores utilizados em expressões regulares, 539
quantificador ganancioso (expressões regulares, 539
quantificador preguiçoso (expressões regulares), 539
quantificador relutante (expressões regulares), 539
radical (base) de um número, 533, 534
regionMatches, método da classe String, 519
replaceAll, método da classe Matcher, 541
replaceAll, método da classe String, 540
replaceFirst, método da classe Matcher, 541
replaceFirst, método da classe String, 540
reverse, método da classe StringBuilder, 528
setCharAt, método da classe StringBuilder, 528
split, método da classe String, 535
startsWith, método da classe String, 521
String, objeto, imutável, 517
StringBuffer, classe, 526
StringBuilder, classe, 526
StringIndexOutOfBoundsException, classe, 523
string vazia, 517
toCharArray, método da classe String, 525
token de uma String, 535
toLowerCase, método da classe Character, 533
toLowerCase, método da classe String, 525
toUpperCase, método da classe Character, 533
toUpperCase, método da classe String, 525
trim, método da classe String, 525
Unicode, conjunto de caracteres, 516
valueOf, método da classe String, 525

Exercícios de autorrevisão

16.1 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Quando os objetos String são comparados utilizando ==, o resultado é true se as Strings contiverem os mesmos valores.

b) Uma String pode ser modificada depois de criada.

16.2 Para cada um dos seguintes, escreva uma única instrução que realiza a tarefa indicada:

a) Compare a string em s1 com string em s2 quanto à igualdade de conteúdo.

b) Acrescente a string s2 à string s1, utilizando +=.

c) Determine o comprimento da string em s1.

Respostas dos exercícios de autorrevisão

16.1 a) Falsa. Os objetos string são comparados utilizando o operador == para determinar se eles são o mesmo objeto na memória.

b) Falsa. Objetos String são imutáveis e não podem ser modificados depois de criados. Objetos StringBuilder podem ser modificados depois de criados.

16.2 a)
```
s1.equals( s2 )
```

b)
```
s1 += s2;
```

c)
```
s1.length()
```

Exercícios

16.3 (***Comparando Strings***) Escreva um aplicativo que utiliza o método String compareTo para comparar duas entradas de strings pelo usuário. Crie uma saída informando se a primeira string é menor que, igual a ou maior que a segunda.

16.4 (***Comparando partes de Strings***) Escreva um aplicativo que utiliza o método `String` `regionMatches` para comparar duas entradas de strings pelo usuário. O aplicativo deve inserir o número de caracteres que será comparado e o índice inicial da comparação. O aplicativo deve declarar se as strings são iguais. Ignore a distinção entre maiúsculas e minúsculas dos caracteres ao realizar a comparação.

16.5 (***Sentenças aleatórias***) Escreva um aplicativo que utiliza geração de números aleatórios para criar frases. Utilize quatro arrays de strings chamados `article`, `noun`, `verb` e `preposition`. Crie uma frase selecionando uma palavra aleatoriamente de cada array na seguinte ordem: `article`, `noun`, `verb`, `preposition`, `article` e `noun`. À medida que cada palavra for selecionada, concatene-a as primeiras palavras na frase. As palavras devem ser separadas por espaços. Quando a frase final for enviada para saída, ela deve iniciar com uma letra maiúscula e terminar com um ponto. O aplicativo deve gerar e exibir 20 frases.

O array de artigos deve conter os artigos `"the"`, `"a"`, `"one"`, `"some"` e `"any"`; o array de substantivos deve conter os substantivos `"boy"`, `"girl"`, `"dog"`, `"town"` e `"car"`; o array de verbos deve conter os verbos `"drove"`, `"jumped"`, `"ran"`, `"walked"` e `"skipped"`; o array de preposições deve conter as preposições `"to"`, `"from"`, `"over"`, `"under"` e `"on"`.

16.6 (***Limericks***) Um limerick é um poema humorístico de cinco versos em que a primeira e a segunda linha rimam com a quinta, e a terceira linha rima com a quarta. Utilizando técnicas semelhantes àquelas desenvolvidas no Exercício 16.5, escreva um aplicativo Java que produz limericks aleatórios. Polir esse aplicativo para produzir bons limericks é um problema desafiador, mas o resultado vale o esforço!

16.7 (***Latim de porco***) Escreva um aplicativo que codifica frases da língua inglesa em latim de porco. O Pig Latin é uma forma de linguagem codificada. Há muitos métodos diferentes para formar frases em Pig Latin. Para simplificar, utilize o seguinte algoritmo:

Para formar uma frase em latim de porco a partir de uma frase em inglês, tokenize a frase em palavras com o método `String split`. Para traduzir cada palavra inglesa em uma palavra do latim de porco, coloque a primeira letra da palavra inglesa no fim da palavra e adicione as letras "ay". Assim, a palavra "jump" torna-se "umpjay," a palavra "the" torna-se "hetay," e a palavra "computer" torna-se "omputercay." Os espaços entre as palavras permanecem iguais. Assuma o seguinte: a frase inglesa consiste em palavras separadas por espaços, não há nenhuma marcação de pontuação e todas as palavras têm duas ou mais letras. O método `printLatinWord` deve exibir cada palavra. Cada token é passado para o método `printLatinWord` para imprimir a palavra em latim de porco (*Pig Latin*). Permita que o usuário insira a frase. Continue exibindo todas as frases convertidas em uma área de texto.

16.8 (***Tokenizando números de telefone***) Escreva um aplicativo que insere um número de telefone como uma string na forma `(555) 555-5555`. O aplicativo deve utilizar o método `String split` para extrair o código de área como um token, os três primeiros dígitos do número de telefone como um segundo token e os últimos quatro dígitos do número de telefone como um terceiro token. Os sete dígitos do número de telefone devem ser concatenados em uma string. O código de área e o número de telefone devem ser impressos. Lembre-se de que você terá de alterar caracteres delimitadores durante o processo de tokenização.

16.9 (***Exibindo uma oração com suas palavras invertidas***) Escreva um aplicativo que introduz uma linha do texto, tokeniza a linha com o método `String split` e gera saída do token em ordem inversa. Utilize caracteres espaço em branco como delimitadores.

16.10 (***Exibindo Strings em letras maiúsculas e minúsculas***) Escreva um aplicativo que insere uma linha de texto e gera duas vezes a saída do texto — uma vez em letras maiúsculas e uma vez em letras minúsculas.

16.11 (***Pesquisando Strings***) Escreva um aplicativo que insere uma linha de texto e um caractere de pesquisa e utiliza o método `String indexOf` para determinar o número de ocorrências do caractere no texto.

16.12 (***Pesquisando Strings***) Escreva um aplicativo com base no aplicativo do Exercício 16.11 que insere uma linha de texto e utiliza o método `String indexOf` para determinar o número total de ocorrências de cada letra do alfabeto no texto. As letras minúsculas e maiúsculas devem ser contadas juntas. Armazene os totais para cada letra em um array e imprima os valores em formato tabular depois que os totais foram determinados.

16.13 (***Tokenizando e comparando Strings***) Escreva um aplicativo que lê uma linha de texto, tokeniza essa linha utilizando caracteres de espaço em branco como delimitadores e gera a saída apenas daquelas palavras que iniciam com a letra "`b`".

16.14 (***Tokenizando e comparando Strings***) Escreva um aplicativo que lê uma linha de texto, tokeniza essa linha utilizando caracteres de espaço em branco como delimitadores e gera a saída apenas daquelas palavras que terminem com as letras "`ED`".

16.15 (***Convertendo valores int em caracteres***) Escreva um aplicativo que introduz um código inteiro de um caractere e exibe o caractere correspondente. Modifique esse aplicativo de modo que ele gere todos os possíveis códigos de três dígitos no intervalo de 000 a 255 e tente imprimir os caracteres correspondentes.

16.16 (***Definindo seus próprios métodos String***) Escreva suas próprias versões de métodos de pesquisa `String indexOf` e `lastIndexOf`.

16.17 (***Criando Strings de três letras de uma palavra de cinco letras***) Escreva um aplicativo que lê uma palavra de cinco letras do usuário e produz cada possível string de três letras que pode ser derivada das letras dessa palavra. Por exemplo, as palavras de três letras produzidas a partir da palavra "bathe" incluem "ate," "bat," "bet," "tab," "hat," "the" e "tea".

Seção especial: exercícios de manipulação avançada de string

Os exercícios precedentes são voltados para o texto e para projetados para testar seu entendimento de conceitos fundamentais de manipulação de string. Esta seção inclui uma coleção de exercícios de manipulação de string avançados e intermediários. Você deve achar esses problemas desafiadores, mas divertidos. Os problemas variam consideravelmente em dificuldade. Alguns requerem uma hora ou duas para escrever e implementar o aplicativo. Outros são úteis para atribuições de laboratório que talvez requeiram duas ou três semanas de estudo e implementação. Alguns são projetos de conclusão de curso desafiadores.

16.18 (***Análise de texto***) A disponibilidade de computadores com capacidades de manipulação de string resultou em algumas abordagens bastante interessantes para analisar textos de grandes autores. Muita atenção foi dada à polêmica de que William Shakespeare não teria existido de fato. Alguns acadêmicos acreditam haver evidências substanciais que indicam que Christopher Marlowe realmente escreveu as obras-primas atribuídas a Shakespeare. Os pesquisadores têm utilizado computadores para encontrar semelhanças na escrita desses dois autores. Esse exercício examina três métodos para analisar textos com um computador.

a) Escreva um aplicativo que lê uma linha de texto do teclado e imprime uma tabela que indica o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

```
To be, or not to be: that is the question:
```

contém um "a," dois "b", nenhum "c," e assim por diante.

b) Escreva um aplicativo que lê uma linha de texto e imprime uma tabela que indique o número de palavras de uma letra, palavras de duas letras, palavras de três letras e assim por diante, aparecer no texto. Por exemplo, a Figura 16.25 mostra as contagens para a frase

```
Whether 'tis nobler in the mind to suffer
```

Comprimento de palavra	Ocorrências
1	0
2	2
3	1
4	2 (incluindo `'tis`)
5	0
6	2
7	1

Figura 16.25 | Contagens de comprimento de palavra da string `"Whether 'tis nobler in the mind to suffer"`.

c) Escreva um aplicativo que lê uma linha de texto e imprime uma tabela que indica o número de ocorrências de cada palavra diferente no texto. O aplicativo deve incluir as palavras na tabela na mesma ordem em que elas aparecem no texto. Por exemplo, as linhas

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contém a palavra "to" três vezes, a palavra "be" duas vezes, a palavra "or" uma vez etc.

16.19 (***Imprimindo datas em vários formatos***) As datas são impressas em vários formatos comuns. Dois dos formatos mais comuns em inglês são

```
04/25/1955 and April 25, 1955
```

Escreva um aplicativo que lê uma data no primeiro formato e imprime no segundo formato.

16.20 (***Proteção de cheque***) Os computadores são frequentemente empregados em sistemas de verificação de escrita como aplicativos de folha de pagamento e contas a pagar. Ouvimos muitas histórias estranhas relacionadas a cheques de pagamento semanal que são impressos (por engano) com quantias de mais de US$ 1 milhão. Quantidades incorretas são impressas por sistemas computadorizados de preenchimento de cheque por causa de erro humano ou falha de máquina. Os projetistas de sistemas embutem controles em seus sistemas para evitar a emissão desses cheques errados.

Outro problema sério é a alteração intencional do valor de um cheque por alguém que planeja receber um cheque fraudulentamente. Para evitar que uma quantia monetária seja alterada, alguns sistemas computadorizados de preenchimento de cheque empregam uma técnica chamada proteção de cheque. Cheques projetados para imprimir por computador contém um número fixo de espaços em que o computador pode imprimir uma quantia. Suponha que um cheque de pagamento contenha oito espaços em branco em que o computador deve imprimir a quantidade de um cheque de pagamento semanal. Se o valor for alto, então todos os oito espaços serão preenchidos. Por exemplo,

```
1,230.60 (check amount)
--------
12345678 (position numbers)
```

Por outro lado, se a quantidade for menor que US$ 1000, então vários dos espaços seriam comumente deixados em branco. Por exemplo,

```
   99.87
--------
12345678
```

contém três espaços em branco. Se um cheque for impresso com espaços em branco, é mais fácil para alguém alterar a quantia. Para impedir a alteração, muitos sistemas de preenchimento de cheque inserem *asteriscos iniciais* para proteger a quantia como mostrado a seguir:

```
***99.87
--------
12345678
```

Escreva um aplicativo que insere uma quantia monetária que será impressa em um cheque e então imprime o valor em formato de cheque protegido com asteriscos iniciais se necessário. Assuma que nove espaços estão disponíveis para imprimir o valor.

16.21 (***Escrevendo o valor de um cheque por extenso***) Continuando a discussão do Exercício 16.20, reiteramos a importância de se projetar sistemas de preenchimento de cheques para impedir a alteração de valores do cheque. Um método comum de segurança requer que o valor seja escrito em números e "por extenso" também. Mesmo se alguém for capaz de alterar o valor numérico do cheque, é extremamente difícil alterar o valor por extenso. Escreva um aplicativo que insere um valor numérico de cheque menor do que $1000 e escreve o valor por extenso em inglês. Por exemplo, o valor 112,43 deve ser escrito assim

```
ONE hundred TWELVE and 43/100
```

16.22 (***Código Morse***) Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para utilização com o sistema de telégrafo. O código Morse atribui uma série de pontos e traços para cada letra do alfabeto, para cada dígito e alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas orientados para áudio, o ponto representa um som curto e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em sinais de bandeira. A separação entre palavras é indicada por um espaço, ou, simplesmente, a ausência de um ponto ou traço. Em um sistema orientado a som, um espaço é indicado por um tempo curto durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Figura 16.26

Escreva um aplicativo que lê uma frase em inglês e a codifica em código Morse. Escreva também um aplicativo que lê uma frase em código Morse e a converte no equivalente em inglês. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse.

Caractere	Código	Caractere	Código
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Dígitos*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Figura 16.26 | Letras e dígitos como expressos em código Morse internacional.

16.23 (***Conversões métricas***) Escreva um aplicativo que auxiliará o usuário com conversões métricas. Seu aplicativo deve permitir que o usuário especifique os nomes das unidades como strings (isto é, centímetros, litros, gramas etc. para o sistema métrico e polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Seu aplicativo deve reconhecer conversões inválidas. Por exemplo, a pergunta

```
"How many feet are in 5 kilograms?"
```

não é significativo porque `"feet"` é uma unidade de comprimento, enquanto `"kilograms"` é uma unidade de massa.

Seção especial: projetos de manipulação de string desafiadores

16.24 (***Projeto: Um corretor ortográfico***) Muitos pacotes populares de software processador de texto têm verificadores ortográficos integrados. Nesse projeto, exige-se que você desenvolva seu próprio utilitário de verificação ortográfica. Fazemos sugestões para ajudá-lo a começar. Você então deve considerar a adição de mais capacidades. Utilize um dicionário computadorizado (se tiver acesso a um) como uma fonte de palavras.

Por que digitamos tantas palavras com ortografia incorreta? Em alguns casos, isso é porque simplesmente não conhecemos a ortografia correta, então fazemos nossa "melhor suposição". Em alguns casos, é porque transpomos duas letras (por exemplo, "pardão" em vez de "padrão)". Ocasionalmente digitamos duas vezes uma letra acidentalmente (por exemplo, "canssado" em vez de "cansado)". Às vezes digitamos uma tecla próxima em vez daquela pretendida (por exemplo, "amiverário" em vez de "aniversário") e assim por diante.

Projete e implemente um aplicativo de verificador ortográfico em Java. Seu aplicativo deve manter um array `wordList` de strings. Permita que o usuário insira essas strings. [*Nota:* no Capítulo 17, introduzimos processamento de arquivo. Com essa capacidade, você pode obter as palavras para o verificador ortográfico de um dicionário computadorizado armazenado em um arquivo].

Seu aplicativo deve solicitar que um usuário insira uma palavra. O aplicativo deve então pesquisar essa palavra no array `wordList`. Se a palavra estiver no array, seu aplicativo deve imprimir `"Word is spelled correctly."` Se a palavra não estiver no array, seu aplicativo deve imprimir `"Word is not spelled correctly."` Então o aplicativo deve tentar localizar outras palavras em `wordList` que podem ser a palavra que o usuário pretendia digitar. Por exemplo, você pode tentar todas as transposições possíveis simples de letras adjacentes para descobrir que a palavra "default" é uma correspondência direta com uma palavra na `wordList`. Naturalmente, isso implica que seu aplicativo verificará todas as outras transposições simples, como "edfault", "dfeault", "deafult", "defalut", e "defautl". Quando você encontrar uma nova palavra que localiza uma palavra na `wordList`, imprima essa palavra em uma mensagem, como

```
Did you mean "default"?
```

Implemente outros testes, como substituir cada letra dupla por uma única letra e algum outro teste que você pode desenvolver para aprimorar o valor de seu verificador ortográfico.

16.25 (***Projeto: Um gerador de palavras cruzadas***) A maioria das pessoas já brincou de palavras cruzadas, mas poucos tentaram gerar um jogo de palavras cruzadas. Gerar um jogo de palavras cruzadas é sugerido aqui como um projeto de manipulação de string que requer bastante sofisticação e esforço.

Há muitas questões que o programador deve resolver para que até mesmo o mais simples aplicativo gerador de palavras cruzadas funcione. Por exemplo, como a grade das palavras cruzadas é representada dentro do computador? Você deve utilizar uma série de strings ou arrays bidimensionais?

O programador precisa de uma fonte de palavras (isto é, um dicionário computadorizado) que possa ser referenciado diretamente pelo aplicativo. De que forma essas palavras devem ser armazenadas para facilitar as complexas manipulações requeridas pelo aplicativo?

Se você for realmente ambicioso, vai querer gerar a parte de pistas do quebra-cabeça, em que breves dicas para palavras na horizontal e na vertical são impressas. Meramente imprimir uma versão da parte em branco do jogo não é um problema simples.

Fazendo a diferença

16.26 (***Cozinhando com ingredientes mais saudáveis***) A obesidade nos Estados Unidos está aumentando a uma taxa alarmante. Consulte o mapa dos Centers for Disease Control and Prevention (CDC) em `www.cdc.gov/nccdphp/dnpa/Obesity/trend/maps/index.htm`, que mostra tendências de obesidade nos Estados Unidos nos últimos 20 anos. À medida que a obesidade aumenta, crescem também as ocorrências de problemas relacionados (por exemplo, doença de coração, hipertensão, colesterol alto, diabetes tipo 2). Escreva um programa que ajude usuários a escolher ingredientes mais saudáveis ao cozinhar e ajude usuários alérgicos a encontrar substitutos para certos alimentos (por exemplo, castanhas, glúten). O programa deve ler uma receita de uma `JTextArea` e sugerir substituições mais saudáveis para alguns ingredientes. Por uma questão de simplicidade, ele deve supor que a receita não tem abreviações para medidas como colheres de chá, xícaras e colheres de sopa, e utilizar dígitos numéricos para as quantidades (por exemplo, 1 ovo, 2 xícaras) em vez de escrevê-los (um ovo, duas xícaras). Algumas substituições comuns são mostradas na Figura 16.27. O programa deve exibir um aviso como: "Sempre consulte o médico antes de fazer mudanças significativas na sua dieta".

Ingrediente	Substituto
1 xícara de creme de leite	1 xícara de iogurte
1 xícara de leite	½ xícara de leite evaporado e ½ xícara de água
1 colher de chá de suco de limão	½ de chá de vinagre
1 xícara de açúcar	½ xícara de mel, 1 xícara de melado ou ¼ de xícara de néctar de agave
1 xícara de manteiga	1 xícara de margarina ou iogurte
1 xícara de farinha	1 xícara de farinha de centeio ou arroz
1 xícara de maionese	1 xícara de queijo minas ou 1/8 de xícara de iogurte e 7/8 de xícara de iogurte

Ingrediente	Substituto
1 ovo	2 de colheres de sopa de amido de milho, farinha de araruta ou fécula de batata ou 2 claras de ovo ou 1/2 banana grande (amassada)
1 xícara de leite	1 xícara de leite de soja
¼ de xícara de óleo	¼ de xícara de suco de maçã
pão branco	pão integral

Figura 16.27 | Substitutos de ingredientes típicos.

O programa deve levar em conta que as substituições nem sempre são um para um. Por exemplo, se uma receita de bolo tem três ovos, em vez disso, ela poderia utilizar razoavelmente seis claras. Dados de conversão para medidas e substitutos podem ser obtidos em sites como:

```
chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm
```

O programa deve considerar as preocupações de saúde do usuário, como colesterol alto, hipertensão, perda de peso, alergia a glúten e assim por diante. Para o colesterol alto, o programa deve sugerir substitutos para ovos e laticínios; se o usuário quiser perder peso, substitutos de baixa caloria para ingredientes como açúcar devem ser sugeridos.

16.27 **(*Scanner de Spam*)** O spam (ou *junk email*) custa bilhões de dólares às organizações dos Estados Unidos por ano em software de prevenção do spam, equipamento, recursos de rede, largura de banda e produtividade perdida. Pesquise on-line algumas das mensagens e palavras de e-mail de spam mais comuns, e verifique sua própria pasta de lixo eletrônico. Crie uma lista das 30 palavras e frases mais encontradas nas mensagens de spam. Escreva um aplicativo no qual o usuário insere uma mensagem de e-mail em uma `JTextArea`. Em seguida, varra a mensagem de cada uma das 30 palavras-chave ou frases. Para cada ocorrência de uma dessas dentro da mensagem, adicione um ponto ao "escore de spam" da mensagem. Depois, avalie a probabilidade de a mensagem ser um spam, com base no número de pontos que ela recebeu.

16.28 **(*Linguagem SMS*)** O SMS (Short Message Service) é um serviço de comunicação que permite enviar mensagens de texto de 160 ou menos caracteres entre telefones celulares. Com a proliferação do uso do celular no mundo inteiro, o SMS está sendo utilizado em muitos países em desenvolvimento com objetivos políticos (por exemplo, expressar opiniões e fazer oposição), reportagens sobre catástrofes naturais, e assim por diante. Por exemplo, consulte `comunica.org/radio2.0/archives/87`. Como o tamanho das mensagens de SMS é limitado, linguagem SMS — as abreviações de palavras e frases comuns em mensagens mobile text, e-mails, mensagens instantâneas etc. — são muito usadas. Por exemplo, "in my opinion" é "imo" na linguagem SMS. Pesquise on-line o termo SMS Language. Escreva um aplicativo GUI em que o usuário pode inserir uma mensagem utilizando linguagem SMS, depois clicar em um botão para traduzi-lo para o inglês (ou para sua própria língua). Também forneça um mecanismo para traduzir o texto escrito para o inglês (ou para sua própria língua) na linguagem SMS. Um problema potencial é que uma abreviação de SMS poderia ter vários significados. Por exemplo, IMO (como utilizado acima) também pode significar "International Maritime Organization" "in memory of" etc.

Só posso assumir que um documento marcado como "Não Arquivar" está arquivado como "Não Arquivar".
— Senador Frank Church, Depoimento ao Subcomitê de Inteligência do Senado, 1975

A consciência … em si não aparece dividida em pedaços [bits]. Um "rio" ou um "fluxo" são as metáforas com que ela é mais naturalmente descrita.
— William James

Li parte dele até o fim.
— Samuel Goldwyn

Uma memória excelente não faz um filósofo, assim como um dicionário não pode ser chamado de gramática.
— John Henry, Cardeal Newman

17

Arquivos, fluxos e serialização de objetos

Objetivos

Neste capítulo, você aprenderá:

- O que são arquivos e como eles são usados para reter dados de aplicativo entre sucessivas execuções.
- A criar, ler, gravar e atualizar arquivos.
- A utilizar a classe `File` para recuperar informações sobre arquivos e diretórios.
- A hierarquia de classes para fluxo de entrada/saída do Java.
- As diferenças entre arquivos de texto e arquivos binários.
- O processamento de arquivo de acesso sequencial.
- A utilizar as classes `Scanner` e `Formatter` para processar arquivos de texto.
- A utilizar as classes `FileInputStream` e `FileOutputStream` para ler e gravar arquivos.
- A utilizar as classes `ObjectInputStream` e `ObjectOutputStream` para ler e gravar objetos em arquivos.
- A utilizar um diálogo de `JFileChooser`.

Sumário

17.1 Introdução
17.2 Hierarquia de dados
17.3 Arquivos e fluxos
17.4 Classe `File`
17.5 Arquivos de texto de acesso sequencial
17.5.1 Criando um arquivo de texto de acesso sequencial
17.5.2 Lendo dados a partir de um arquivo de texto de acesso sequencial
17.5.3 Estudo de caso: um programa de consulta de crédito
17.5.4 Atualizando arquivos de acesso sequencial
17.6 Serialização de objeto
17.6.1 Criando um arquivo de acesso sequencial com a serialização de objeto
17.6.2 Lendo e desserializando dados a partir de um arquivo de acesso sequencial
17.7 Classes `java.io` adicionais
17.7.1 Interfaces e classes para entrada e saída baseadas em bytes
17.7.2 Interfaces e classes para entrada e saída baseadas em caracteres
17.8 Abrindo arquivos com `JFileChooser`
17.9 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

17.1 Introdução

Dados armazenados em variáveis e arrays são temporários — eles são perdidos quando uma variável local "sai do escopo" ou quando o programa termina. Para retenção de longo prazo dos dados, mesmo depois de os programas que criam os dados terminarem, os computadores utilizam **arquivos**. Você utiliza arquivos diariamente para tarefas como escrever um e-mail ou criar uma planilha. Os computadores armazenam arquivos em **dispositivos de armazenamento secundários**, como discos rígidos, discos ópticos e fitas magnéticas. Os dados mantidos nos arquivos são **dados persistentes** porque existem além da duração da execução do programa. Neste capítulo, explicaremos como programas Java criam, atualizam e processam arquivos.

Uma linguagem de programação deve ser capaz de processar arquivos para suportar aplicativos comerciais, que tipicamente processam quantidades maciças de dados persistentes. Este capítulo discute o processamento de arquivos do Java e os recursos de entrada/saída de fluxo. O termo "fluxo" refere-se a dados ordenados que são lidos ou gravados em um arquivo. Discutiremos os fluxos em detalhes na Seção 17.3. O processamento de arquivos é um subconjunto das capacidades de processamento de fluxos do Java, que incluem ler e gravar dados de memória, arquivos e conexões de rede. Neste capítulo, introduziremos os conceitos do processamento de arquivos (fazendo com que você se sinta mais à vontade com o uso de arquivos programaticamente) e forneceremos capacidades de processamento de fluxo suficientes para suportar os recursos em rede introduzidos no Capítulo 27, "Redes". Discutimos duas formas do processamento de arquivos aqui — processamento de arquivo de texto e serialização de objetos.

Iniciamos discutindo a hierarquia de dados contida em arquivos. Então abrangeremos a arquitetura do Java para tratar arquivos programaticamente — discutimos as várias classes do pacote `java.io`. Em seguida, explicamos que os dados podem ser armazenados em arquivos de texto e arquivos binários — e cobrimos as diferenças entre eles. Demonstramos como recuperar informações sobre arquivos ou diretórios usando a classe `File` e, então, dedicamos várias seções aos diferentes mecanismos para gravar e ler dados de arquivos. Ensinamos como criar e manipular arquivos de texto de acesso sequencial. Trabalhar com arquivos de texto permite que o leitor comece a manipular arquivos rápida e facilmente. Como você aprenderá, porém, é difícil ler dados a partir de arquivos de texto e de volta na forma de objetos. Felizmente, várias linguagens orientadas a objetos (incluindo Java) fornecem maneiras de gravar e ler objetos em arquivos (conhecido como serialização e desserialização de objetos). Para demonstrar isso, recriamos alguns de nossos programas de acesso sequencial que utilizaram arquivos de texto, dessa vez armazenando os objetos em arquivos binários.

17.2 Hierarquia de dados

Em última instância, um computador processa os itens de dados como combinações de zeros e uns, porque é simples e econômico para os engenheiros construírem dispositivos eletrônicos que podem assumir dois estados estáveis — um representando `0` e o outro representando `1`. É notável que as impressionantes funções realizadas pelos computadores envolvam somente as manipulações mais fundamentais de `0`s e `1`s.

Bits

O menor item de dados em um computador pode assumir o valor `0` ou o valor `1`. Esse item de dados é chamado de **bit** (abreviação de "binary digit" — um dígito que pode assumir um de dois valores). O circuito elétrico dos computadores executa várias manipulações de bits simples, como examinar o valor de um bit, configurar o valor de um bit e inverter o valor de um bit (de `1` a `0` ou de `0` a `1`).

Caracteres

É incômodo para os programadores trabalharem com dados na forma de baixo nível de bits. Em vez disso, eles preferem trabalhar com **dígitos decimais** (0–9), **letras** (A–Z e a–z) e **símbolos especiais** (por exemplo, $, @, %, &, *, (,), –, +, ", :, ? e /). Dígitos, letras e símbolos especiais são conhecidos como **caracteres**. O **conjunto de caracteres** do computador é o conjunto de todos os caracteres utilizados para escrever programas e representar itens de dados. Os computadores processam somente 1s e 0s, assim o conjunto de caracteres de um computador representa cada caractere como um padrão de 1s e 0s. O Java utiliza caracteres **Unicode** compostos de dois **bytes**, cada um composto de oito bits. O tipo `byte` do Java pode ser utilizado para representar dados de byte. O Unicode contém caracteres para muitos idiomas do mundo. Consulte o Apêndice L para informações adicionais sobre esse conjunto de caracteres. Consulte o Apêndice B, para informações adicionais sobre o conjunto de caracteres **ASCII (American Standard Code for Information Interchange)**, um subconjunto popular do Unicode que representa letras maiúsculas e minúsculas, dígitos e vários caracteres especiais comuns.

Campos

Assim como caracteres são compostos de bits, **campos** são compostos de caracteres ou bytes. Um campo é um grupo de caracteres ou bytes que transmitem um significado. Por exemplo, um campo consistindo em letras minúsculas em letras maiúsculas ser utilizado para representar um nome da pessoa.

Itens de dados processados pelos computadores formam uma **hierarquia de dados** que se torna, com relação à estrutura, maior e mais complexa à medida que progredimos de bits para caracteres, campos e assim por diante.

Registros e arquivos

Em geral, vários campos compõem um **registro** (implementado como uma `classe` em Java). Por exemplo, em um sistema de folha de pagamento o registro para um funcionário poderia consistir nos seguintes campos (possíveis tipos para esses campos são mostrados entre parênteses):

- número de identificação do funcionário (`int`)
- nome (`String`)
- endereço (`String`)
- salário-hora (`double`)
- número de isenções reivindicadas (`int`)
- rendimentos no ano até a presente data (`int` ou `double`)
- total de impostos retidos (`int` ou `double`)

Portanto, um registro é um grupo de campos relacionados. No exemplo anterior, todos os campos pertencem ao mesmo funcionário. Uma empresa poderia ter vários funcionários e assim ter um registro de folha de pagamentos para cada um. Um **arquivo** é um grupo de registros relacionados. [*Nota*: De maneira mais geral, um arquivo contém dados arbitrários em formatos arbitrários. Em alguns sistemas operacionais, um arquivo é visualizado como nada além de uma coleção de bytes — qualquer organização dos bytes em um arquivo (por exemplo, organizar os dados em registros) é uma visualização criada pelo programador do aplicativo]. O arquivo de folha de pagamento de uma empresa normalmente contém um registro por empregado. Portanto, o arquivo de folha de pagamento de uma pequena empresa poderia conter apenas 22 registros, enquanto o de uma grande empresa poderia conter milhares. Não é incomum uma empresa ter muitos arquivos, que contêm alguns bilhões, ou mesmo trilhões, de caracteres de informações. A Figura 17.1 ilustra uma parte da hierarquia de dados.

Chaves de registro

Para facilitar a recuperação de registros específicos de um arquivo, pelo menos um campo em cada registro é escolhido como uma **chave de registro**. Uma chave de registro identifica um registro como pertencente a uma pessoa ou empresa em particular e é única para cada registro. Esse campo em geral é utilizado para pesquisar e classificar registros. No registro de folha de pagamento descrito previamente, o número de identificação de empregado normalmente seria escolhido como a chave de registro.

Arquivos sequenciais

Há muitas maneiras de organizar registros em um arquivo. O mais comum é chamado **arquivo sequencial**, em que registros são armazenados na ordem pelo campo-chave de registro. Por exemplo, um arquivo de folha de pagamentos comumente insere os registros em ordem crescente pelo número de identificação dos funcionários.

Bancos de dados

A maioria das organizações armazena dados em vários arquivos diferentes. Empresas podem ter arquivos de folha de pagamento, arquivos de contas a receber (listagem de dinheiro devido por clientes), arquivo de contas a pagar (listagem de dinheiro devido a fornecedores), arquivo de inventário (listagem de fatos sobre todos os itens abrangidos pelo negócio) e muitos outros. Um grupo de arquivos relacionados costuma ser chamado de **banco de dados**. Uma coleção de programas projetados para criar e gerenciar bancos de dados é chamada sistema de gerenciamento de bancos de dados **(DataBase Management System — DBMS)**. Discutiremos esse tópico no Capítulo 28, "Acesso a banco de dados com JDBC".

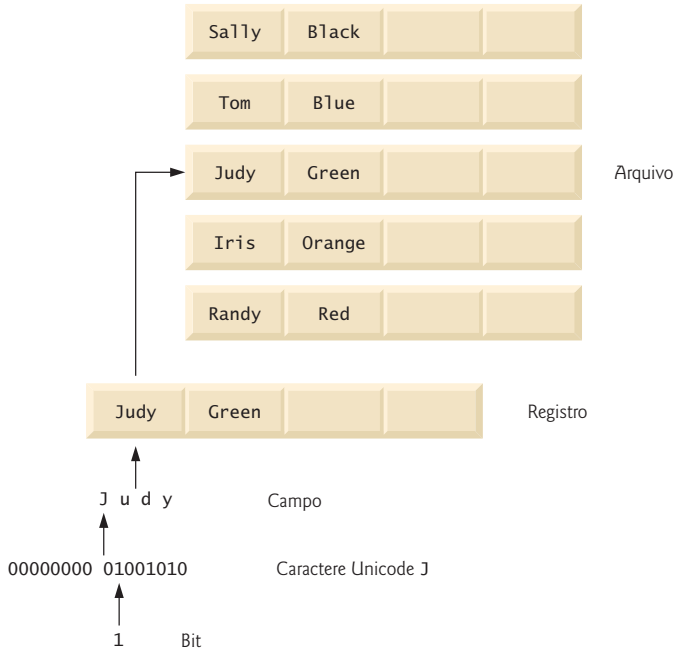


Figura 17.1 | Hierarquia de dados.

17.3 Arquivos e fluxos

O Java vê cada arquivo como um **fluxo** sequencial de bytes (Figura 17.2). Cada sistema operacional fornece um mecanismo para determinar o término de um arquivo, como um **marcador de fim de arquivo** ou uma contagem do total de bytes no arquivo que é registrado nos dados mantidos na estrutura do sistema administrativo. Um programa Java que processa um fluxo de bytes simplesmente recebe uma indicação do sistema operacional quando ele alcança o fim do fluxo — o programa não precisa saber como a plataforma subjacente representa arquivos ou fluxos. Em alguns casos, a indicação de fim de arquivo ocorre como uma exceção. Em outros casos, a indicação é um valor de retorno de um método invocado sobre um objeto que processa o fluxo.

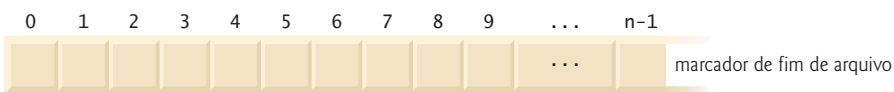


Figura 17.2 | Visualização do Java de um arquivo de *n* bytes.

Fluxos baseados em bytes e fluxos baseados em caracteres

Fluxos de arquivos podem ser utilizados para entrada e saída de dados como bytes ou caracteres. Os fluxos de entrada e saída de bytes são conhecidos como **fluxos baseados em bytes**, representando os dados no seu formato binário. Os fluxos de entrada e saída de caracteres para arquivos são conhecidos como **fluxos baseados em caracteres,** representando os dados como uma sequência de caracteres. Por exemplo, se o valor `5` fosse armazenado utilizando um fluxo baseado bytes, ele seria armazenado no formato binário do valor numérico `5`, ou `101`. Se o valor `5` fosse armazenado usando um fluxo baseado em caracteres, ele seria armazenado no formato binário do caractere `5`, ou `00000000 00110101` (essa é a representação binária para o valor numérico `53`, que indica o caractere `5` no conjunto de caracteres Unicode). A diferença entre as duas formas é que o valor numérico pode ser utilizado como um número inteiro nos cálculos, enquanto o caractere `5` é simplesmente um caractere que pode ser utilizado em uma string do texto, como em `"Sarah Miller is 15 years old"`. Arquivos criados usando fluxos baseados em bytes são chamados **arquivos binários** e arquivos criados usando fluxos baseados em caracteres são chamados **arquivos de texto**. Arquivos de texto podem ser lidos por editores de textos, enquanto arquivos binários são lidos por programas que entendem o conteúdo específico do arquivo e ordenamento desse conteúdo.

Entrada padrão, saída padrão e fluxos de erros padrão

Um programa Java **abre** um arquivo criando e associando um objeto ao fluxo de bytes ou de caracteres. As classes usadas para criar esses objetos são discutidas mais adiante. O Java também pode associar fluxos a diferentes dispositivos. De fato, o Java cria três objetos de fluxo que são associados a dispositivos quando um programa Java inicia a execução — `System.in`, `System.out` e `System.err`. `System.in` (o objeto de fluxo de entrada padrão) normalmente permite a um programa inserir bytes a partir do teclado; o objeto `System.out` (o objeto de fluxo de saída padrão) normalmente permite a um programa enviar a saída dos dados de caractere para a tela; e o objeto `System.err` (o

objeto fluxo de erro padrão) normalmente permite a um programa gerar a saída de mensagens de erros baseados em caractere na tela. Cada um desses fluxos pode ser redirecionado. Para `System.in`, essa capacidade permite ao programa ler bytes a partir de uma origem diferente. Para `System.out` e `System.err`, essa capacidade permite que a saída seja enviada para um local diferente, como a um arquivo em disco. A classe `System` fornece os métodos **`setIn`**, **`setOut`** e **`setErr`** para **redirecionar** os fluxos de entrada, saída e erro padrão, respectivamente.

O pacote `java.io`

Os programas Java realizam o processamento do arquivo utilizando classes do pacote **`java.io`**. Esse pacote inclui definições para classes de fluxo, como **`FileInputStream`** (para entrada baseada em bytes de um arquivo), **`FileOutputStream`** (para saída baseada em bytes de um arquivo), **`FileReader`** (para entrada baseada em caractere de um arquivo) e **`FileWriter`** (para saída baseada em caractere de um arquivo), que herda das classes `InputStream`, `OutputStream`, `Reader` e `Writer`, respectivamente (essas classes serão discutidas mais tarde neste capítulo). Assim, os métodos das classes stream também podem ser aplicados a fluxos de arquivo. Você abre um arquivo criando um objeto de uma dessas classes stream. O construtor do objeto interage com o sistema operacional para abrir o arquivo.

O Java contém classes que permitem realizar a entrada e saída de objetos ou variáveis de tipos de dados primitivos. Os dados ainda serão armazenados como bytes ou caracteres nos bastidores, permitindo que você leia ou grave dados na forma de `int`s, `String`s ou outros tipos sem ter de se preocupar com os detalhes da conversão desses valores em formato de bytes. Para realizar essa entrada e saída, os objetos das classes **`ObjectInputStream`** e **`ObjectOutputStream`** podem ser utilizados juntos com as classes de arquivos baseados em fluxos de bytes `FileInputStream` e `FileOutputStream` (essas classes serão discutidas em mais detalhes a seguir). A hierarquia completa das classes no pacote `java.io` poder ser visualizada na documentação on-line em

```
java.sun.com/javase/6/docs/api/java/io/package-tree.html
```

Cada classe é identada sob sua superclasse. Por exemplo, a classe `InputStream` é uma subclasse de `Object`. Para visualizar os detalhes de uma classe, clique no seu nome na hierarquia.

Como você pode ver na hierarquia, o Java oferece muitas classes para realizar operações de entrada/saída. Utilizamos várias dessas classes neste capítulo para implementar programas de processamento de arquivos que criam e manipulam arquivos de acesso sequencial. Também incluímos um exemplo detalhado da classe **`File`**, que é útil para obter as informações sobre arquivos e diretórios. No Capítulo 27, utilizamos extensamente classes de fluxo para implementar aplicativos de rede. Várias outras classes no pacote `java.io` que não utilizamos neste capítulo serão discutidas brevemente na Seção 17.7.

Além das classes `java.io`, a entrada e saída baseadas em caracteres podem ser executadas com as classes `Scanner` e **`Formatter`**. A classe `Scanner` é usada extensamente para a entrada de dados a partir do teclado. Essa classe também pode ler dados de um arquivo. A classe `Formatter` permite que os dados formatados sejam impressos em qualquer fluxo baseado em texto de uma maneira semelhante ao método `System.out.printf`. O Apêndice G apresenta os detalhes da saída formatada com `printf`. Todos esses recursos também podem ser utilizados para formatar arquivos de texto.

17.4 Classe `File`

Esta seção apresenta a classe `File`, particularmente útil para recuperar informações sobre arquivos ou diretórios em disco. Os objetos da classe `File` não abrem arquivos nem fornecem quaisquer capacidades de processamento de arquivos. Entretanto, os objetos `File` são utilizados frequentemente com objetos de outras classes `java.io` para especificar arquivos ou diretórios a manipular.

Criando objetos `File`

A classe `File` fornece quatro construtores. Aquele com um argumento `String` especifica o `name` de um arquivo ou diretório para associar com o objeto `File`. O `name` pode conter as **informações de caminho** bem como um nome de arquivo ou diretório. Um caminho de arquivo ou diretório especifica sua localização em disco. O caminho inclui alguns ou os principais diretórios para o arquivo ou diretório. Um **caminho absoluto** contém todos os diretórios, desde o **diretório-raiz**, que levam a um arquivo ou diretório específico. Cada arquivo ou diretório em uma unidade de disco particular tem o mesmo diretório-raiz em seu caminho. Um **caminho relativo** normalmente inicia a partir do diretório no qual o aplicativo começou a executar e, portanto, é "relativo" ao diretório atual. O construtor com dois argumentos `String` especifica um caminho absoluto ou relativo como o primeiro argumento e o arquivo ou diretório a associar com o objeto `File` como o segundo argumento. O construtor com os argumentos `File` e `String` utiliza um objeto `File` existente que especifica o diretório pai do arquivo ou diretório especificado pelo argumento `String`. O quarto construtor utiliza um objeto `URI` para localizar o arquivo. Um **Uniform Resource Identifier (URI)** é uma forma mais geral dos **Uniform Resource Locators (URLs)** que são utilizados para localizar sites na Web. Por exemplo, `http://www.deitel.com/` é o URL do site Web Deitel & Associates. URIs para localizar arquivos variam em diferentes sistemas operacionais. Em plataformas Windows, o URI

```
file://C:/data.txt
```

identifica o arquivo `data.txt` armazenado no diretório-raiz da unidade C: unidade. Em plataformas UNIX/Linux, o URI

```
file:/home/student/data.txt
```

identifica o arquivo `data.txt` armazenado no diretório `home` do usuário `student`.

Dica de prevenção de erro 17.1

*O método `File` usa **isFile** para determinar se um objeto `File` representa um arquivo (não um diretório) antes de tentar abri-lo.*

A Figura 17.3 lista alguns métodos `File` comuns. A relação completa pode ser vista em `java.sun.com/javase/6/docs/api/java/io/File.html`.

Método	Descrição
`boolean canRead()`	Retorna `true` se um arquivo for legível pelo aplicativo atual; `false` caso contrário.
`boolean canWrite()`	Retorna `true` se um arquivo for gravável pelo aplicativo atual; `false` caso contrário.
`boolean exists()`	Retorna `true` se o arquivo ou diretório representado pelo objeto `File` existir; `false` do contrário.
`boolean isFile()`	Retorna `true` se o nome especificou como o argumento para o `File` o construtor é um arquivo; caso contrário, `false`.
`boolean isDirectory()`	Retorna `true` se o nome especificado como o argumento para o construtor `File` é um diretório; caso contrário, `false`.
`boolean isAbsolute()`	Retorna `true` se os argumentos especificados para o construtor `File` indicam um caminho absoluto para um arquivo ou diretório; caso contrário, `false`.
`String getAbsolutePath()`	Retorna uma `String` com o caminho absoluto do arquivo ou diretório.
`String getName()`	Retorna uma `String` com o nome do arquivo ou diretório.
`String getPath()`	Retorna uma `String` com o caminho do arquivo ou diretório.
`String getParent()`	Retorna uma `String` com o diretório pai do arquivo ou diretório (isto é, o diretório em que o arquivo ou diretório está localizado).
`long length()`	Retorna o comprimento do arquivo, em bytes. Se o objeto `File` representa um diretório, um valor não especificado é retornado.
`long lastModified()`	Retorna uma representação dependente de plataforma da data/hora em que o arquivo ou diretório for modificado pela última vez. O valor retornado é útil somente para a comparação com outros valores retornados por esse método.
`String[] list()`	Retorna um array de `String`s que representam o conteúdo de um diretório. Retorna `null` se o objeto `File` não representar um diretório.

Figura 17.3 | Métodos `File`.

Demonstrando a classe `File`

A Figura 17.4 solicita que o usuário digite o nome de um arquivo ou diretório, utiliza então a classe `File` para imprimir as informações sobre o arquivo ou diretório.

```
1  // Figura 17.4: FileDemonstration.java
2  // A classe File utilizada para obter informações de arquivo e de diretório.
3  import java.io.File;
4  import java.util.Scanner;
5
6  public class FileDemonstration
7  {
8     public static void main( String[] args )
9     {
10       Scanner input = new Scanner( System.in );
11
12       System.out.print( "Enter file or directory name: " );
13       analyzePath( input.nextLine() );
14    } // fim de main
15
16    // exibe informações sobre o arquivo que o usuário especifica
17    public static void analyzePath( String path )
18    {
19       // cria o objeto File com base na entrada de usuário
```

```
20          File name = new File( path );
21
22          if (name.exists() ) // se o nome existir, gera saída das informações sobre ele
23          {
24             // exibe informações sobre o arquivo (ou diretório)
25             System.out.printf(
26                "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
27                name.getName(), " exists",
28                (name.isFile() ? "is a file" : "is not a file" ),
29                (name.isDirectory() ? "is a directory" :
30                   "is not a directory" ),
31                (name.isAbsolute() ? "is absolute path" :
32                   "is not absolute path" ), "Last modified: ",
33                name.lastModified(), "Length: ", name.length(),
34                "Path: ", name.getPath(), "Absolute path: ",
35                name.getAbsolutePath(), "Parent: ", name.getParent() );
36
37             if (name.isDirectory() ) // listagem de diretório de saída
38             {
39                String[] directory = name.list();
40                System.out.println( "\n\nDirectory contents:\n" );
41
42                for ( String directoryName : directory )
43                   System.out.println( directoryName );
44             } // fim do if
45          } // fim do if externo
46          else // não for arquivo ou diretório, gera saída da mensagem de erro
47          {
48             System.out.printf( "%s %s", path, "does not exist." );
49          } // fim de else
50       } // fim do método analyzePath
51    } // fim da classe FileDemonstration
```

```
Enter file or directory name: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1228404395024
Length: 4096
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc
Parent: E:\Program Files\Java\jdk1.6.0_11\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Laffy
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
SwingSet3

Enter file or directory name: C:\Program Files\Java\jdk1.6.0_11\demo\jfc \Java2D\README.txt
README.txt exists
is a file
is not a directory
is absolute path
Last modified: 1228404384270
Length: 7518
Path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Absolute path: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D\README.txt
Parent: E:\Program Files\Java\jdk1.6.0_11\demo\jfc\Java2D
```

Figura 17.4 | A classe `File` utilizada para obter informações de arquivo e de diretório.

O programa começa solicitando ao usuário um arquivo ou diretório (linha 12). A linha 13 gera as entradas do nome de arquivo ou nome de diretório e passa-o para o método `analyzePath` (linhas 17–50). O método cria um novo objeto `File` (linha 20) e atribui sua referência a `name`. A linha 22 invoca o método `File` `exists` para determinar se o nome inserido pelo usuário existe (como um arquivo ou diretório) no disco. Se o nome não existir, o controle prosseguirá para as linhas 46–49 e exibirá uma mensagem na tela contendo o nome que o usuário digitou, seguido por "`does not exist`". Caso contrário, a instrução `if` (linhas 22–45) executa. O programa envia para a saída o nome do arquivo ou diretório (linha 27), seguido pelos resultados do teste do objeto `File` com `isFile` (linha 28), `isDirectory` (linha 29) e `isAbsolute` (linha 31). Em seguida, o programa exibe os valores retornandos por `lastModified` (linha 33), `length` (linha 33), `getPath` (linha 34), `getAbsolutePath` (linha 35) e `getParent` (linha 35). Se o objeto `File` representar um diretório (linha 37), o programa obterá uma lista do conteúdo do diretório como um array de `Strings` utilizando o método `File` `list` (linha 39) e exibirá a lista na tela.

A primeira saída desse programa demonstra um objeto `File` associado com o diretório `jfc` do JDK. A segunda saída demonstra um objeto `File` associado com o arquivo `README.txt` do exemplo do Java 2D que vem com o JDK. Em ambos os casos, especificamos um caminho absoluto em nosso computador.

Um **caractere separador** é utilizado para separar diretórios e arquivos no caminho. Em um computador Windows, o caractere separador é uma barra invertida (\). Em um sistema UNIX, é uma barra (/). O Java processa esses dois caracteres de maneira idêntica em um nome de caminho. Por exemplo, se fôssemos utilizar o caminho

```
c:\Program Files\Java\jdk1.6.0_11\demo/jfc
```

que emprega cada caractere separador, ainda assim o Java processaria o caminho adequadamente. Ao criar `Strings` que representam informações de caminho, utilize `File.separator` para obter caractere de separador adequado do computador local, em vez de utilizar explicitamente /.ou \. Essa constante retorna uma `String` que consiste em um caractere — o separador adequado para o sistema.

Erro comum de programação 17.1

Utilizar \ como um separador de diretório em vez de \\ em uma literal de string é um erro de lógica. Uma \ simples indica que a \ seguida pelo próximo caractere representa uma sequência de escape. Utilize \\ para inserir um \ em uma literal de string.

17.5 Arquivos de texto de acesso sequencial

Em seguida, criamos e manipulamos arquivos de acesso sequencial nos quais os registros são armazenados na ordem pelo campo de chave de registro. Iniciamos com arquivos de texto, permitindo que o leitor crie e edite rapidamente arquivos legíveis por seres humanos. Discutimos como criar, gravar dados, ler dados e atualizar arquivos de texto de acesso sequencial. Também incluímos um programa de consulta de crédito que recupera dados específicos em um arquivo.

17.5.1 Criando um arquivo de texto de acesso sequencial

O Java não impõe nenhuma estrutura a um arquivo — noções, como registros, não fazem parte da linguagem Java. Portanto, você deve estruturar os arquivos para satisfazer os requisitos dos seus aplicativos. No exemplo a seguir, veremos como impor uma estrutura de registro chaveado a um arquivo.

O programa nas figuras 17.5, 17.6 e 17.8 cria um arquivo simples de acesso sequencial que poderia ser utilizado em um sistema de contas a receber para monitorar os valores devidos pelos seus clientes a uma empresa. Para cada cliente, o programa obtém do usuário um número de conta e o nome além do saldo do cliente (isto é, o valor que o cliente deve à empresa por bens e serviços recebidos). Os dados de cada cliente constituem "um registro" para esse cliente. Esse aplicativo utiliza o número de conta como a chave de registro — o arquivo será criado e mantido na ordem do número de conta. O programa assume que o usuário insere os registros em ordem de número de conta. Em um sistema abrangente de contas a receber (baseado em arquivos de acesso sequencial), seria fornecido um recurso de classificação de modo que o usuário pudesse inserir o registro em qualquer ordem. Os registros seriam então classificados e gravados no arquivo.

Classe AccountRecord

A classe `AccountRecord` (Figura 17.5) encapsula as informações sobre o registro do cliente utilizadas pelos exemplos neste capítulo. `AccountRecord` é declarada no pacote `com.deitel.ch17` (linha 3) para que possa ser importado em vários exemplos deste capítulo para reutilização. (A Seção 8.14 fornece informações sobre como compilar e utilizar seus próprios pacotes.) A classe `AccountRecord` contém as variáveis de instância `private` `account`, `firstName`, `lastName` e `balance` (linhas 7–10) e os métodos *set* e *get* para acessar esses campos. Embora os métodos *set* não validem os dados nesse exemplo, eles devem fazer isso em um sistema de força industrial.

```
1  // Figura 17.5: AccountRecord.java
2  // A classe AccountRecord mantém informações para uma conta.
3  package com.deitel.ch17; // empacotada para reutilização
4
5  public class AccountRecord
```

```
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

   // construtor sem argumentos chama outro construtor com valores padrão
   public AccountRecord()
   {
      this( 0, "", "", 0.0 ); // chama o construtor com quatro argumentos
   } // fim do construtor de AccountRecord sem argumentos

   // inicializa um registro
   public AccountRecord( int acct, String first, String last, double bal )
   {
      setAccount( acct );
      setFirstName( first );
      setLastName( last );
      setBalance( bal );
   } // fim do construtor de AccountRecord com quatro argumentos

   // configura o número de conta
   public void setAccount( int acct )
   {
      account = acct;
   } // fim do método setAccount

   // obtém número de conta
   public int getAccount()
   {
      return account;
   } // fim do método getAccount

   // configura o nome
   public void setFirstName( String first )
   {
      firstName = first;
   } // fim do método setFirstName

   // obtém o primeiro nome
   public String getFirstName()
   {
      return firstName;
   } // fim do método getFirstName

   // configura o sobrenome
   public void setLastName( String last )
   {
      lastName = last;
   } // fim do método setLastName

   // obtém o ultimo nome
   public String getLastName()
   {
      return lastName;
   } // fim do método getLastName

   // configura saldo
   public void setBalance( double bal )
   {
      balance = bal;
   } // fim do método setBalance

   // obtém saldo
   public double getBalance()
   {
      return balance;
   } // fim do método getBalance
} // fim da classe AccountRecord
```

Figura 17.5 | A classe `AccountRecord` mantém informações de uma conta.

Para compilar a classe `AccountRecord`, abra uma janela de comando, altere os diretórios para o diretório `fig17_05` deste capítulo (que contém `AccountRecord.java`) e, então, digite:

```
javac -d .. AccountRecord.java
```

Isso insere `AccountRecord.class` na estrutura de diretórios do pacote e insere o pacote na pasta `ch17` que contém todos os exemplos deste capítulo. Ao compilar a classe `AccountRecord` (ou quaisquer outras classes que serão reutilizadas neste capítulo), você deve colocá-las em um diretório comum. Ao compilar ou executar as classes que utilizam a classe `AccountRecord` (por exemplo, `CreateTextFile` na Figura 17.6), você deve especificar o argumento de linha de comando `-classpath` para `javac` e `java`, como em

```
javac -classpath .;c:\examples\ch17 CreateTextFile.java
java -classpath .;c:\examples\ch17 CreateTextFile
```

Observe que o diretório atual (especificado com `.`) foi incluído no classpath para assegurar que o compilador possa localizar outras classes no mesmo diretório que o da classe sendo compilada. O separador de caminho utilizado nos comandos precedentes deve ser apropriado à sua plataforma — um ponto-e-vírgula (`;`) no Windows e um dois-pontos (`:`) no UNIX/Linux/Mac OS X. Os comandos precedentes supõem que o pacote que contém `AccountRecord` esteja localizado no diretório `C:\examples\ch17` em um computador Windows.

Classe `CreateTextFile`

Agora vamos examinar a classe `CreateTextFile` (Figura 17.6). A linha 14 declara a variável `Formatter output`. Como discutido na Seção 17.3, um objeto `Formatter` gera saída para `String`s formatadas, usando as mesmas capacidades de formato do método `System.out.printf`. Um objeto `Formatter` pode gerar saída para vários locais, tela ou arquivo, como ocorre aqui. O objeto `Formatter` é instanciado na linha 21 no método `openFile` (linhas 17–34). O construtor utilizado na linha 21 recebe um argumento — uma `String` contendo o nome do arquivo, incluindo seu caminho. Se um caminho não for especificado, como é o caso aqui, a JVM assume que o arquivo está no diretório a partir do qual o programa foi executado. Para arquivos de texto, utilizamos a extensão de arquivo `.txt`. Se o arquivo não existir, ele será criado. Se um arquivo existente for aberto, seu conteúdo será **truncado** — todos os dados no arquivo serão descartados. Nesse ponto, o arquivo é aberto para gravação e o objeto `Formatter` resultante pode ser utilizado para gravar dados no arquivo.

```
1  // Figura 17.6: CreateTextFile.java
2  // Gravando dados em um arquivo de texto sequencial com a classe Formatter.
3  import java.io.FileNotFoundException;
4  import java.lang.SecurityException;
5  import java.util.Formatter;
6  import java.util.FormatterClosedException;
7  import java.util.NoSuchElementException;
8  import java.util.Scanner;
9
10 import com.deitel.ch17.AccountRecord;
11
12 public class CreateTextFile
13 {
14    private Formatter output; // objeto utilizado para gerar saída de texto no arquivo
15
16    // permite ao usuário abrir o arquivo
17    public void openFile()
18    {
19       try
20       {
21          output = new Formatter( "clients.txt" ); // abre o arquivo
22       } // fim do try
23       catch ( SecurityException securityException )
24       {
25          System.err.println(
26             "You do not have write access to this file." );
27          System.exit( 1 ); // termina o programa
28       } // fim do catch
29       catch ( FileNotFoundException fileNotFoundException )
30       {
31          System.err.println( "Error opening or creating file." );
32          System.exit( 1 ); // termina o programa
33       } // fim do catch
34    } // fim do método openFile
35
36    // adiciona registros ao arquivo
37    public void addRecords()
```

```
38      {
39         // objeto a ser gravado no arquivo
40         AccountRecord record = new AccountRecord();
41
42         Scanner input = new Scanner( System.in );
43
44         System.out.printf( "%s\n%s\n%s\n%s\n\n",
45            "To terminate input, type the end-of-file indicator ",
46            "when you are prompted to enter input.",
47            "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
48            "On Windows type <ctrl> z then press Enter" );
49
50         System.out.printf( "%s\n%s",
51            "Enter account number (> 0), first name, last name and balance.",
52            "? " );
53
54         while (input.hasNext() ) // faz um loop até o indicador de fim de arquivo
55         {
56            try // gera saída dos valores para o arquivo
57            {
58               // recupera os dados para saída
59               record.setAccount( input.nextInt() ); // lê o número de conta
60               record.setFirstName( input.next() ); // lê o primeiro nome
61               record.setLastName( input.next() ); // lê o sobrenome
62               record.setBalance( input.nextDouble() ); // lê o saldo
63
64               if ( record.getAccount() > 0 )
65               {
66                  // grava um novo registro
67                  output.format( "%d %s %s %.2f\n", record.getAccount(),
68                     record.getFirstName(), record.getLastName(),
69                     record.getBalance() );
70               } // fim do if
71               else
72               {
73                  System.out.println(
74                     "Account number must be greater than 0." );
75               } // fim de else
76            } // fim do try
77            catch ( FormatterClosedException formatterClosedException )
78            {
79               System.err.println( "Error writing to file." );
80               return;
81            } // fim do catch
82            catch ( NoSuchElementException elementException )
83            {
84               System.err.println( "Invalid input. Please try again." );
85               input.nextLine(); // descarta a entrada para que o usuário possa tentar novamente
86            } // fim do catch
87
88            System.out.printf( "%s %s\n%s", "Enter account number (>0),",
89               "first name, last name and balance.", "? " );
90         } // fim do while
91      } // fim do método addRecords
92
93      // fecha o arquivo
94      public void closeFile()
95      {
96         if ( output != null )
97            output.close();
98      } // fim do método closeFile
99   } // fim da classe CreateTextFile
```

Figura 17.6 | Gravando dados em um arquivo de texto sequencial com a classe `Formatter`.

As linhas 23–28 tratam da **`SecurityException`**, que ocorre se o usuário não tiver permissão de gravar dados no arquivo. As linhas 29–33 tratam da **`FileNotFoundException`**, que ocorre se o arquivo não existir e um novo arquivo não puder ser criado. Essa exceção também pode ocorrer se houver um erro ao abrir o arquivo. Observe que nos dois handlers de exceção, chamamos o método `static` **`System.exit`** e passamos o valor `1`. Esse método termina o aplicativo. Um argumento de `0` para o método `exit` indica terminação bem-

-sucedida do programa. Um valor de não zero, como `1` nesse exemplo, normalmente indica que ocorreu um erro. Esse valor é passado à janela de comando que executou o programa. O argumento é útil se o programa for executado de um **arquivo em lote** em sistemas Windows ou de um **script de shell** nos sistemas UNIX/Linux/Mac OS X. Os arquivos em lote e scripts de shell oferecem uma maneira conveniente de executar vários programas em sequência. Quando o primeiro programa termina, o próximo programa inicia a execução. É possível utilizar o argumento para o método `exit` em um arquivo em lote ou script de shell a fim de determinar se outros programas devem ser executados. Para mais informações sobre arquivos de lote ou scripts de shell, veja a documentação do seu sistema operacional.

O método `addRecords` (linhas 37–91) pede que o usuário insira os vários campos para cada registro ou inserir a sequência e teclas de fim de arquivo quando a entrada de dados estiver completa. A Figura 17.7 lista as combinações de teclas para inserir o fim de arquivo para vários sistemas de computador.

Sistema operacional	Combinação de teclas
UNIX/Linux/Mac OS X	*<Enter> <Ctrl> d*
Windows	*<Ctrl> z*

Figura 17.7 | Combinações de chaves de fim de arquivo.

A linha 40 cria um objeto `AccountRecord`, que será utilizado para armazenar os valores do registro atual inseridos pelo usuário. A linha 42 cria um objeto `Scanner` para ler a entrada de usuário a partir do teclado. As linhas 44–48 e 50–52 solicitam a entrada do usuário.

A linha 54 utiliza o método `Scanner` `hasNext` para determinar se a combinação de teclas de fim de arquivo foi inserida. O loop executa até que `hasNext` encontre o fim de arquivo.

As linhas 59–62 leem os dados do usuário, armazenando informações de registro no objeto `AccountRecord`. Cada instrução lança uma **`NoSuchElementException`** (tratada nas linhas 82–86) se os dados estiverem no formato errado (por exemplo, uma `String` quando um `int` é esperado) ou se não houver mais dados para saída. Se o número de conta for maior do que 0 (linha 64), as informações do registro são gravadas em `clients.txt` (linhas 67–69) empregando o método **`format`**, que pode executar formatação idêntica ao método `System.out.printf` utilizado extensivamente nos capítulos anteriores. O método `format` gera saída para uma `String` formatada ao destino de saída do objeto `Formatter` — o arquivo `clients.txt`. A string de formato `"%d %s %s %.2f\n"` indica que o registro atual será armazenado como um inteiro (o número de conta) seguido por uma `String` (o primeiro nome), outra `String` (o sobrenome) e um valor de ponto flutuante (o saldo). Cada informação é separada da seguinte por um espaço e a saída do valor de double (o saldo) é gerada com dois dígitos à direita do ponto de fração decimal (como indicado pelo `.2` em `%.2f`). Os dados no arquivo de texto podem ser visualizados com um editor de textos ou recuperados mais tarde por um programa projetado para ler o arquivo (Seção 17.5.2).

Quando as linhas 67–69 executam, se o objeto `Formatter` for fechado, uma **`FormatterClosedException`** será lançada. Essa exceção é tratada nas linhas 77–81. [*Nota:* você também pode gerar saída de dados para um arquivo de texto utilizando a classe **`java.io.PrintWriter`**, que fornece os métodos `format` e `printf` para gerar saída de dados formatados.]

As linhas 94–98 declaram o método `closeFile`, que fecha o `Formatter` e o arquivo de saída subjacente. A linha 97 fecha o objeto simplesmente chamando o método **`close`**. Se o método `close` não for chamado explicitamente, o sistema operacional normalmente fechará o arquivo quando a execução do programa termina — esse é um exemplo da "faxina" feita pelo sistema operacional. Mas você sempre deve fechar explicitamente um arquivo quando ele não for mais necessário.

Caracteres separadores de linha específicos à plataforma

As linhas 67–69 geram saída de uma linha do texto seguida por uma nova linha (`\n`). Se utilizar um editor de textos para abrir o arquivo `clients.txt` produzido, cada registro talvez não seja exibido em uma linha separada. Por exemplo, no Notepad (Microsoft Windows), os usuários verão uma linha contínua do texto. Isso ocorre porque plataformas distintas utilizam diferentes caracteres separadores de linha. No UNIX/Linux/Mac OS X, o separador de linha é uma nova linha (`\n`). No Windows, é uma combinação de um retorno de carro e um feed de linha — representado como `\r\n`. Você pode utilizar o especificador de formato **`%n`** em uma string de controle de formato para imprimir um separador de linha específico à plataforma, assegurando assim que o arquivo de texto possa ser aberto e visualizado corretamente em um editor de textos para a plataforma na qual o arquivo foi criado. Observe que o método `System.out.println` gera saída a um separador de linha específico à plataforma depois do seu argumento. Além disso, observe que independentemente do separador de linha utilizado em um arquivo de texto, um programa Java ainda pode reconhecer as linhas do texto e lê-las.

Classe CreateTextFileTest

A Figura 17.8 executa o programa. A linha 8 cria um objeto `CreateTextFile`, que é então utilizado para abrir, adicionar registros e fechar o arquivo (linhas 10–12). Os dados de exemplo para esse aplicativo são mostrados na Figura 17.9. Na execução de exemplo desse programa, o usuário insere informações para cinco contas e depois insere o fim de arquivo para sinalizar que a entrada de dados está completa. A execução de exemplo não mostra como os registros de dados na verdade aparecem no arquivo. Na próxima seção, para verificar se o arquivo foi criado com sucesso, apresentamos um programa que lê o arquivo e imprime seu conteúdo. Como é um arquivo de texto, você também pode verificar as informações abrindo o arquivo em um editor de textos.

```
1  // Figura 17.8: CreateTextFileTest.java
2  // Testando a classe CreateTextFile.
3
4  public class CreateTextFileTest
5  {
6     public static void main( String[] args )
7     {
8        CreateTextFile application = new CreateTextFile();
9
10       application.openFile();
11       application.addRecords();
12       application.closeFile();
13    } // fim de main
14 } // fim da classe CreateTextFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter
Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

Figura 17.8 | Testando a classe `CreateTextFile`.

Dados de exemplo			
100	Bob	Jones	24.98
200	Steve	Doe	-345.67
300	Pam	White	0.00
400	Sam	Stone	-42.16
500	Sue	Rich	224.62

Figura 17.9 | Dados de exemplo para o programa nas Figuras 17.6–17.8.

17.5.2 Lendo dados a partir de um arquivo de texto de acesso sequencial

Os dados são armazenados em arquivos de modo que possam ser recuperados para processamento quando necessário. A Seção 17.5.1 demonstrou como criar um arquivo de acesso sequencial. Esta seção mostra como ler dados sequencialmente em um arquivo de texto. Demonstramos como a classe `Scanner` pode ser utilizada para inserir dados a partir de um arquivo em vez de utilizar o teclado.

O aplicativo nas figuras 17.10 e 17.11 lê os registros no arquivo `"clients.txt"` criados pelo aplicativo da Seção 17.5.1 e exibe o conteúdo do registro. A linha 13 da Figura 17.10 declara um `Scanner` que será utilizado para recuperar a entrada no arquivo.

```
1  // Figura 17.10: ReadTextFile.java
2  // Esse programa lê um arquivo de texto e exibe cada registro.
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.lang.IllegalStateException;
6  import java.util.NoSuchElementException;
```

```
7  import java.util.Scanner;
8
9  import com.deitel.ch17.AccountRecord;
10
11 public class ReadTextFile
12 {
13    private Scanner input;
14
15    // permite ao usuário abrir o arquivo
16    public void openFile()
17    {
18       try
19       {
20          input = new Scanner( new File( “clients.txt” ) );
21       } // fim do try
22       catch ( FileNotFoundException fileNotFoundException )
23       {
24          System.err.println( "Error opening file." );
25          System.exit( 1 );
26       } // fim do catch
27    } // fim do método openFile
28
29    // lê o registro no arquivo
30    public void readRecords()
31    {
32       // objeto a ser gravado na tela
33       AccountRecord record = new AccountRecord();
34
35       System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
36          "First Name", "Last Name", "Balance" );
37
38       try // lê os registros no arquivo utilizando o objeto Scanner
39       {
40          while (input.hasNext() )
41          {
42             record.setAccount( input.nextInt() ); // lê o número de conta
43             record.setFirstName( input.next() ); // lê o primeiro nome
44             record.setLastName( input.next() ); // lê o sobrenome
45             record.setBalance( input.nextDouble() ); // lê o saldo
46
47             // exibe o conteúdo de registro
48             System.out.printf( “%-10d%-12s%-12s%10.2f\n”,
49                record.getAccount(), record.getFirstName(),
50                record.getLastName(), record.getBalance() );
51          } // fim do while
52       } // fim do try
53       catch ( NoSuchElementException elementException )
54       {
55          System.err.println( "File improperly formed." );
56          input.close();
57          System.exit( 1 );
58       } // fim do catch
59       catch ( IllegalStateException stateException )
60       {
61          System.err.println( "Error reading from file." );
62          System.exit( 1 );
63       } // fim do catch
64    } // fim do método readRecords
65
66    // fecha o arquivo e termina o aplicativo
67    public void closeFile()
68    {
69       if ( input != null )
70          input.close(); // fecha o arquivo
71    } // fim do método closeFile
72 } // fim da classe ReadTextFile
```

Figura 17.10 | Leitura de arquivo sequencial utilizando um `Scanner`.

```
1  // Figura 17.11: ReadTextFileTest.java
2  // Testando a classe ReadTextFile.
3
4  public class ReadTextFileTest
5  {
6     public static void main( String[] args )
7     {
8        ReadTextFile application = new ReadTextFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // fim de main
14 } // fim da classe ReadTextFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White            0.00
400       Sam         Stone          -42.16
500       Sue         Rich           224.62
```

Figura 17.11 | Testando a classe `ReadTextFile`.

O método `openFile` (linhas 16–27) abre o arquivo para leitura instanciando um objeto `Scanner` na linha 20. Passamos um objeto `File` para o construtor, o qual especifica que o objeto `Scanner` vai ler do arquivo `"clients.txt"` localizado no diretório em que o aplicativo é executado. Se o arquivo não puder ser localizado, ocorrerá uma `FileNotFoundException`. O tratamento de exceções nas linhas 22–26.

O método `readRecords` (linhas 30–64) lê e exibe os registros no arquivo. A linha 33 cria objeto `AccountRecord` `record` para armazenar informações do registro atual. As linhas 35–36 exibem os cabeçalhos das colunas na saída do aplicativo. As linhas 40–51 leem os dados no arquivo até que o marcador de fim de arquivo seja alcançado (em que caso, o método `hasNext` retornará `false` na linha 40). As linhas 42–45 utilizam os métodos `Scanner` `nextInt`, `next` e `nextDouble` para inserir um `int` (o número de conta), duas `Strings` (os nomes e sobrenomes) e um valor `double` (o saldo). Cada registro é uma linha de dados no arquivo. Os valores são armazenados no objeto `record`. Se as informações no arquivo não estiverem adequadamente formatadas (por exemplo, há um sobrenome no qual deveria haver um saldo), ocorrerá uma `NoSuchElementException` quando o registro é inserido. Essa exceção é tratada nas linhas 53–58. Se `Scanner` foi fechado antes de os dados serem inseridos, uma **`IllegalStateException`** ocorrerá (tratada nas linhas 59–63). Se nenhuma exceção ocorrer, as informações do registro são exibidas na tela (linhas 48–50). Observe na string de formato na linha 48 que o número da conta, o primeiro nome e o sobrenome são alinhados à esquerda, enquanto o saldo é justificado à direita e enviado para a saída com dois dígitos de precisão. Cada iteração do loop insere uma linha de texto no arquivo de texto, que representa um registro.

As linhas 67–71 definem o método `closeFile`, que fecha o `Scanner`. O método `main` é definido na Figura 17.11, nas linhas 6–13. A linha 8 cria um objeto `ReadTextFile`, que é então utilizado para abrir, adicionar registros e fechar o arquivo (linhas 10–12).

17.5.3 Estudo de caso: um programa de consulta de crédito

Para recuperar dados sequencialmente de um arquivo, os programas começam no início do arquivo e leem todos os dados consecutivamente até que as informações desejadas sejam encontradas. Talvez seja necessário processar o arquivo várias vezes sequencialmente (a partir do início do arquivo) durante a execução de um programa. A classe `Scanner` não permite reposicionamento no início do arquivo. Se for necessário ler o arquivo novamente, o programa deverá fechar e reabrir o arquivo.

O programa nas figuras 17.12–17.14 permite que um gerente de crédito obtenha listas de clientes com saldo zero (isto é, clientes que não devem nada à empresa), clientes com saldos credores (isto é, os clientes aos quais a empresa deve dinheiro) e clientes com saldos devedores (isto é, os clientes que devem à empresa por bens e serviços recebidos). Um saldo credor é um valor monetário negativo e um saldo devedor, um valor positivo.

Enumeração `MenuOption`

Iniciamos criando um tipo `enum` (Figura 17.12) para definir as diferentes opções de menu que o usuário terá. As opções e seus valores são listados nas linhas 7–10. O método `getValue` (linhas 19–22) recupera o valor de uma constante `enum` específica.

```
1  // Figura 17.12: MenuOption.Java
2  // Enumeração para as opções do programa de consulta de crédito.
3
4  public enum MenuOption
5  {
6     // declara o conteúdo do tipo enum
7     ZERO_BALANCE( 1 ),
8     CREDIT_BALANCE( 2 ),
9     DEBIT_BALANCE( 3 ),
10    END( 4 );
11
12    private final int value; // opção atual de menu
13
14    MenuOption( int valueOption )
15    {
16       value = valueOption;
17    } // fim do construtor do enum de MenuOptions
18
19    public int getValue()
20    {
21       return value;
22    } // fim do método getValue
23 } // fim do enum de MenuOption
```

Figura 17.12 | Enumeração das opções de menu do programa de consulta de crédito.

Classe `CreditInquiry`

A Figura 17.13 contém a funcionalidade para o programa de consulta de crédito e a Figura 17.14 contém o método `main` que executa o programa. O programa exibe um menu de texto e permite ao gerente de crédito inserir uma de três opções para obter informações de crédito. A opção 1 (`ZERO_BALANCE`) exibe as contas com saldos zero. A opção 2 (`CREDIT_BALANCE`) exibe as contas com saldos positivos. A opção 3 (`DEBIT_BALANCE`) exibe as contas com saldos negativos. A opção 4 (`END`) termina a execução do programa.

```
1  // Figura 17.13: CreditInquiry.java
2  // Esse programa lê um arquivo sequencialmente e exibe o
3  // conteúdo com base no tipo de conta que o usuário solicita
4  // (saldo credor, saldo devedor ou saldo zero).
5  import java.io.File;
6  import java.io.FileNotFoundException;
7  import java.lang.IllegalStateException;
8  import java.util.NoSuchElementException;
9  import java.util.Scanner;
10
11 import com.deitel.ch17.AccountRecord;
12
13 public class CreditInquiry
14 {
15    private MenuOption accountType;
16    private Scanner input;
17    private final static MenuOption[] choices = { MenuOption.ZERO_BALANCE,
18       MenuOption.CREDIT_BALANCE, MenuOption.DEBIT_BALANCE,
19       MenuOption.END };
20
21    // lê registros de arquivo e exibe somente os registros do tipo apropriado
22    private void readRecords()
23    {
24       // objeto para armazenar dados que serão gravados no arquivo
25       AccountRecord record = new AccountRecord();
26
27       try // lê registros
28       {
29          // abre o arquivo para leitura a partir do início
30          input = new Scanner( new File( "clients.txt" ) );
31
32          while (input.hasNext() ) // insere os valores do arquivo
33          {
```

```
34              record.setAccount( input.nextInt() ); // lê o número de conta
35              record.setFirstName( input.next() ); // lê o primeiro nome
36              record.setLastName( input.next() ); // lê o sobrenome
37              record.setBalance( input.nextDouble() ); // lê o saldo
38
39              // se o tipo for a conta adequada, exibe o registro
40              if ( shouldDisplay( record.getBalance() ) )
41                 System.out.printf( "%-10d%-12s%-12s%10.2f\n",
42                    record.getAccount(), record.getFirstName(),
43                    record.getLastName(), record.getBalance() );
44           } // fim do while
45        } // fim do try
46        catch ( NoSuchElementException elementException )
47        {
48           System.err.println( "File improperly formed." );
49           input.close();
50           System.exit( 1 );
51        } // fim do catch
52        catch ( IllegalStateException stateException )
53        {
54           System.err.println( "Error reading from file." );
55           System.exit( 1 );
56        } // fim do catch
57        catch ( FileNotFoundException fileNotFoundException )
58        {
59           System.err.println( "File cannot be found." );
60           System.exit( 1 );
61        } // fim do catch
62        finally
63        {
64           if ( input != null )
65              input.close(); // fecha o Scanner e o arquivo
66        } // fim de finally
67     } // fim do método readRecords
68
69     // utiliza o tipo de registro para determinar se registro deve ser exibido
70     private boolean shouldDisplay( double balance )
71     {
72        if ( ( accountType == MenuOption.CREDIT_BALANCE )
73           && ( balance < 0 ) )
74           return true;
75
76        else if ( ( accountType == MenuOption.DEBIT_BALANCE )
77           && ( balance > 0 ) )
78           return true;
79
80        else if ( ( accountType == MenuOption.ZERO_BALANCE )
81           && ( balance == 0 ) )
82           return true;
83
84        return false;
85     } // fim do método shouldDisplay
86
87     // obtém a solicitação do usuário
88     private MenuOption getRequest()
89     {
90        Scanner textIn = new Scanner( System.in );
91        int request = 1;
92
93        // exibe opções de solicitação
94        System.out.printf( "\n%s\n%s\n%s\n%s\n%s\n",
95           "Enter request", " 1 - List accounts with zero balances",
96           " 2 - List accounts with credit balances",
97           " 3 - List accounts with debit balances", " 4 - End of run" );
98
99        try // tenta inserir a escolha de menu
100       {
101          do // insere a solicitação de usuário
102          {
103             System.out.print( "\n? " );
104             request = textIn.nextInt();
105          } while ( ( request < 1 ) || ( request > 4 ) );
106       } // fim do try
```

```
107          catch ( NoSuchElementException elementException )
108          {
109             System.err.println( "Invalid input." );
110             System.exit( 1 );
111          } // fim do catch
112
113          return choices[ request - 1 ]; // retorna o valor enum da opção
114       } // fim do método getRequest
115
116       public void processRequests()
117       {
118          // obtém a solicitação do usuário (por exemplo, saldo zero, credor ou devedor)
119          accountType = getRequest();
120
121          while ( accountType != MenuOption.END )
122          {
123             switch ( accountType )
124             {
125                case ZERO_BALANCE:
126                   System.out.println( "\nAccounts with zero balances:\n" );
127                   break;
128                case CREDIT_BALANCE:
129                   System.out.println( "\nAccounts with credit balances:\n" );
130                   break;
131                case DEBIT_BALANCE:
132                   System.out.println( "\nAccounts with debit balances:\n" );
133                   break;
134             } // fim do switch
135
136             readRecords();
137             accountType = getRequest();
138          } // fim do while
139       } // fim do método processRequests
140    } // fim da classe CreditInquiry
```

Figura 17.13 | Programa de consulta de crédito.

```
1  // Figura 17.14: CreditInquiryTest.java
2  // Esse programa testa a classe CreditInquiry.
3
4  public class CreditInquiryTest
5  {
6     public static void main( String[] args )
7     {
8        CreditInquiry application = new CreditInquiry();
9        application.processRequests();
10    } // fim de main
11 } // fim da classe CreditInquiryTest
```

Figura 17.14 | Testando a classe `CreditInquiry`.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 1

Accounts with zero balances:

300       Pam         White             0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
```

```
? 2

Accounts with credit balances:
200       Steve       Doe           -345.67
400       Sam         Stone          -42.16

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run

? 3

Accounts with debit balances:
100       Bob         Jones           24.98
500       Sue         Rich           224.62

? 4
```

Figura 17.15 | A saída de exemplo do programa de consulta de crédito na Figura 17.14.

As informações sobre o registro são coletadas lendo o arquivo do princípio ao fim e determinando se cada registro satisfaz os critérios para o tipo de conta selecionado. O método `processRequests` (linhas 116–139 da Figura 17.13) chama o método `getRequest` para exibir as opções de menu (linha 119), converte o número digitado pelo usuário. As linhas 121–138 fazem um loop até que o usuário especifique que o programa deve terminar. As linhas 123–134 exibem um cabeçalho para o conjunto atual de registros a ser impresso na tela. A linha 136 chama o método `readRecords` (linhas 22–67), que faz o loop pelo arquivo e lê cada registro.

A linha 30 do método `readRecords` abre o arquivo para ler com um `Scanner`. Observe que o arquivo será aberto para leitura com um novo objeto `Scanner` toda vez que esse método é chamado de modo que possamos ler novamente do início do arquivo. As linhas 34–37 leem um registro. A linha 40 chama o método `shouldDisplay` (linhas 70–85) para determinar se o registro atual satisfaz o tipo de conta solicitado. Se `shouldDisplay` retornar `true`, o programa exibirá as informações de conta. Quando o marcador de fim de arquivo for alcançado, o loop termina e a linha 65 chama o método `close` de `Scanner` para fechar `Scanner` e o arquivo. Observe que isso ocorre em um bloco `finally`, que será executado independentemente de o arquivo ter sido lido ou não com sucesso. Depois que todos os registros foram lidos, o controle retorna ao método `processRequests` e `getRequest` é novamente chamado (linha 137) para recuperar a próxima opção de menu do usuário. A Figura 17.14 contém o método `main` e chama o método `processRequests` na linha 9.

17.5.4 Atualizando arquivos de acesso sequencial

Os dados em muitos arquivos sequenciais não podem ser modificados sem o risco de destruir outros dados no arquivo. Por exemplo, se for necessário alterar o nome "`White`" para "`Worthington`", o nome antigo simplesmente não poderá ser sobrescrito, porque o novo nome requer mais espaço. O registro para `White` foi gravado no arquivo como

```
300 Pam White 0.00
```

Se o registro fosse regravado começando no mesmo local no arquivo utilizando o novo nome, o registro seria

```
300 Pam Worthington 0.00
```

O novo registro é maior (tem mais caracteres) que o registro original. Os caracteres além do segundo "`o`" em "`Worthington`" sobrescreverá o começo do próximo registro sequencial no arquivo. O problema aqui é que os campos em um arquivo de texto — e, consequentemente, registros — podem variar de tamanho. Por exemplo, 7, 14, –117, 2074 e 27383 são todos `ints` armazenados no mesmo número de bytes (4) internamente, mas são campos com diferentes tamanhos quando exibidos na tela ou gravados em um arquivo como texto. Portanto, registros em um arquivo de acesso sequencial normalmente não são atualizados no lugar. Em vez disso, o arquivo inteiro é normalmente regravado. Para fazer a alteração no nome anterior, os registros antes de `300 Pam White 0.00` seriam copiados para um novo arquivo, o novo registro (que pode ter um tamanho diferente daquele que o substitui) seria gravado e os registros depois de `300 Pam White 0.00` seriam copiados para o novo arquivo. Reescrever todo o arquivo não é uma boa ideia para atualizar um único registro, mas razoável se um número substancial de registros tiver de ser atualizado.

17.6 Serialização de objetos

Na Seção 17.5, demonstramos como gravar os campos individuais de um objeto `AccountRecord` em um arquivo como texto e ler esses campos de um arquivo e colocar seus valores em um objeto `AccountRecord` na memória. Nos exemplos, `AccountRecord` foi utilizado para agregar as informações a um registro. Quando a saída das variáveis de instância para um `AccountRecord` fosse gerada para um ar-

quivo em disco, certas informações seriam perdidas, como o tipo de cada valor. Por exemplo, se o valor `"3"` for lido a partir de um arquivo, não haverá como informar se ele veio de um `int`, um `String` ou um `double`. Temos apenas dados, não informações de tipo, em um disco. Se o programa que fosse ler esses dados "soubesse" a que tipo de objeto os dados correspondem, os dados então seriam simplesmente lidos e transferidos para objetos desse tipo. Por exemplo, na Seção 17.5.2, sabemos que estamos inserindo um `int` (o número de conta), seguido por duas `Strings` (o nome e sobrenome) e um `double` (o saldo). Também sabemos que esses valores são separados por espaços, com somente um registro em cada linha. Às vezes, nós saberemos exatamente como os dados serão armazenados em um arquivo. Nesses casos, queremos ler ou gravar um objeto inteiro a partir de um arquivo. O Java fornece esse mecanismo, chamado **serialização de objetos**. Um **objeto serializado** é um objeto representado como uma sequência de bytes que inclui os dados do objeto bem como as informações sobre o tipo do objeto e os tipos dos dados armazenados no objeto. Depois que um objeto serializado foi gravado em um arquivo, ele pode ser lido a partir do arquivo e **desserializado**, isto é, as informações dos tipos e bytes que representam o objeto e seus dados podem ser utilizadas para recriar o objeto na memória.

Observação de engenharia de software 17.1

O mecanismo de serialização cria cópias exatas dos objetos. Isso simplificará a maneira de clonar objetos sem a necessidade de sobrescrever o método `Object` `clone`.

Classes `ObjectInputStream` e `ObjectOutputStream`

As classes `ObjectInputStream` e `ObjectOutputStream` que, respectivamente, implementam as interfaces **`ObjectInput`** e **`ObjectOutput`**, permitem que objetos inteiros sejam lidos ou gravados em um fluxo (possivelmente um arquivo). Para utilizar a serialização com arquivos, inicializamos os objetos `ObjectInputStream` e `ObjectOutputStream` com objetos de fluxo que leem de e gravam em arquivos — os objetos das classes `FileInputStream` e `FileOutputStream`, respectivamente. Esse tipo de inicialização de objetos de fluxo com outros objetos de fluxo é chamado, às vezes, de **empacotamento** — o novo objeto de fluxo a ser criado empacota o objeto de fluxo especificado como um argumento do construtor. Por exemplo, para empacotar um `FileInputStream` em um `ObjectInputStream`, passamos o objeto `FileInputStream` para o construtor de `ObjectInputStream`.

Interfaces `ObjectOutput` e `ObjectInput`

A interface `ObjectOutput` contém o método **`writeObject`**, que recebe um `Object` como um argumento e grava suas informações em um `OutputStream`. Uma classe que implementa a interface `ObjectOuput` (como `ObjectOutputStream`) declara esse método e assegura que o objeto sendo gerado implementa a interface `Serializable` (discutida mais adiante). Correspondentemente, a interface `ObjectInput` contém o método **`readObject`**, que lê e retorna uma referência a um `Object` a partir de um `InputStream`. Depois que um objeto foi lido, podemos fazer uma coerção da sua referência para o tipo real do objeto. Como você verá no Capítulo 27, aplicativos que se comunicam via uma rede, como a Internet, também podem transmitir objetos inteiros pela rede.

17.6.1 Criando um arquivo de acesso sequencial com a serialização de objeto

Esta seção e a Seção 17.6.2 criam e manipulam arquivos de acesso sequencial utilizando a serialização de objeto. A serialização de objetos que mostramos aqui é realizada com fluxos baseados em bytes, assim arquivos sequenciais criados e manipulados serão arquivos binários. Lembre-se de que, em geral, arquivos binários não podem ser visualizados nos editores de textos padrão. Por essa razão, escrevemos um aplicativo separado que sabe ler e exibir objetos serializados. Iniciamos criando e gravando objetos serializados em um arquivo de acesso sequencial. O exemplo é semelhante àquele na Seção 17.5, portanto focalizamos apenas os novos recursos.

Definindo a classe `AccountRecordSerializable`

Vamos começar modificando nossa classe `AccountRecord` para que objetos dessa classe possam ser serializados. A classe `AccountRecordSerializable` (Figura 17.16) implementa a interface **`Serializable`** (linha 7), que permite aos objetos de `AccountRecordSerializable` ser serializados e desserializados com `ObjectOutputStreams` e `ObjectInputStreams`, respectivamente. A interface `Serializable` é uma **interface de tags**. Essa interface não contém nenhum método. Uma classe que implementa `Serializable` é marcada com tags como um objeto `Serializable`. Isso é importante, porque um `ObjectOutputStream` não enviará para a saída um objeto a menos que ele *seja um* objeto `Serializable`, que é o caso para qualquer objeto de uma classe que implementa `Serializable`.

```
1  // Figura 17.16: AccountRecordSerializable.java
2  // Classe AccountRecordSerializable para objetos serializáveis.
3  package com.deitel.ch17; // empacotada para reutilização
4
5  import java.io.Serializable;
6
7  public class AccountRecordSerializable implements Serializable
8  {
9     private int account;
```

```
10       private String firstName;
11       private String lastName;
12       private double balance;
13
14       // construtor sem argumentos chama outro construtor com valores padrão
15       public AccountRecordSerializable()
16       {
17          this( 0, "", "", 0.0 );
18       } // fim do construtor sem argumentos AccountRecordSerializable
19
20       // construtor com quatro argumentos inicializa um registro
21       public AccountRecordSerializable(
22          int acct, String first, String last, double bal )
23       {
24          setAccount( acct );
25          setFirstName( first );
26          setLastName( last );
27          setBalance( bal );
28       } // fim do construtor de AccountRecordSerializable com quatro argumentos
29
30       // configura o número de conta
31       public void setAccount( int acct )
32       {
33          account = acct;
34       } // fim do método setAccount
35
36       // obtém o número de conta
37       public int getAccount()
38       {
39          return account;
40       } // fim do método getAccount
41
42       // configura o nome
43       public void setFirstName( String first )
44       {
45          firstName = first;
46       } // fim do método setFirstName
47
48       // obtém o primeiro nome
49       public String getFirstName()
50       {
51          return firstName;
52       } // fim do método getFirstName
53
54       // configura o sobrenome
55       public void setLastName( String last )
56       {
57          lastName = last;
58       } // fim do método setLastName
59
60       // obtém o último nome
61       public String getLastName()
62       {
63          return lastName;
64       } // fim do método getLastName
65
66       // configura o saldo
67       public void setBalance( double bal )
68       {
69          balance = bal;
70       } // fim do método setBalance
71
72       // obtém o saldo
73       public double getBalance()
74       {
75          return balance;
76       } // fim do método getBalance
77    } // fim da classe AccountRecordSerializable
```

Figura 17.16 | Classe `AccountRecordSerializable` para objetos serializáveis.

Em uma classe que implementa `Serializable`, o programador deve assegurar que cada variável de instância seja um tipo `Serializable`. Do contrário, deve ser declarado **`transient`** para indicar que não é `Serializable` e deve ser ignorado durante o processo de serialização. Por padrão, todas as variáveis de tipo primitivo são serializáveis. Para variáveis de tipo por referência, você deve verificar a documentação da classe (e possivelmente suas superclasses) para assegurar que o tipo é `Serializable`. Por exemplo, `String`s são `Serializable`. Por padrão, os array são serializáveis; mas, em um array de tipo por referência, os objetos referenciados podem ou não ser serializáveis. A classe `AccountRecordSerializable` contém membros de dados `private` `account`, `firstName`, `lastName` e `balance` — os quais são `Serializable`. Essa classe também fornece os métodos `public` *get* e *set* para acessar os campos `private`.

Gravando objetos serializados em um arquivo de acesso sequencial

Agora vamos discutir o código que cria o arquivo de acesso sequencial (figuras 17.17–17.18). Aqui, vamos nos concentrar apenas nos novos conceitos. Como afirmado na Seção 17.3, um programa pode abrir um arquivo criando um objeto da classe de fluxo `FileInputStream` ou `FileOutputStream`. Nesse exemplo, o arquivo deve ser aberto para saída, assim o programa cria um `FileOutputStream` (linha 21 da Figura 17.17). O argumento de `String` passado para o construtor de `FileOutputStream` representa o nome e caminho do arquivo a ser aberto. Arquivos existentes que são abertos para saída dessa maneira são truncados. Note que é usada a extensão de arquivo `.ser` — utilizamos essa extensão de arquivo para arquivos binários que contêm objetos serializados.

```
1  // Figura 17.17: CreateSequentialFile.java
2  // Gravando objetos sequencialmente em um arquivo com a classe ObjectOutputStream.
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectOutputStream;
6  import java.util.NoSuchElementException;
7  import java.util.Scanner;
8
9  import com.deitel.ch17.AccountRecordSerializable;
10
11 public class CreateSequentialFile
12 {
13    private ObjectOutputStream output; // gera saída dos dados no arquivo
14
15    // permite que o usuário especifique o nome do arquivo
16    public void openFile()
17    {
18       try // abre o arquivo
19       {
20          output = new ObjectOutputStream(
21             new FileOutputStream( "clients.ser" ) );
22       } // fim do try
23       catch ( IOException ioException )
24       {
25          System.err.println( "Error opening file." );
26       } // fim do catch
27    } // fim do método openFile
28
29    // adiciona registros ao arquivo
30    public void addRecords()
31    {
32       AccountRecordSerializable record; // objeto a ser gravado no arquivo
33       int accountNumber = 0; // número da conta para o objeto de registro
34       String firstName; // primeiro nome para o objeto de registro
35       String lastName; // sobrenome para o objeto de registro
36       double balance; // saldo para o objeto de registro
37
38       Scanner input = new Scanner( System.in );
39
40       System.out.printf( "%s\n%s\n%s\n%s\n\n",
41          "To terminate input, type the end-of-file indicator ",
42          "when you are prompted to enter input.",
43          "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
44          "On Windows type <ctrl> z then press Enter" );
45
46       System.out.printf( "%s\n%s",
47          "Enter account number (> 0), first name, last name and balance.",
48          "? " );
49
50       while ( input.hasNext() ) // faz um loop até o indicador de fim de arquivo
51       {
52          try // gera saída dos valores para o arquivo
```

```
53            {
54               accountNumber = input.nextInt(); // lê o número de conta
55               firstName = input.next(); // lê o primeiro nome
56               lastName = input.next(); // lê o sobrenome
57               balance = input.nextDouble(); // lê o saldo
58
59               if ( accountNumber > 0 )
60               {
61                  // cria um novo registro
62                  record = new AccountRecordSerializable( accountNumber,
63                     firstName, lastName, balance );
64                  output.writeObject( record ); // gera a saída de registro
65               } // fim do if
66               else
67               {
68                  System.out.println(
69                     "Account number must be greater than 0." );
70               } // fim de else
71            } // fim do try
72            catch ( IOException ioException )
73            {
74               System.err.println( "Error writing to file." );
75               return;
76            } // fim do catch
77            catch ( NoSuchElementException elementException )
78            {
79               System.err.println( "Invalid input. Please try again." );
80               input.nextLine(); // descarta a entrada para que o usuário possa tentar novamente
81            } // fim do catch
82
83            System.out.printf( "%s %s\n%s", "Enter account number (>0),",
84               "first name, last name and balance.", "? " );
85         } // fim do while
86      } // fim do método addRecords
87
88      // fecha o arquivo e termina o aplicativo
89      public void closeFile()
90      {
91         try // fecha o arquivo
92         {
93            if ( output != null )
94               output.close();
95         } // fim do try
96         catch ( IOException ioException )
97         {
98            System.err.println( "Error closing file." );
99            System.exit( 1 );
100        } // fim do catch
101     } // fim do método closeFile
102  } // fim da classe CreateSequentialFile
```

Figura 17.17 | Arquivo sequencial criado com `ObjectOutputStream`.

```
1  // Figura 17.18: CreateSequentialFileTest.java
2  // Testando a classe CreateSequentialFile.
3
4  public class CreateSequentialFileTest
5  {
6     public static void main( String[] args )
7     {
8        CreateSequentialFile application = new CreateSequentialFile();
9
10       application.openFile();
11       application.addRecords();
12       application.closeFile();
13    } // fim de main
14 } // fim da classe CreateSequentialFileTest
```

```
To terminate input, type the end-of-file indicator
when you are prompted to enter input.
On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
On Windows type <ctrl> z then press Enter

Enter account number (> 0), first name, last name and balance.
? 100 Bob Jones 24.98
Enter account number (> 0), first name, last name and balance.
? 200 Steve Doe -345.67
Enter account number (> 0), first name, last name and balance.
? 300 Pam White 0.00
Enter account number (> 0), first name, last name and balance.
? 400 Sam Stone -42.16
Enter account number (> 0), first name, last name and balance.
? 500 Sue Rich 224.62
Enter account number (> 0), first name, last name and balance.
? ^Z
```

Figura 17.18 | Testando a classe `CreateSequentialFile`.

Erro comum de programação 17.2

É um erro de lógica abrir um arquivo existente para saída quando, de fato, você deseja preservar o arquivo. A classe `FileOutputStream` fornece um construtor sobrecarregado que permite abrir um arquivo e acrescentar dados ao fim do arquivo. Isso preservará o conteúdo do arquivo.

A classe `FileOutputStream` fornece métodos para gravar array de `byte` e `byte` individuais em um arquivo, mas queremos gravar objetos em um arquivo. Por essa razão, empacotamos um `FileOutputStream` em um `ObjectOutputStream` passando o novo objeto `FileOutputStream` para o construtor de `ObjectOutputStream` (linhas 20–21). O objeto `ObjectOutputStream` utiliza o objeto `FileOutputStream` para gravar objetos no arquivo. As linhas 20–21 podem lançar uma **`IOException`** se um problema ocorrer ao abrir o arquivo (por exemplo, quando um arquivo é aberto para gravação em uma unidade com espaço insuficiente ou quando um arquivo de leitura é aberto para gravação). Se isso ocorrer, o programa exibirá uma mensagem de erro (linhas 23–26). Se nenhuma exceção ocorrer, o arquivo é aberto e a variável `output` pode ser utilizada para gravar objetos nela.

Esse programa supõe que os dados foram inseridos corretamente e na ordem de número de registro adequada. O método `addRecords` (linhas 30–86) realiza a operação de gravação. As linhas 62–63 criam um objeto `AccountRecordSerializable` a partir dos dados inseridos pelo usuário. A linha 64 chama o método `ObjectOutputStream` `writeObject` para gravar o objeto `record` no arquivo de saída. Observe que somente uma instrução é requerida para gravar todo o objeto.

O método `closeFile` (linhas 89–101) chama o método `ObjectOutputStream` **`close`** em `output` para fechar tanto o `ObjectOutputStream` como seu `FileOutputStream` subjacente (linha 94). Observe que a chamada ao método `close` está contida em um bloco `try`. O método `close` lança uma `IOException` se o arquivo não puder ser fechado adequadamente. Nesse caso, é importante notificar o usuário de que as informações no arquivo talvez estejam corrompidas. Ao utilizar fluxos empacotados, fechar o fluxo mais externo também fecha o arquivo subjacente.

Na execução de exemplo do programa na Figura 17.18, inserimos informações para cinco contas — as mesmas informações mostradas na Figura 17.9. O programa, na verdade, não mostra como os registros dos dados aparecem no arquivo. Lembre-se de que agora estamos utilizando arquivos binários, que não são humanamente legíveis. Para verificar se o arquivo foi criado com sucesso, a próxima seção apresenta um programa para ler o conteúdo do arquivo.

17.6.2 Lendo e desserializando dados a partir de um arquivo de acesso sequencial

A seção anterior mostrou como criar um arquivo de acesso sequencial utilizando a serialização de objetos. Nesta seção, discutimos como ler dados serializados sequencialmente a partir de um arquivo.

O programa nas figuras 17.19–17.20 lê registros de um arquivo criado pelo programa na Seção 17.6.1 e exibe o conteúdo. O programa abre o arquivo para entrada criando um objeto `FileInputStream` (linha 21). O nome do arquivo a ser aberto é especificado como um argumento para o construtor `FileInputStream`. Na Figura 17.17, os objetos no arquivo foram gravados com um objeto `ObjectOutputStream`. Os dados devem ser lidos do arquivo no mesmo formato em que foram gravados. Portanto, utilizamos um `ObjectInputStream` empacotado em um `FileInputStream` nesse programa (linhas 20–21). Se nenhuma exceção ocorrer ao abrir o arquivo, a variável `input` poderá ser utilizada para ler objetos a partir do arquivo.

```
1  // Figura 17.19: ReadSequentialFile.java
2  // Lendo um arquivo de objetos sequencialmente com ObjectInputStream
3  // e exibe cada registro.
4  import java.io.EOFException;
5  import java.io.FileInputStream;
6  import java.io.IOException;
7  import java.io.ObjectInputStream;
8
9  import com.deitel.ch17.AccountRecordSerializable;
10
11 public class ReadSequentialFile
12 {
13    private ObjectInputStream input;
14
15    // permite que o usuário selecione o arquivo a abrir
16    public void openFile()
17    {
18       try // abre o arquivo
19       {
20          input = new ObjectInputStream(
21             new FileInputStream( "clients.ser" ) );
22       } // fim do try
23       catch ( IOException ioException )
24       {
25          System.err.println( "Error opening file." );
26       } // fim do catch
27    } // fim do método openFile
28
29    // lê o registro no arquivo
30    public void readRecords()
31    {
32       AccountRecordSerializable record;
33       System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
34          "First Name", "Last Name", "Balance" );
35
36       try // insere os valores do arquivo
37       {
38          while ( true )
39          {
40             record = ( AccountRecordSerializable ) input.readObject();
41
42             // exibe o conteúdo de registro
43             System.out.printf( "%-10d%-12s%-12s%10.2f\n",
44                record.getAccount(), record.getFirstName(),
45                record.getLastName(), record.getBalance() );
46          } // fim do while
47       } // fim do try
48       catch ( EOFException endOfFileException )
49       {
50          return; // fim do arquivo foi alcançado
51       } // fim do catch
52       catch ( ClassNotFoundException classNotFoundException )
53       {
54          System.err.println( "Unable to create object." );
55       } // fim do catch
56       catch ( IOException ioException )
57       {
58          System.err.println( "Error during read from file." );
59       } // fim do catch
60    } // fim do método readRecords
61
62    // fecha o arquivo e termina o aplicativo
63    public void closeFile()
64    {
65       try // fecha o arquivo e encerra
66       {
67          if ( input != null )
68             input.close();
```

```
69          } // fim do try
70          catch ( IOException ioException )
71          {
72             System.err.println( "Error closing file." );
73             System.exit( 1 );
74          } // fim do catch
75       } // fim do método closeFile
76    } // fim da classe ReadSequentialFile
```

Figura 17.19 | Lendo um arquivo de objetos sequencialmente com `ObjectInputStream` e exibindo cada registro.

O programa lê registros do arquivo no método `readRecords` (linhas 30–60). A linha 40 chama o método `ObjectInputStream` `readObject` para ler um `Object` a partir do arquivo. Para utilizar os métodos específicos a `AccountRecordSerializable`, fazemos um downcast do `Object` retornado para o tipo `AccountRecordSerializable`. O método `readObject` lança uma **`EOFException`** (processada nas linhas 48–51) se ocorrer uma tentativa de leitura depois do fim do arquivo. O método `readObject` lança uma `ClassNotFoundException` se a classe para o objeto sendo lido não puder ser localizada. Isso pode ocorrer se o arquivo acessado em um computador não tiver essa classe. A Figura 17.20 contém o método `main` (linhas 6–13), que abre o arquivo, chama o método `readRecords` e fecha o arquivo.

```
1  // Figura 17.20: ReadSequentialFileTest.java
2  // Testando a classe ReadSequentialFile.
3
4  public class ReadSequentialFileTest
5  {
6     public static void main( String[] args )
7     {
8        ReadSequentialFile application = new ReadSequentialFile();
9
10       application.openFile();
11       application.readRecords();
12       application.closeFile();
13    } // fim de main
14 } // fim da classe ReadSequentialFileTest
```

```
Account   First Name  Last Name      Balance
100       Bob         Jones            24.98
200       Steve       Doe            -345.67
300       Pam         White            0.00
400       Sam         Stone          -42.16
500       Sue         Rich           224.62
```

Figura 17.20 | Testando a classe `ReadSequentialFile`.

17.7 Classes `java.io` adicionais

Essa seção apresenta uma visão geral das interfaces e classes adicionais (do pacote `java.io`) para fluxos de entrada e saída baseados em bytes e fluxos de entrada e saída baseados em caracteres.

17.7.1 Interfaces e classes para entrada e saída baseadas em bytes

`InputStream` e **`OutputStream`** são classes `abstract` que declaram os métodos para realizar entrada e saída baseadas em bytes, respectivamente. Utilizamos várias subclasses concretas `FileInputStream` `InputStream` e `OutputStream` para manipular arquivos neste capítulo.

Fluxos de pipe

Pipes são canais de comunicação sincronizados entre threads. Discutiremos threads no Capítulo 26, "Multithreading". O Java fornece **`PipedOutputStream`** (uma subclasse de `OutputStream`) e **`PipedInputStream`** (uma subclasse de `InputStream`) para estabelecer pipes entre duas threads em um programa. Um thread envia dados a outro gravando em um `PipedOutputStream`. A thread-alvo lê informações do pipe via um `PipedInputStream`.

Fluxos de filtro

Um **FilterInputStream** filtra um InputStream e um FilterOutputStream filtra um OutputStream. A **filtragem** significa simplesmente que o fluxo do filtro fornece funcionalidades adicionais, como agregar bytes de dados a unidades de tipos primitivos significativas. FilterInputStream e FilterOutputStream são geralmente estendidas, assim algumas das suas capacidades de filtragem são fornecidas pelas suas subclasses.

Um **PrintStream** (uma subclasse de FilterOutputStream) realiza a saída de texto para o fluxo especificado. Na verdade, já utilizamos a saída PrintStream por todo o texto até esse ponto — System.out e System.err são objetos PrintStream.

Fluxos de dados

Ler dados como bytes brutos é rápido, mas grosseiro. Normalmente, os programas leem os dados como agregado de bytes que formam ints, floats, doubles e assim por diante. Programas Java podem utilizar várias classes para inserir e gerar saída de dados na forma agregada.

A interface DataInput descreve os métodos para ler tipos primitivos a partir de um fluxo de entrada. As classes **DataInputStream** e RandomAccessFile implementam essa interface para ler conjuntos de bytes e visualizá-los como valores de tipos primitivos. A interface DataInput inclui métodos como readBoolean, readByte, readChar, readDouble, readFloat, readFully (para arrays byte), readInt, readLong, readShort, readUnsignedByte, readUnsignedShort, readUTF (para ler caracteres Unicode codificados pelo Java — discutimos a codificação UTF no Apêndice L) e skipBytes.

A interface DataOutput descreve o conjunto de métodos para gravar tipos primitivos em um fluxo de saída. As classes **DataOutputStream** (uma subclasse de FilterOutputStream) e RandomAccessFile implementam essa interface para gravar valores de tipos primitivos como bytes. A interface DataOutput inclui versões sobrecarregadas do método write (para um byte ou para um array byte) e os métodos writeBoolean, writeByte, writeBytes, writeChar, writeChars (para Strings Unicode), writeDouble, writeFloat, writeInt, writeLong, writeShort e writeUTF (para imprimir texto modificado para o Unicode).

Fluxos armazenados em buffer

Armazenamento em buffer (buffering) é uma técnica de aprimoramento do desempenho de E/S. Com um **BufferedOutputStream** (uma subclasse da classe FilterOutputStream), cada instrução de saída não necessariamente resulta em uma transferência física real de dados para o dispositivo de saída (uma operação lenta se comparada com as velocidades do processador e da memória principal). Em vez disso, cada operação de saída é dirigida para uma região na memória chamada **buffer** que é suficientemente grande para armazenar os dados de muitas operações de saída. Então a transferência real para o dispositivo de saída é realizada em uma grande **operação física de saída** toda vez que o buffer se enche. As operações de saída dirigidas para o buffer de saída na memória são frequentemente chamadas **operações lógicas de saída**. Com um BufferedOutputStream, um buffer parcialmente preenchido pode ser forçado a enviar para o dispositivo a qualquer momento invocando o método **flush** do objeto de fluxo.

Utilizar o armazenamento em buffer pode aumentar significativamente o desempenho de um aplicativo. Operações típicas de E/S são extremamente lentas se comparadas à velocidade de acesso aos dados na memória do computador. O armazenamento em buffer reduz o número de operações de E/S combinando primeiro saídas menores na memória. O número de operações físicas reais de E/S é pequeno se comparado ao número de solicitações de E/S emitidas pelo programa. Portanto, o programa que utiliza armazenamento em buffer é mais eficiente.

Dica de desempenho 17.1

E/S armazenada em buffer produz melhorias significativas de desempenho em relação a E/S não armazenada em buffer.

Com um **BufferedInputStream** (uma subclasse de classe FilterInputStream) muitos fragmentos ou trechos "lógicos" de dados de um arquivo são lidos como uma grande **operação física de entrada** em um buffer de memória. À medida que o programa solicita novos fragmentos de dados, eles são selecionados do buffer. (Esse procedimento é às vezes chamado de **operação lógica de entrada**.) Quando o buffer está vazio, a próxima operação física real de entrada do dispositivo de entrada é realizada para ler (read) o próximo grupo de trechos "lógicos" de dados. Portanto, o número de operações físicas reais de entrada é pequeno comparado ao número de solicitações de leitura emitido pelo programa.

Fluxos de array byte baseados em memória

O fluxo de E/S do Java inclui capacidades para entrada de arrays de byte na memória e saída de arrays de byte na memória. Um ByteArrayInputStream (uma subclasse de InputStream) lê de um array byte na memória. Um ByteArrayOutputStream (uma subclasse de OutputStream) gera saída a um array byte na memória. Um dos usos de E/S de array de byte é a validação de dados. Um programa pode inserir uma linha inteira por vez do fluxo de entrada em um array de byte. Então uma rotina de validação pode escrutinar o conteúdo do array de byte e corrigir os dados se necessário. Por fim, o programa pode prosseguir para inserir do array de byte, "sabendo" que os dados de entrada estão no formato adequado. Dar saída para um array de byte é uma boa maneira de tirar proveito das poderosas capacidades de formatação de fluxos de saída do Java. Por exemplo, os dados podem ser armazenados em um array de byte, utilizando

a mesma formatação que será exibida em um momento posterior; podemos, então, gerar a saída do array de `byte` em um arquivo para preservar a formatação.

Sequenciando entrada a partir de múltiplos fluxos

Um `SequenceInputStream` (uma subclasse de `InputStream`) concatena logicamente vários `InputStreams` — o programa vê o grupo como um `InputStream` contínuo. Quando o programa alcança o fim de um fluxo de entrada, esse fluxo se fecha e o próximo fluxo na sequência se abre.

17.7.2 As interfaces e classes para entrada e saída baseada em caracteres

Além dos fluxos baseados em bytes, o Java fornece as classes `abstract` **`Reader`** e **`Writer`**, que são fluxos baseados em caracteres de dois bytes Unicode. A maioria dos fluxos baseados em bytes tem classes `Reader` ou `Writer` concretas correspondentes baseadas em caracteres.

Leitores e gravadores para buffer baseados em caracteres

As classes **`BufferedReader`** (uma subclasse da classe `abstract Reader`) e **`BufferedWriter`** (uma subclasse da classe `abstract Writer`) permitem armazenamento em buffer para fluxos baseados em caractere. Lembre-se de que fluxos baseados em caracteres utilizam caracteres Unicode — esses fluxos podem processar dados em qualquer idioma que o conjunto de caracteres Unicode representa.

Leitores e gravadores de array `char` baseados na memória

As classes **`CharArrayReader`** e **`CharArrayWriter`** leem e gravam, respectivamente, um fluxo de caracteres em um array de `char`. Um **`LineNumberReader`** (uma subclasse de `Buffered-Reader`) é um fluxo de caracteres armazenado em buffer que monitora o número de linhas lidas — novas linhas, combinações de quebra de linha e e retorno de carro incrementam a contagem de linhas. Monitorar os números da linha pode ser útil se o programa precisar informar o leitor sobre um erro em uma linha específica.

Leitores e gravadores de arquivos baseados em caracteres, pipes e strings

Um `InputStream` pode ser convertido para um `Reader` via classe **`InputStreamReader`**. De maneira semelhante, um `OuputStream` pode ser convertido para **`OutputStreamWriter`**. A classe `FileReader` (uma subclasse de `InputStreamReader`) e a classe `FileWriter` (uma subclasse de `OutputStreamWriter`) leem e gravam caracteres em um arquivo, respectivamente. As classes **`PipedReader`** e **`Piped-Writer`** implementam fluxos de caracteres baseados em pipe para transferir dados entre threads. A classe **`StringReader`** e **`StringWriter`** leem e gravam caracteres em `Strings`, respectivamente. Uma `PrintWriter` grava caracteres em um fluxo.

17.8 Abrindo arquivos com `JFileChooser`

A classe **`JFileChooser`** exibe uma caixa de diálogo (conhecida como caixa de diálogo `JFileChooser`) que permite ao usuário selecionar facilmente arquivos ou diretórios. Para demonstrar o diálogo de `JFileChooser`, aprimoramos o exemplo na Seção 17.4, como mostrado nas Figuras 17.21–17.22. O exemplo agora contém uma interface gráfica com o usuário, mas continua a exibir os mesmos dados como anteriormente. O construtor chama o método `analyzePath` na linha 34. Esse método chama então o método `getFile` na linha 68 para recuperar o objeto `File`.

```
1  // Figura 17.21: FileDemonstration.java
2  Demonstrando JFileChooser.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.File;
7  import javax.swing.JFileChooser;
8  import javax.swing.JFrame;
9  import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextArea;
12 import javax.swing.JTextField;
13
14 public class FileDemonstration extends JFrame
15 {
16    private JTextArea outputArea; // utilizado para saída
17    private JScrollPane scrollPane; // utilizado para fornecer rolagem para saída
18
19    // configura a GUI
20    public FileDemonstration()
21    {
22       super( "Testing class File" );
```

```
23
24          outputArea = new JTextArea();
25
26          // adiciona outputArea a scrollPane
27          scrollPane = new JScrollPane( outputArea );
28
29          add( scrollPane, BorderLayout.CENTER ); // adiciona scrollPane a GUI
30
31          setSize( 400, 400 ); // configura o tamanho da GUI
32          setVisible( true ); // exibe a GUI
33
34          analyzePath(); // cria e analisa o objeto File
35       } // fim do construtor de FileDemonstration
36
37       // permite que o usuário especifique o nome de arquivo ou diretório
38       private File getFileOrDirectory()
39       {
40          // exibe o diálogo de arquivo para que o usuário possa escolher o arquivo a abrir
41          JFileChooser fileChooser = new JFileChooser();
42          fileChooser.setFileSelectionMode(
43             JFileChooser.FILES_AND_DIRECTORIES );
44
45          int result = fileChooser.showOpenDialog( this );
46
47          // se o usuário clicou no botão Cancel no diálogo, retorna
48          if (result == JFileChooser.CANCEL_OPTION )
49             System.exit( 1 );
50
51          File fileName = fileChooser.getSelectedFile(); // obtém o arquivo
52
53          // exibe erro se inválido
54          if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
55          {
56             JOptionPane.showMessageDialog( this, "Invalid Name",
57                "Invalid Name", JOptionPane.ERROR_MESSAGE );
58             System.exit( 1 );
59          } // fim do if
60
61          return fileName;
62       } // fim do método getFile
63
64       // exibe informações sobre o arquivo ou diretório que o usuário especifica
65       public void analyzePath()
66       {
67          // cria o objeto File com base na entrada de usuário
68          File name = getFileOrDirectory();
69
70          if ( name.exists() ) // se o nome existir, gera saída das informações sobre ele
71          {
72             // exibe informações sobre o arquivo (ou diretório)
73             outputArea.setText( String.format(
74                "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
75                name.getName(), " exists",
76                ( name.isFile() ? "is a file" : "is not a file" ),
77                ( name.isDirectory() ? "is a directory" :
78                   "is not a directory" ),
79                ( name.isAbsolute() ? "is absolute path" :
80                   "is not absolute path" ), "Last modified: ",
81                name.lastModified(), "Length: ", name.length(),
82                "Path: ", name.getPath(), "Absolute path: ",
83                name.getAbsolutePath(), "Parent: ", name.getParent() ) );
84
85             if ( name.isDirectory() ) // listagem de diretório de saída
86             {
87                String[] directory = name.list();
88                outputArea.append( "\n\nDirectory contents:\n" );
89
90                for ( String directoryName : directory )
91                   outputArea.append( directoryName + "\n" );
```

```
92            } // fim de else
93         } // fim do if mais externo
94         else // não se trata de um arquivo nem de um diretório, então imprime uma mensagem de erro
95         {
96            JOptionPane.showMessageDialog( this, name +
97               " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE );
98         } // fim do else
99      } // fim do método analyzePath
100  } // fim da classe FileDemonstration
```

Figura 17.21 | Demonstrando `JFileChooser`.

```
1  // Figura 17.22: FileDemonstrationTest.java
2  // Testando a classe FileDemonstration.
3  import javax.swing.JFrame;
4
5  public class FileDemonstrationTest
6  {
7     public static void main( String[] args )
8     {
9        FileDemonstration application = new FileDemonstration();
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11    } // fim de main
12 } // fim da classe FileDemonstrationTest
```

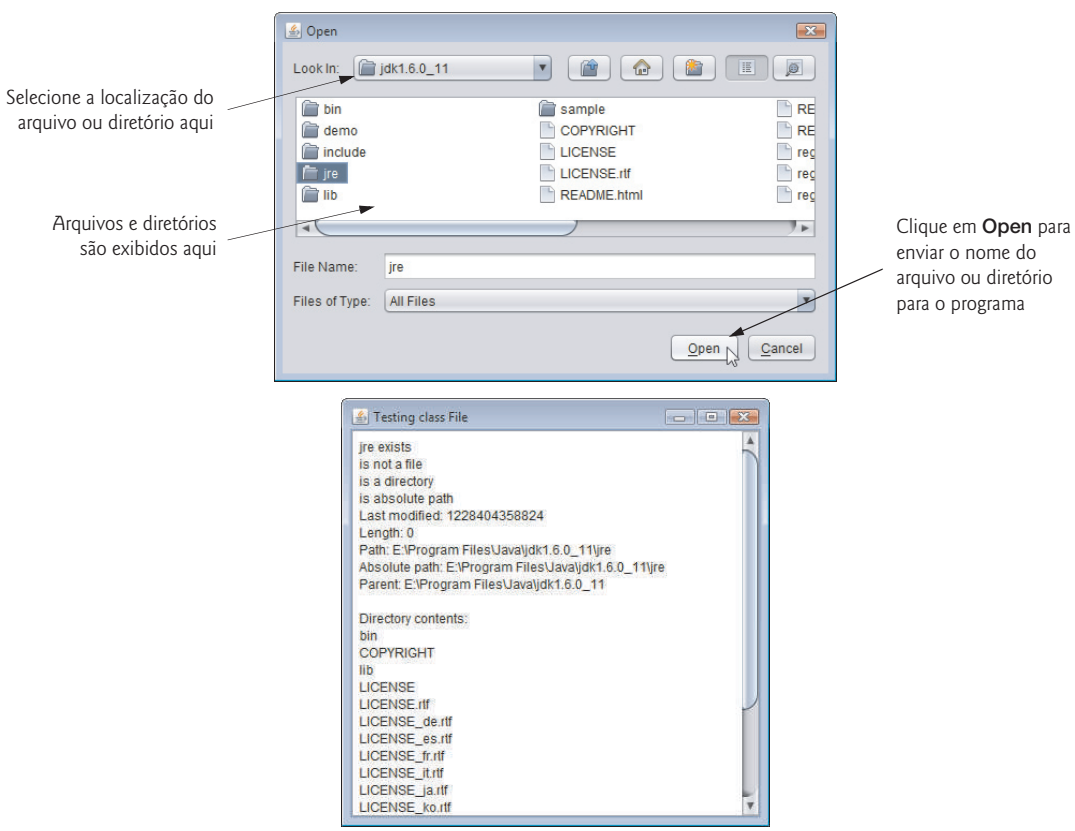


Figura 17.22 | Testando a classe `FileDemonstration`.

O método `getFile` é definido nas linhas 38–62 da Figura 17.21. A linha 41 cria um `JFileChooser` e atribui sua referência a `fileChooser`. As linhas 42–43 chamam o método **`setFileSelectionMode`** para especificar o que o usuário pode selecionar no `fileChooser`. Para esse programa, utilizamos a constante `JFileChooser` `static` **`FILES_AND_DIRECTORIES`** para indicar que arquivos e diretórios podem ser selecionados. Outras constantes `static` incluem **`FILES_ONLY`** (o padrão) e **`DIRECTORIES_ONLY`**.

A linha 45 chama o método de **showOpenDialog** para exibir o diálogo de `JFileChooser` intitulado **Open**. O argumento `this` especifica a janela pai do diálogo de `JFileChooser`, que determina a posição do diálogo na tela. Se `null` for passado, o diálogo será exibido no centro da tela — caso contrário, o diálogo é centralizado na janela do aplicativo (especificado pelo argumento `this`). Um diálogo `JFileChooser` é um diálogo modal que não permite ao usuário interagir com nenhuma outra janela no programa até que o usuário feche o `JFileChooser` clicando no botão **Open** ou **Cancel**. O usuário seleciona a unidade, diretório ou nome de arquivo e, então, clica em **Open**. O método `showOpenDialog` retorna um inteiro especificando em qual botão (**Open** ou **Cancel**) o usuário clicou para fechar o diálogo. A linha 48 testa se o usuário clicou em **Cancel** comparando o resultado com a constante `static` **CANCEL_OPTION**. Se forem iguais, o programa termina. A linha 51 recupera o arquivo que o usuário selecionou chamando o método **getSelectedFile** de `JFileChooser`. Então o programa exibe as informações sobre o arquivo ou diretório selecionado.

17.9 Conclusão

Neste capítulo, você aprendeu a utilizar o processamento de arquivos para manipular dados persistentes. Você aprendeu que os dados são armazenados em computadores como `0`s e `1`s e que combinações desses valores são utilizadas para formar bytes, campos, registros e arquivos. Comparamos fluxos baseados em caracteres e em bytes, e introduzimos várias classes de processamento de arquivo fornecidas pelo pacote `java.io`. Você utilizou a classe `File` para recuperar informações sobre um arquivo ou diretório. Você utilizou o processamento de arquivo de acesso sequencial para manipular registros que são armazenados na ordem pelo campo de chave de registro. Você aprendeu as diferenças entre o processamento de arquivos de texto e a serialização de objetos e utilizou a serialização para armazenar e recuperar objetos inteiros. O capítulo concluiu com uma visão geral de outras classes fornecida no pacote `java.io` um pequeno exemplo da utilização de um diálogo de `JFileChooser` para permitir que os usuários possam selecionar facilmente arquivos em uma GUI. No próximo capítulo, você aprenderá o conceito da recursão — métodos que chamam a eles mesmos. Definir métodos dessa maneira pode resultar em programas mais intuitivos.

Resumo

Seção 17.1 Introdução

- Computadores utilizam arquivos para armazenamento de longo prazo de grandes volumes de dados persistentes, mesmo depois dos programas que criaram os dados terminarem.
- Os computadores armazenam arquivos em dispositivos de armazenamento secundários, como discos rígidos.

Seção 17.2 Hierarquia de dados

- O menor item de dados em um computador pode assumir o valor `0` ou o valor `1` e é chamado bit. Em última instância, um computador processa todos os itens de dados como combinações de zeros e uns.
- O conjunto de caracteres do computador é o conjunto de todos os caracteres utilizados para escrever programas e representar dados.
- Os caracteres em Java são caracteres Unicode compostos de dois bytes, cada um composto de oito bits.
- Um campo é um grupo de caracteres ou bytes que transmitem um significado.
- Itens de dados processados pelos computadores formam uma hierarquia de dados que se torna, com relação à estrutura, maior e mais complexa à medida que progredimos de bits para caracteres, campos e assim por diante.
- Em geral, vários campos compõem um registro (implementado como uma `classe` em Java).
- Um registro é um grupo de campos relacionados. Um arquivo é um grupo de registros relacionados.
- Uma chave de registro identifica um registro como pertencente a uma pessoa ou empresa em particular e é única para cada registro. Chaves de registro facilitam a recuperação de registros específicos a partir de um arquivo.
- Há muitas maneiras de organizar registros em um arquivo. O mais comum é chamado arquivo sequencial, em que registros são armazenados na ordem pelo campo-chave de registro.
- Um grupo de arquivos relacionados costuma ser chamado de banco de dados. Uma coleção de programas projetados para criar e gerenciar bancos de dados é chamada sistema de gerenciamento de bancos de dados (*DataBase Management System* — DBMS).

Seção 17.3 Arquivos e fluxos

- O Java vê cada arquivo como um fluxo sequencial de bytes.
- Cada sistema operacional fornece um mecanismo para determinar o fim de um arquivo, como um marcador de fim de arquivo ou uma contagem dos bytes totais no arquivo.
- Os fluxos baseados em bytes representam dados no formato binário.
- Os fluxos baseados em caracteres representam dados como sequências de caracteres.

- Os arquivos criados utilizando os fluxos baseados em bytes são arquivos binários. Os arquivos criados utilizando os fluxos baseados em caracteres são arquivos de texto. Os arquivos de texto podem ser lidos por editores de textos, enquanto arquivos binários são lidos por um programa que converte os dados em um formato legível por humanos.
- O Java também pode associar fluxos a diferentes dispositivos. Três objetos stream estão associados com dispositivos quando um programa Java começa a executar — `System.in`, `System.out` e `System.err`.
- O pacote `java.io` inclui classes de fluxo, como `FileInputStream` (para entrada baseada em bytes em um arquivo), `FileOutputStream` (para saída baseada em bytes para um arquivo), `FileReader` (para entrada baseada em caracteres em um arquivo) e `FileWriter` (para saída baseada em caracteres para um arquivo). Os arquivos são abertos criando objetos dessas classes de fluxo.

Seção 17.4 Classe `File`

- A classe `File` é utilizada para obter informações sobre arquivos e diretórios.
- A entrada e saída baseada em caracteres pode ser realizada com as classes `Scanner` e `Formatter`.
- A classe `Formatter` permite que a saída de dados formatados na tela ou para um arquivo de uma maneira semelhante a `System.out.printf`.
- Um caminho de arquivo ou diretório especifica sua localização em disco.
- Um caminho absoluto contém todos os diretórios desde o diretório-raiz que levam a um arquivo ou diretório específico. Cada arquivo ou diretório em uma unidade de disco tem o mesmo diretório-raiz em seu caminho.
- Um caminho relativo normalmente inicia do diretório em que o aplicativo começou a execução.
- Um caractere separador é utilizado para separar diretórios e arquivos no caminho.

Seção 17.5 Arquivos de texto de acesso sequencial

- O Java não impõe nenhuma estrutura a um arquivo. Você deve estruturar os arquivos para satisfazer as necessidades do seu aplicativo.
- Para recuperar dados sequencialmente de um arquivo, os programas normalmente começam a ler a partir do início do arquivo e leem todos os dados consecutivamente até que as informações desejadas sejam encontradas.
- Os dados em muitos arquivos sequenciais não podem ser modificados sem o risco de destruir outros dados no arquivo. Portanto, registros em um arquivo de acesso sequencial normalmente não são atualizados no lugar. Em vez disso, o arquivo inteiro é normalmente regravado.

Seção 17.6 Serialização de objeto

- O Java fornece um mecanismo chamado serialização de objetos que permite a objetos inteiros serem gravados ou lidos de um fluxo.
- Um objeto serializado é representado como uma sequência de bytes que inclui os dados do objeto bem como as informações sobre o tipo de objeto e os tipos de dados armazenados no objeto.
- Depois que um objeto serializado foi gravado em um arquivo, ele pode ser lido a partir do arquivo e desserializado para recriar o objeto na memória.
- As classes `ObjectInputStream` e `ObjectOutputStream` permitem que objetos inteiros sejam lidos de ou gravados em um fluxo (possivelmente um arquivo).
- Somente as classes que implementam a interface `Serializable` podem ser serializadas e desserializadas.
- A interface `ObjectOutput` contém o método `writeObject`, que recebe um `Object` como um argumento e grava suas informações em um `OutputStream`. Uma classe que implementa essa interface, como `ObjectOutputStream`, asseguraria que o `Object` é `Serializable`.
- A interface `ObjectInput` contém o método `readObject`, que lê e retorna uma referência a um `Object` a partir de um `InputStream`. Depois que um objeto foi lido, podemos fazer uma coerção da sua referência para o tipo real do objeto.

Seção 17.7 Classes `java.io` adicionais

- `InputStream` e `OutputStream` são classes `abstract` para executar a entrada e saída baseada em bytes.
- As classes `FileInputStream` (uma subclasse de `InputStream`) e `FileOutputStream` (uma subclasse de `OutputStream`) manipulam arquivos.
- Pipes são canais de comunicação sincronizados entre threads. Um thread envia dados a outro gravando em um `PipedOutputStream`. O thread alvo lê informações do pipe via um `PipedInputStream`.
- Um fluxo do filtro fornece funcionalidades adicionais, como agregar bytes de dados a unidades de tipos primitivos significativas. `FilterInputStream` e `FilterOutputStream` são, em geral, classes estendidas, assim algumas das suas capacidades de filtragem são fornecidas pelas suas subclasses concretas.
- Um `PrintStream` executa a saída de texto. `System.out` e `System.err` são objetos `PrintStream`.
- A interface `DataInput` descreve os métodos para ler tipos primitivos a partir de um fluxo de entrada. As classes `DataInputStream` e `RandomAccessFile` implementam essa interface.
- A interface `DataOutput` descreve os métodos para gravar tipos primitivos em um fluxo de saída. As classes `DataOutputStream` e `RandomAccessFile` implementam essa interface.
- Armazenamento em buffer (buffering) é uma técnica de aprimoramento do desempenho de E/S. Operações típicas de E/S são extremamente lentas se comparadas ao acesso dos dados na memória do computador. O armazenamento em buffer reduz o número de operações de E/S combinando saídas menores na memória. O número de operações físicas reais de E/S é muito menor do que o número de solicitações de E/S emitidas pelo programa.

- Com um `BufferedOutputStream` cada operação de saída é direcionada a um buffer suficientemente grande para conter os dados de muitas operações de saída. A transferência real para o dispositivo de saída é realizada em uma grande operação física de saída toda vez que o buffer é preenchido. Um buffer parcialmente preenchido pode ser forçado a enviar para o dispositivo a qualquer momento invocando o método `flush` do objeto de fluxo.
- Com um `BufferedInputStream`, muitos fragmentos ou trechos "lógicos" de dados de um arquivo são lidos como uma grande operação física de entrada em um buffer de memória. À medida que um programa solicita dados, ele é selecionado do buffer. Quando o buffer está vazio, a próxima operação de entrada física real é executada.
- Um `ByteArrayInputStream` lê a partir de um array `byte` na memória. Um `ByteArrayOutputStream` gera saída de um array de `byte`s na memória.
- Um `SequenceInputStream` concatena vários `InputStream`s. Quando o programa alcança o fim de um fluxo de entrada, esse fluxo se fecha e o próximo fluxo na sequência se abre.
- As classes `Reader` e `Writer` `abstract` são fluxos baseados em caracteres Unicode. A maioria dos fluxos baseados em bytes tem classes `Reader` ou `Writer` concretas correspondentes baseadas em caracteres
- As classes `BufferedReader` e `BufferedWriter` permitem o armazenamento em buffer de fluxos baseados em caracteres.
- As classes `CharArrayReader` e `CharArrayWriter` manipulam um array de `char`s na memória.
- Um `LineNumberReader` é um fluxo de caracteres armazenado em buffer que monitora o número de linhas lido.
- As classes `FileReader` e `FileWriter` leem caracteres de e gravam caracteres em um arquivo.
- A classe `PipedReader` e a classe `PipedWriter` implementam fluxo de caracteres baseados em pipe para transferir dados entre threads.
- As classes `StringReader` e `StringWriter` leem e gravam caracteres em `String`s, respectivamente. Uma `PrintWriter` grava caracteres em um fluxo.

Seção 17.8 Abrindo arquivos com `JFileChooser`

- A classe `JFileChooser` é utilizada para exibir um diálogo que permite aos usuários de um programa selecionar facilmente arquivos ou diretórios em uma GUI.

Terminologia

`%n`, formatar o especificador (separador de linha), 562
abrir um arquivo, 554
arquivo, 552
arquivo binário, 554
arquivo de acesso sequencial, 553
arquivo de texto, 554
ASCII (American Standard Code for Information Interchange), conjunto de caracteres, 553
banco de dados, 553
bit (dígito binário), 552
buffer, 577
`BufferedInputStream`, classe, 577
`BufferedOutputStream`, classe, 577
`BufferedReader`, classe, 578
`BufferedWriter`, classe, 578
byte, 553
`byte`, tipo primitivo, 553
caminho absoluto, 555
caminho relativo, 555
campo, 553
`CANCEL_OPTION`, constante de `JFileChooser`, 581
caractere, 553
caractere separador, 558
`CharArrayReader`, classe, 578
`CharArrayWriter`, classe, 578
`close`, método de `Formatter`, 562
`close`, método de `ObjectOutputStream`, 574
conjunto de caracteres, 553
dados persistentes, 552
`DataInputStream`, classe, 577
`DataOutputStream`, classe, 577
desserializar um objeto, 570
dígito binário (bit), 552
dígito decimal, 553
`DIRECTORIES_ONLY`, constante de `JFileChooser`, 580
diretório-raiz, 555
dispositivo de armazenamento secundário, 552
empacotando objetos fluxo, 570
`EOFException`, classe, 576
E/S armazenada em buffer, 577
`exit`, método da classe `System`, 561
`File`, classe, 555
`FileInputStream`, classe, 555
`FileNotFoundException`, classe, 561
`FileOutputStream`, classe, 555
`FileReader`, classe, 555
`FILES_AND_DIRECTORIES`, constante de `JFileChooser`, 580
`FILES_ONLY`, constante de `JFileChooser`, 580
`FileWriter`, classe, 555
`FilterInputStream`, classe, 577
`FilterOutputStream`, classe, 577
filtrar um fluxo, 577
`flush`, método da classe `BufferedOutputStream`, 577
fluxo baseado em bytes, 554
fluxo baseado em caracteres, 554
fluxo de bytes, 554
`format`, método da classe Formatter, 562
`Formatter`, classe, 555
`FormatterClosedException`, classe, 562
`getSelectedFile`, método da classe `JFileChooser`, 581
`IllegalStateException`, classe, 565
informações de caminho, 555
`InputStream`, classe, 576
`InputStreamReader`, classe, 578
interface de tags, 570
`IOException`, classe, 574
`isFile`, método de `File`, 556
`java.io`, pacote, 555
`JFileChooser`, classe, 578
letra, 553
`LineNumberReader`, classe, 578
lote, arquivo, 562
marcador de fim de arquivo, 554
`NoSuchElementException`, classe, 562
`ObjectInput`, interface, 570
`ObjectInputStream`, classe, 555
`ObjectOutput`, interface, 570
`ObjectOutputStream`, classe, 555
objeto desserializado, 570
objeto serializado, 570
operação física de entrada, 577
operação física de saída, 577
operações lógicas de entrada, 577
operações lógicas de saída, 577
`OutputStream`, classe, 576
`OutputStreamWriter`, classe, 578
pipe, 576
`PipedInputStream`, classe, 576
`PipedOutputStream`, classe, 576
`PipedReader`, classe, 578
`PipedWriter`, classe, 578
`PrintStream`, classe, 577
`PrintWriter`, classe, 562
`Reader`, classe, 578
`readObject`, método de `ObjectInput`, 570
redirecionar um fluxo padrão, 555
registro, 553
registro, chave, 553

script de shell, 562
SecurityException, classe, 561
Serializable, interface, 570
serialização de objeto, 570
setErr, método da classe System, 555
setFileSelectionMode, método da classe JFileChooser, 580
setIn, método da classe System, 555
setOut, método de System, 555
showOpenDialog, método da classe JFileChooser, 581
símbolos especiais, 553
sistema de gerenciamento de bancos de dados (*DataBase Management System* — DBMS), 553
StringReader, classe, 578
StringWriter, classe, 578
transient, palavra-chave, 572
truncar o conteúdo de um arquivo, 560
Unicode, conjunto de caracteres, 553
Uniform Resource Identifier (URI), 555
Uniform Resource Locator (URL), 555
writeObject, método da interface ObjectOutput, 570
Writer, classe, 578

Exercícios de autorrevisão

17.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Em última instância, todos os itens de dados processados por um computador são reduzidos a combinações de _____ e _____.

b) O menor item de dados que um computador pode processar é chamado de _____.

c) Um _____ às vezes pode ser visualizado como um grupo de registros relacionados.

d) Dígitos, letras e símbolos especiais são referidos como _____.

e) Um banco de dados é um grupo de _____ relacionados.

f) O objeto _____ normalmente permite a um programa gerar saída de mensagens de erros na tela.

17.2 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Você deve criar explicitamente os objetos de fluxo `System.in`, `System.out` e `System.err`.

b) Ao ler dados de um arquivo utilizando a classe `Scanner`, se você quiser ler dados no arquivo múltiplas vezes, o arquivo deve ser fechado e reaberto para ler a partir do início do arquivo.

c) O método `exists` da classe `File` retorna `true` se o nome especificado como o argumento para o construtor `File` for um arquivo ou diretório no caminho especificado.

d) Arquivos binários são legíveis por seres humanos em um editor de textos.

e) Um caminho absoluto contém todos os diretórios desde o diretório-raiz que levam a um arquivo ou diretório específico.

f) A classe `Formatter` contém o método `printf`, que permite gerar a saída de dados formatados na tela ou para um arquivo.

17.3 Complete as seguintes tarefas, supondo que cada uma se aplica ao mesmo programa:

a) Escreva uma instrução que abre o arquivo `"oldmast.txt"` para entrada — utilize a variável `Scanner inOldMaster`.

b) Escreva uma instrução que abre o arquivo `"trans.txt"` para entrada — utilize a variável `Scanner inTransaction`.

c) Escreva uma instrução que abre arquivo `"newmast.txt"` para saída (e criação) — utilize a variável `formatter` de `outNewMaster`.

d) Escreva as instruções necessárias para ler um registro do arquivo `"oldmast.txt"`. Utilize os dados para criar um objeto da classe `AccountRecord` — use a variável `Scanner inOldMaster`. Suponha que a classe `AccountRecord` é idêntica à classe `AccountRecord` na Figura 17.5.

e) Escreva as instruções necessárias para ler um registro do arquivo `"trans.txt"`. O registro é um objeto da classe `TransactionRecord` — use a variável `Scanner inTransaction`. Suponha que a classe `TransactionRecord` contenha o método `setAccount` (que recebe um `int`) para configurar o número de conta e o método `setAmount` (que recebe um `double`) para configurar o valor monetário da transação.

f) Escreva uma instrução que gera a saída de um registro para o arquivo `"newmast.txt"`. O registro é um objeto do tipo `AccountRecord` — use a variável `Formatter outNewMaster`.

17.4 Complete as seguintes tarefas, supondo que cada uma se aplica ao mesmo programa:

a) Escreva uma instrução que abre o arquivo `"oldmast.ser"` para entrada — use a variável `ObjectInputStream inOldMaster` para empacotar um objeto `FileInputStream`.

b) Escreva uma instrução que abre o arquivo `"trans.ser"` para entrada — use a variável `ObjectInputStream inTransaction` para empacotar um objeto `FileInputStream`.

c) Escreva uma instrução que abre arquivo `"newmast.ser"` para saída (e criação) — utilize variável `Object-OutputStream outNewMaster` para empacotar um `FileOutputStream`.

d) Escreva uma instrução que lê um registro no arquivo `"oldmast.ser"`. O registro é um objeto da classe `AccountRecordSerializable` — use a variável `ObjectInputStream inOldMaster`. Assuma que a classe `AccountRecordSerializable` é a mesma que a classe `Account-Record-Serializable` na Figura 17.16

e) Escreva uma instrução que lê um registro no arquivo `"trans.ser"`. O registro é um objeto da classe `TransactionRecord` — use a variável `ObjectInputStream inTransaction`.

f) Escreva uma instrução que gera saída de um registro do tipo `AccountRecordSerializable` para o arquivo `"newmast.ser"` — use a variável `ObjectOutputStream outNewMaster`.

17.5 Encontre o erro em cada bloco de código e mostre como corrigi-lo.

a) Assuma que `account`, `company` e `amount` são declarados.

```
ObjectOutputStream outputStream;
outputStream.writeInt( account );
outputStream.writeChars( company );
outputStream.writeDouble( amount );
```

b) As seguintes instruções devem ler um registro do arquivo `"payables.txt"`. A variável `inPayable` de `Scanner` deve ser usada para referir-se a esse arquivo.

```
Scanner inPayable = new Scanner( new File( "payables.txt" ) );
PayablesRecord record = ( PayablesRecord ) inPayable.readObject();
```

Respostas dos exercícios de autorrevisão

17.1 a) uns, zeros. b) bit. c) arquivo. d) caracteres. e) arquivos. f) `System.err`.

17.2 a) Falsa. Esses três fluxos são criados para você quando um aplicativo Java inicia a execução.

b) Verdadeira.

c) Verdadeira.

d) Falsa. Arquivos de texto são legíveis por seres humanos em um editor de textos. Arquivos binários podem ser legíveis por seres humanos, mas apenas se os bytes no arquivo representarem caracteres ASCII

e) Verdadeira.

f) Falsa. A classe `Formatter` contém o método `format`, que permite gerar a saída de dados formatados na tela ou para um arquivo.

17.3 a)
```
Scanner inOldMaster = new Scanner( new File ( "oldmast.txt" ) );
```

b)
```
Scanner inTransaction = new Scanner( new File( "trans.txt" ) );
```

c)
```
Formatter outNewMaster = new Formatter( "newmast.txt" );
```

d)
```
AccountRecord account = new AccountRecord();
account.setAccount( inOldMaster.nextInt() );
account.setFirstName( inOldMaster.next() );
account.setLastName( inOldMaster.next() );
account.setBalance( inOldMaster.nextDouble() );
```

e)
```
TransactionRecord transaction = new Transaction();
transaction.setAccount( inTransaction.nextInt() );
transaction.setAmount( inTransaction.nextDouble() );
```

f)
```
outNewMaster.format( "%d %s %s %.2f\n",
account.getAccount(), account.getFirstName(),
account.getLastName(), account.getBalance() );
```

17.4 a)
```
ObjectInputStream inOldMaster = new ObjectInputStream(
      new FileInputStream( "oldmast.ser" ) );
```

b)
```
ObjectInputStream inTransaction = new ObjectInputStream(
   new FileInputStream( "trans.ser" ) );
```

c)
```
ObjectOutputStream outNewMaster = new ObjectOutputStream(
   new FileOutputStream( "newmast.ser" ) );
```

d)
```
accountRecord = ( AccountRecordSerializable ) inOldMaster.readObject();
```

e)
```
transactionRecord = ( TransactionRecord ) inTransaction.readObject();
```

f)
```
outNewMaster.writeObject( newAccountRecord );
```

17.5 a) Erro: O arquivo não foi aberto antes da tentativa de gerar a saída dos dados para o fluxo.

Correção: Abra um arquivo para saída criando um novo objeto de `ObjectOutputStream` que empacota um objeto de `FileOutputStream`.

b) Erro: Esse exemplo utiliza arquivos de texto com um `Scanner`; não há nenhuma serialização de objeto. Como resultado, o método `readObject` não pode ser utilizado para ler esses dados do arquivo. Cada fragmento de dados deve ser lido separadamente e então utilizado para criar um objeto de `PayablesRecord`.

Correção: Utilize os métodos de `inPayable` para ler cada parte do objeto `PayablesRecord`.

Exercícios

17.6 Preencha as lacunas em cada uma das seguintes afirmações:

a) Os computadores armazenam quantidades grandes de dados em dispositivos de armazenamento secundários como _____.

b) O _____ é composto de vários campos.

c) Para facilitar a recuperação de registros específicos de um arquivo, um campo em cada registro é escolhido como uma _____.

d) Os arquivos criados utilizando fluxos baseados em bytes são chamados arquivos_____ , enquanto arquivos criados utilizando fluxos baseados em caracteres são chamados arquivos _____.

e) Os objetos padrão de fluxo são _____, _____ e _____.

17.7 Determine quais das seguintes afirmações são *verdadeiras* e quais são *falsas*. Se *falso*, explique por quê.

a) As impressionantes funções realizadas pelos computadores envolvem essencialmente a manipulação de zeros e uns.

b) As pessoas especificam programas e itens de dados como caracteres. Os computadores então manipulam e processam esses caracteres como grupos de zeros e uns.

c) Os itens de dados representados em computadores assumem uma hierarquia de dados em que itens de dados tornam-se cada vez maiores e mais complexos à medida que progredimos de campos para caracteres, então para bits e assim por diante.

d) Uma chave de registro identifica um registro como pertencente a um campo particular.

e) As empresas armazenam todas suas informações em um único arquivo para facilitar o processamento dessas informações pelo computador. Quando um programa cria um arquivo, o arquivo é retido pelo computador para referência futura.

17.8 (***Correspondência de arquivos***) O Exercício de autorrevisão 17.3 pede que você escreva uma série de instruções únicas. De fato, essas instruções formam o núcleo de um importante tipo de programa processador de arquivo, a saber, um programa de correspondência de arquivo (file-matching program). Em processamento de dados comercial, é comum ter vários arquivos em cada sistema de aplicativo. Em um sistema de contas a receber, por exemplo, há em geral um arquivo-mestre contendo informações detalhadas sobre cada cliente, como seu nome, endereço, número de telefone, saldo, limite de crédito, termos de desconto, arranjos de contrato e possivelmente um histórico condensado de compras recentes e pagamentos de conta.

À medida que as transações ocorrem (isto é, são feitas vendas e pagamentos chegam pelo correio), as informações sobre elas são inseridas em um arquivo. No fim de cada período de negócios (um mês para algumas empresas, uma semana para outras e um dia em alguns casos), o arquivo de transações (chamado `"trans.txt"`) é aplicado ao arquivo-mestre (chamado `"oldmast.txt"`) para atualizar o registro de compra e pagamento de cada conta. Durante uma atualização, o arquivo-mestre é regravado como o arquivo `"newmast.txt"`, que é então utilizado no fim do próximo período de negócios para começar o processo de atualização novamente.

Programas de correspondência de arquivo devem lidar com certos problemas que não surgem em programas de um único arquivo. Por exemplo, nem sempre ocorre uma correspondência. Se um cliente no arquivo-mestre não fez nenhuma compra ou pagamentos à vista no período de negócios atual, nenhum registro para esse cliente aparecerá no arquivo de transações. De maneira semelhante, um cliente que fez alguma compra ou pagamento em dinheiro poderia apenas ter mudado para essa comunidade e a empresa pode não ter tido uma oportunidade de criar um registro-mestre para esse cliente.

Escreva um programa completo de correspondência de arquivos de contas a receber. Utilize o número de conta em cada arquivo como a chave de registro para propósitos de correspondência. Assuma que cada arquivo é um arquivo de texto sequencial com registros armazenados em ordem de número de conta crescente.

a) Defina a classe `TransactionRecord`. Os objetos dessa classe contêm um número de conta e o valor monetário para a transação. Forneça métodos para modificar e recuperar esses valores monetários.

b) Modifique a classe `AccountRecord` na Figura 17.5 para incluir o método `combine`, que recebe um objeto `TransactionRecord` e combina o saldo do objeto `AccountRecord` e o valor da quantidade monetária do objeto `TransactionRecord`.

c) Escreva um programa para criar dados a fim de testar o programa. Utilize os dados da conta de exemplo nas figuras 17.23 e 17.24. Execute o programa para criar os arquivos `trans.txt` e `oldmast.txt` a serem utilizados pelo seu programa de correspondência de arquivos.

Número de conta de arquivo-mestre	Nome	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

Figura 17.23 | Dados de exemplo para o arquivo-mestre.

Número de conta de arquivo de transação	Quantia da transação
100	27.14
300	62.11
400	100.56
900	82.17

Figura 17.24 | Dados de exemplo para o arquivo de transações.

d) Crie a classe `FileMatch` para realizar a funcionalidade de correspondência de arquivos. A classe deve conter métodos que leem `oldmast.txt` e `trans.txt`. Quando uma correspondência ocorre (isto é, registros com o mesmo número de conta aparecem tanto no arquivo-mestre como no arquivo de transações), adicione o valor monetário no registro de transação ao saldo atual no registro-mestre e grave o registro em `"newmast.txt"`. (Suponha que compras são indicadas por valores monetários positivos no arquivo de transações e os pagamentos por valores

monetários negativos). Quando houver um registro-mestre para uma conta particular, mas nenhum correspondente registro de transação, simplesmente grave o registro mestre em `"newmast.txt"`. Se houver um registro de transação, mas nenhum registro-mestre correspondente, imprima a mensagem `"Unmatched transaction record for account number..."` `[Registro de transação não correspondido para o número da conta]` em um arquivo de log (preencha o número da conta a partir do registro de transação). O arquivo de log deve ser um arquivo de texto chamado `"log.txt"`.

17.9 **(*Correspondência de arquivos com múltiplas transações*)** É possível (e, na verdade, comum) ter vários registros de transações com a mesma chave de registro. Essa situação ocorre, por exemplo, quando um cliente faz várias compras e pagamentos à vista durante um período de negócios. Reescreva seu programa de correspondência de arquivo de contas a receber do Exercício 17.8 para oferecer a possibilidade de tratamento de vários registros de transação com a mesma chave de registro. Modifique os dados de teste de `CreateData.java` para incluir registros de transações adicionais na Figura 17.25.

Número de conta	Quantia em dólar
300	83.89
700	80.78
700	1.53

Figura 17.25 | Registros de transações adicionais.

17.10 **(*Correspondência de arquivos com serialização de objetos*)** Recrie sua solução para o Exercício 17.9 utilizando a serialização de objetos. Utilize as instruções do Exercício 17.4 como sua base para esse programa. Talvez você queira criar aplicativos para ler os dados armazenados nos arquivos `.ser` — o código na Seção 17.6.2 pode ser modificado para esse propósito.

17.11 **(*Gerador de palavra de número de telefone*)** Os teclados de telefone padrão contêm os dígitos de zero a nove. Os números 2 a 9 têm três letras associadas a cada número (Figura 17.26). Muitas pessoas acham difícil memorizar números de telefone, então utilizam a correspondência entre dígitos e letras para criar palavras de sete letras que correspondem com seus números de telefone. Por exemplo, uma pessoa cujo número de telefone é 686-2377 talvez utilize a correspondência indicada na Figura 17.26 para desenvolver a palavra de sete letras "NUMBERS". Cada palavra de sete letras corresponde a exatamente um número de telefone de sete dígitos. Um restaurante que deseja aumentar seu negócio de entregas em domicílio ("takeout", em inglês) seguramente poderia fazer isso com o número 825-3688 (isto é, "TAKEOUT)".

Dígito	Letras	Dígito	Letras	Dígito	Letras
2	A B C	5	J K L	8	T U V
3	D E F	6	M N O	9	W X Y
4	G H I	7	P R S		

Figura 17.26 | Dígitos e letras do teclado do telefone.

Cada número de telefone de sete letras corresponde a muitas diferentes palavras de sete letras, mas a maioria dessas palavras representa justaposições irreconhecíveis das letras. É possível, porém, que o proprietário de um salão de cabeleireiro ficasse contente em saber que o número de telefone de seu salão, 424-7288, corresponde a "HAIRCUT" (corte de cabelo, em inglês). Um veterinário com o número de telefone 738-2273 ficaria satisfeito em saber que seu número corresponde à palavra de sete letras "PETCARE" (cuidado de animais de estimação). Um vendedor de automóvel ficaria satisfeito em saber que o número de telefone de sua loja, 639-2277, corresponde a "NEWCARS".

Escreva um programa que, dado um número de sete dígitos, utiliza um objeto `PrintStream` para gravar em um arquivo cada possível combinação de palavras de sete letras correspondente a esse número. Há 2.187 (3^7) dessas combinações. Evite números de telefone com os dígitos 0 e 1.

17.12 **(*Pesquisa entre alunos*)** A Figura 7.8 contém um array de respostas a uma pesquisa que é codificado diretamente no programa. Suponha que queremos processar os resultados dessa pesquisa que são armazenados em um arquivo. Este exercício requer dois programas separados. Primeiro, crie um aplicativo que solicita ao usuário respostas à pesquisa e gera a saída de cada resposta para um arquivo. Utilize um `Formatter` para criar um arquivo chamado `numbers.txt`. Cada inteiro deve ser escrito com o método `format`. Então modifique o programa na Figura 7.8 para ler as respostas à pesquisa a partir de `numbers.txt`. As respostas devem ser lidas do arquivo utilizando um `Scanner`. Utilize o método `nextInt` para gerar saída de um número inteiro em um dado momento a partir do arquivo. O programa deve continuar a ler respostas até alcançar o fim do arquivo. A saída dos resultados deve ser gerada no arquivo de texto `"output.txt"`.

17.13 **(*Adicionando serialização de objetos ao aplicativo de desenho MyShape*)** Modifique o Exercício 14.17 para permitir ao usuário salvar um desenho em um arquivo ou carregar um desenho prévio a partir de um arquivo utilizando serialização de objetos. Adicione botões **Load** (para ler objetos de um arquivo) e **Save** (para gravar objetos em um arquivo). Utilize um `ObjectOutputStream` para gravar dados no arquivo e um `ObjectInputStream` para ler dados do arquivo. Escreva o array de objetos `MyShape` utilizando o método `writeObject` (classe `ObjectOutputStream`) e leia o array utilizando o método `readObject` (`ObjectInputStream`). Observe que o mecanismo de serialização de objeto pode ler ou gravar arrays inteiros — não é necessário manipular cada elemento do array de objetos `MyShape` individualmente. Simplesmente é exigido que todas as formas sejam `Serializable`. Para os dois botões **Load** e **Save**, utilize um `JFileChooser` para permitir que o usuário selecione o arqui-

vo em que as formas serão armazenadas ou do qual elas serão lidas. Quando o usuário executa o programa pela primeira vez, nenhuma forma deve ser exibida na tela. O usuário pode exibir formas abrindo um arquivo anteriormente salvo ou desenhando novas formas. Havendo formas na tela, os usuários podem salvá-las em um arquivo utilizando o botão **Save**.

Fazendo a diferença

17.14 **(*Scanner de phishing*)** Phishing é uma forma de furto de identidade em que, em um e-mail, um remetente, que se faz passar por uma fonte confiável, tenta adquirir informações privadas, como nomes de usuário, senhas, números de cartão de crédito e cadastros de pessoas físicas. Falsos e-mails de conhecidos bancos, empresas de cartão do crédito, sites de leilão, redes sociais e serviços de pagamento on-line podem parecer bastante autênticos. Essas mensagens fraudulentas muitas vezes fornecem links para sites Web falsos nos quais você é solicitado a inserir informações confidenciais.

Visite McAfee® (`www.mcafee.com/us/threat_center/anti_phishing/phishing_top10.html`), Security Extra (`www.securityextra.com/`), `www.snopes.com` e outros sites Web para localizar listas das principais fraudes relacionadas ao phishing. Também verifique o Anti-Phishing Working Group (`www.antiphishing.org/`), e o site Web Cyber Investigations do FBI (`www.fbi.gov/cyberinvest/cyberhome.htm`), nos quais você localizará informações sobre as fraudes mais recentes e como se proteger.

Crie uma lista de 30 palavras, frases e nomes de empresas comumente encontrados nas mensagens de phishing. Atribua um valor em pontos a cada um com base na sua estimativa da probabilidade de a mensagem ser um esquema de phishing (por exemplo, um ponto se for mais ou menos provável, dois pontos se moderadamente provável ou três pontos se altamente provável). Escreva um aplicativo que procura esses termos e frases em um arquivo de texto. Para cada ocorrência de uma frase ou palavra-chave dentro do arquivo de texto, adicione o valor em ponto atribuído aos pontos totais para essa palavra ou frase. Para cada frase ou palavra-chave encontrada, imprima uma linha com a palavra ou frase, o número de ocorrências e o total em pontos. Mostre então o total em pontos da mensagem inteira. Seu programa atribui um total alto em pontos a alguns e-mails de phishing reais que você recebeu? Ele atribui um total alto em pontos a alguns e-mails legítimos que você recebeu?

Devemos aprender a explorar todas as opções e possibilidades que nos confrontam em um mundo complexo e em constante e rápida mudança.
— James William Fulbright

O! thou hast damnable iteration, and art indeed able to corrupt a saint. [Fazes sempre citações execráveis; és capaz de corromper um santo.]
— William Shakespeare

É um tipo pobre de memória que só funciona para trás.
— Lewis Carroll

A vida só pode ser compreendida olhando-se para trás; mas só pode ser vivida olhando-se para frente.
— Soren Kierkegaard

Prossiga — continue andando.
— Thomas Morton

18

Recursão

Objetivos

Neste capítulo, você aprenderá:

- O conceito de recursão.
- Como escrever e utilizar métodos recursivos.
- Como determinar o caso básico e o passo de recursão em um algoritmo recursivo.
- Como chamadas de método recursivo são tratadas pelo sistema.
- As diferenças entre recursão e iteração, e quando é apropriado utilizar cada uma.
- O que são formas geométricas chamadas fractais e como desenhá-las utilizando a recursão.
- O que é reversão recursiva (recursive backtracking) e por que é uma técnica efetiva para a resolução de problemas.

Sumário

18.1 Introdução
18.2 Conceitos de recursão
18.3 Exemplo que utiliza recursão: fatoriais
18.4 Exemplo que utiliza recursão: série de Fibonacci
18.5 Recursão e a pilha de chamadas de método
18.6 Recursão *vs.* Iteração
18.7 Torres de Hanói
18.8 Fractais
18.9 Retorno recursivo
18.10 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

18.1 Introdução

Os programas que discutimos até aqui geralmente são estruturados como métodos que chamam uns aos outros de uma maneira hierárquica. Para alguns problemas, é útil ter uma chamada de método própria. Um método que faz isso é conhecido como **método recursivo**. Um método recursivo pode chamar a si próprio direta ou indiretamente por outro método. A recursão é um tópico importante discutido demoradamente em cursos de ciência da computação de nível superior. Neste capítulo, consideramos a recursão conceitualmente, então apresentamos vários programas que contêm métodos recursivos. A Figura 18.1 resume os exemplos e os exercícios de recursão deste livro.

Capítulo	Exemplos de recursão e exercícios neste livro
18	Método fatorial (figuras 18.3 e 18.4) Método de Fibonacci (Figura 18.5) Torres do Hanói (Figura 18.11) Fractais (figuras 18.18 e 18.19) O que faz esse código? (Exercício 18.7, Exercício 18.12 e Exercício 18.13) Localize o erro no seguinte código (Exercício 18.8) Elevando um inteiro à potência de um inteiro (Exercício 18.9) Visualizando a recursão (Exercício 18.10) Máximo divisor comum (Exercício 18.11) Determine se uma string é um palíndromo (Exercício 18.14) Oito rainhas (Exercício 18.15) Imprima um array (Exercício 18.16) Imprima um array de trás para frente (Exercício 18.17) Valor mínimo em um array (Exercício 18.18) Estrela fractal (Exercício 18.19) Percorrendo um labirinto com a reversão recursiva (Exercício 18.20) Gerando labirintos aleatoriamente (Exercício 18.21) Labirintos de qualquer tamanho (Exercício 18.22) Tempo necessário para calcular um número de Fibonacci (Exercício 18.23)
19	Classificação por intercalação (figuras 19.10 e 19.11) Pesquisa linear (Exercício 19.8) Pesquisa binária (Exercício 19.9) Classificação rápida (Quicksort) (Exercício 19.10)
22	Inserção de árvore binária (Figura 22.17) Percorrendo uma árvore binária na pré-ordem (Figura 22.17) Percorrendo uma árvore binária na ordem (Figura 22.17) Percorrendo uma árvore binária na pós-ordem (Figura 22.17) Impressão de uma lista vinculada de trás para frente (Exercício 22.20) Pesquisa em uma lista vinculada (Exercício 22.21)

Figura 18.1 | Resumo dos exemplos e exercícios de recursão neste texto.

18.2 Conceitos de recursão

Abordagens de solução de problemas de recursão têm um número de elementos em comum. Quando um método recursivo é chamado para resolver um problema, na verdade, ele é capaz de resolver somente o(s) caso(s) mais simples(s), ou **caso(s) básico(s)**. Se o método é chamado com um caso básico, ele retorna um resultado. Se o método for chamado com um problema mais complexo, em geral, ele divide o problema em duas partes conceituais — uma parte que o método sabe como fazer e uma parte que ele não sabe. Para tornar a recursão realizável, a última parte deve assemelhar-se ao problema original, mas ser uma versão ligeiramente mais simples ou menor dele. Como esse novo problema é parecido com o problema original, o método chama uma cópia nova dele próprio para trabalhar no problema menor — isso é referido como **chamada recursiva** e também é denominado **passo de recursão**. O passo de recursão normalmente inclui a instrução `return` uma vez que seu resultado será combinado com a parte do problema que o método sabia como resolver para formar um resultado que será passado de volta para o chamador original. Esse conceito de separar o problema em duas partes menores é uma forma da abordagem de dividir para conquistar introduzida no Capítulo 6.

O passo de recursão executa enquanto a chamada de método original ainda está aberta (isto é, não terminou de executar). Ele pode resultar em muitas outras chamadas recursivas à medida que o método divide cada novo subproblema em duas partes conceituais. Para a recursão por fim terminar, toda vez que o método chamar a si próprio com uma versão mais simples do problema original, a sequência de problemas cada vez menores deve convergir para um caso básico. Quando o método reconhece o caso básico, ele retorna um resultado para a cópia anterior do método. Uma sequência de retornos segue até a chamada de método original retornar o resultado final para o chamador.

Um método recursivo pode chamar outro método, que, por sua vez, pode fazer uma chamada de volta ao método recursivo. Isso é conhecido como uma **chamada recursiva indireta** ou **recursão indireta**. Por exemplo, o método `A` chama o método `B`, que faz uma chamada de volta ao método `A`. Isso ainda é recursão, porque a segunda chamada para o método `A` é feita enquanto a primeira chamada ao método `A` está ativa, isto é, a primeira chamada ao método `A` ainda não concluiu sua execução (porque está esperando o método `B` retornar um resultado para ela) e não retornou ao chamador original do método `A`.

Para entender melhor o conceito de recursão, veremos um exemplo que é bem comum aos usuários de computador — a definição recursiva de um diretório em um computador. Um computador normalmente armazena arquivos relacionados em um diretório. Um diretório pode estar vazio, pode conter arquivos e/ou conter outros diretórios (normalmente conhecidos como subdiretórios). Cada um desses subdiretórios, por sua vez, também pode conter tanto arquivos como diretórios. Se quisermos listar cada arquivo em um diretório (incluindo todos os arquivos nos subdiretórios do diretório), precisamos criar um método que primeiro lista os arquivos do diretório inicial e, então, faz chamadas recursivas para listar os arquivos em todos os subdiretórios desse diretório. O caso básico ocorre quando um diretório que é alcançado não contém nenhum subdiretório. Nesse ponto, todos os arquivos no diretório original foram listados e nenhuma recursão adicional é necessária.

18.3 Exemplo que utiliza recursão: fatoriais

Vamos escrever um programa recursivo para efetuar um cálculo matemático popular. Considere o fatorial de um inteiro positivo n, escrito $n!$ (pronuncia-se "n fatorial"), que é o produto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

com 1! igual a 1 e 0! definido como 1. Por exemplo, 5! é o produto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que é igual a 120.

O fatorial de inteiro `number` (onde `number` ≥ 0) pode ser calculado iterativamente (não recursivamente) utilizando-se uma instrução `for` como a seguinte:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

Chega-se a uma declaração recursiva do método fatorial observando o seguinte relacionamento:

$$n! = n \cdot (n - 1)!$$

Por exemplo, 5! é claramente igual a $5 \cdot 4!$, como mostrado pelas seguintes equações:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

A avaliação de 5! prosseguiria como mostrado na Figura 18.2. A Figura 18.2(a) mostra como a sucessão de chamadas recursivas prossegue até 1! (o caso básico) é avaliado como 1, que termina a recursão. A Figura 18.2(b) mostra os valores retornados de cada chamada recursiva para seu chamador até que o valor final seja calculado e retornado.

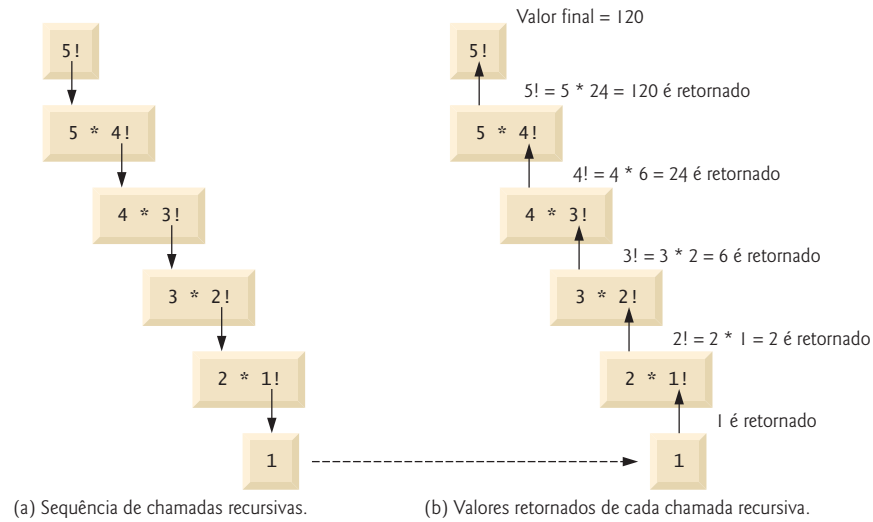


Figura 18.2 | Avaliação recursiva de 5!

A Figura 18.3 utiliza a recursão para calcular e imprimir os fatoriais dos inteiros de 0–21. O método recursivo `factorial` (linhas 7–13) primeiro testa para determinar se uma condição de término (linha 9) é `true`. Se `number` for menor que ou igual a `1` (o caso básico), `factorial` retorna `1`, nenhuma recursão adicional é necessária e o método retorna. Se `number` for maior que `1`, a linha 12 expressa o problema como o produto de `number` e uma chamada recursiva para `factorial` avaliando o fatorial de `number - 1`, que é um problema ligeiramente menor que o cálculo original, `factorial( number )`.

```
1  // Figura 18.3: FactorialCalculator.java
2  // Método fatorial recursivo.
3
4  public class FactorialCalculator
5  {
6     // método recursivo fatorial (assume que seu parâmetro é >= 0)
7     public long factorial( long number )
8     {
9        if ( number <= 1 ) // testa caso básico
10          return 1; // casos básicos: 0! = 1 e 1! = 1
11       else // passo de recursão
12          return number * factorial( number - 1 );
13    } // fim do método factorial
14
15    // gera saída de fatoriais para valores 0-21
16    public static void main( String[] args )
17    {
18       // calcula o fatorial de 0 a 21
19       for ( int counter = 0; counter <= 21; counter++ )
20          System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
21    } // fim de main
22 } // fim da classe FactorialCalculator
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
12! = 479001600 —— 12! causes overflow for int variables
...
20! = 2432902008176640000
21! = -4249290049419214848 —— 21! causes overflow for long variables
```

Figura 18.3 | Cálculos fatoriais com um método recursivo.

Erro comum de programação 18.1

Omitir o caso básico ou escrever o passo de recursão incorretamente de modo que não convirja para o caso básico pode causar um erro de lógica conhecido como ***recursão infinita****, em que as chamadas recursivas são feitas continuamente até acabar a memória. Isso é análogo ao problema de um loop infinito em uma solução iterativa (não recursiva).*

O método `main` (linhas 16–21) exibe os fatoriais de 0–21. A chamada para o método `factorial` ocorre na linha 20. O método `factorial` recebe um parâmetro de tipo `long` e retorna um resultado do tipo `long`. A saída do programa mostra que os valores fatoriais tornam-se rapidamente grandes. Utilizamos o tipo `long` (que pode representar inteiros relativamente grandes) para que o programa possa calcular fatoriais maiores que 12! Infelizmente, o método `factorial` produz valores grandes tão rapidamente que excedemos o maior valor `long` quando tentamos calcular 21!, como você pode ver na última linha da saída do programa.

Por causa das limitações dos tipos inteiros, as variáveis `float` ou `double` podem, em última instância, ser necessárias para calcular fatoriais de números maiores. Isso indica uma fraqueza em algumas linguagens de programação — a saber, que elas não são facilmente estendidas com novos tipos para tratar requisitos de aplicativo únicos. Como vimos no Capítulo 9, o Java é uma linguagem extensível que permite criar inteiros arbitrariamente grandes se quisermos. De fato, o pacote `java.math` fornece classes **`BigInteger`** e **`BigDecimal`** explicitamente para cálculos de precisão arbitrária que não podem ser efetuados com tipos primitivos. Você pode aprender mais sobre essas classes em `java.sun.com/javase/6/docs/api/java/math/BigInteger.html` e `java.sun.com/javase/6/docs/api/java/math/BigDecimal.html`, respectivamente.

Reimplementando a classe `FactorialCalculator` com a classe `BigInteger`

A Figura 18.4 reimplementa a classe `FactorialCalculator` utilizando variáveis `BigInteger`. Para demonstrar valores maiores do que o que as variáveis `long` podem armazenar, calculamos os fatoriais dos números 0–50. A linha 3 importa a classe `BigInteger` do pacote `java.math`. O novo método `factorial` (linhas 8–15) recebe um `BigInteger` como um argumento e retorna um `BigInteger`.

```
1  // Figura 18.4: FactorialCalculator.java
2  // Método fatorial recursivo.
3  import java.math.BigInteger;
4
5  public class FactorialCalculator
6  {
7     // método recursivo fatorial (assume que seu parâmetro é >= 0)
8     public static BigInteger factorial( BigInteger number )
9     {
10       if ( number.compareTo( BigInteger.ONE ) <= 0 ) // caso básico de teste
11          return BigInteger.ONE; // casos básicos: 0! = 1 e 1! = 1
12       else // passo de recursão
13          return number.multiply(
14             factorial( number.subtract( BigInteger.ONE ) ) );
15    } // fim do método factorial
16
17    // gera saída de fatoriais para valores 0-50
18    public static void main( String[] args )
19    {
20       // calcula o fatorial de 0 a 50
21       for ( int counter = 0; counter <= 50; counter++ )
22          System.out.printf( "%d! = %d\n", counter,
23             factorial( BigInteger.valueOf( counter ) ) );
24    } // fim de main
25 } // fim da classe FactorialCalculator
```

```
0! = 1
1! = 1
2! = 2
3! = 6
...
21! = 51090942171709440000021! and larger values no longer cause overflow
22! = 1124000727777607680000
...
47! = 258623241511168180642964355153611979969197632389120000000000
48! = 12413915592536072670862289047373375038521486354677760000000000
49! = 608281864034267560872252163321295376887552831379210240000000000
50! = 30414093201713378043612608166064768844377641568960512000000000000
```

Figura 18.4 | Cálculos fatoriais com um método recursivo.

Como `BigInteger` não é um tipo primitivo, não podemos utilizar operadores aritméticos, relacionais e de igualdade com `BigIntegers`; em vez disso, devemos utilizar os métodos `BigInteger` para realizar essas tarefas. A linha 10 testa o caso básico utilizando o método `BigInteger` **`compareTo`**. Esse método compara o `BigInteger` que chama o método para o argumento `BigInteger` do método. O método retorna `-1` se o `BigIteger` que chama o método for menor do que o argumento, `0` se forem iguais ou `1` se `BigInteger` que chama o método que for maior do que o argumento. A linha 10 compara o `BigInteger number` com a constante `BigInteger`, que representa o valor inteiro 1. Se `compareTo` retorna `-1` ou `0`, então o número é menor ou igual a 1 (o caso básico) e o método retorna a constante `BigInteger.ONE`. Caso contrário, as linhas 13–14 realizam o passo de recursão utilizando os métodos `BigInteger` **`multiply`** e **`subtract`** para implementar os cálculos necessários para multiplicar `number` pelo fatorial de `number - 1`). A saída do programa mostra que `BigInteger` trata facilmente os valores grandes produzidos pelo cálculo fatorial.

18.4 Exemplo que utiliza recursão: série de Fibonacci

A **série de Fibonacci**

0, 1, 1, 2, 3, 5, 8, 13, 21, …

inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois anteriores. Essa série ocorre na natureza e descreve a forma de uma espiral. A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618…, um número que é chamado de **relação áurea** ou **média áurea.** Humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos frequentemente projetam janelas, salas e edifícios cujo comprimento e largura estão na relação da média áurea. Cartões postais são frequentemente projetados com uma relação de comprimento/largura igual à relação áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(*n*) = fibonacci(*n* – 1) + fibonacci(*n* – 2)

Observe que há *dois casos básicos* para o cálculo de Fibonacci: `fibonacci(0)` é definido como 0, e `fibonacci(1)` como 1. O programa na Figura 18.5 calcula recursivamente o i-ésimo número de Fibonacci, utilizando o método `fibonacci` (linhas 10–18). O método `main` (linhas 21–26) testa `fibonacci`, exibindo os valores de Fibonacci de 0–40. A variável `counter` criada no cabeçalho `for` (linha 23) indica qual número de Fibonacci calcular para cada iteração do loop. Os números Fibonacci tendem a se tornar rapidamente grandes (embora não tão rapidamente como os fatoriais). Portanto, utilizamos o tipo `BigInteger` como o tipo de parâmetro e o tipo de retorno do método `fibonacci`.

```
1  // Figura 18.5: FibonacciCalculator.java
2  // Método recursivo de Fibonacci.
3  import java.math.BigInteger;
4
5  public class FibonacciCalculator
6  {
7     private static BigInteger TWO = BigInteger.valueOf( 2 );
8
9     // declaração recursiva do método fibonacci
10    public static BigInteger fibonacci( BigInteger number )
11    {
12       if ( number.equals( BigInteger.ZERO ) ||
13            number.equals( BigInteger.ONE ) ) // casos básicos
14          return number;
15       else // passo de recursão
16          return fibonacci( number.subtract( BigInteger.ONE ) ).add(
17             fibonacci( number.subtract( TWO ) ) );
18    } // fim do método fibonacci
19
20    // exibe os valores de fibonacci de 0-40
21    public static void main( String[] args )
22    {
23       for ( int counter = 0; counter <= 40; counter++ )
24          System.out.printf( "Fibonacci of %d is: %d\n", counter,
25             fibonacci( BigInteger.valueOf( counter ) ) );
26    } // fim de main
27 } // fim da classe FibonacciCalculator
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Figura 18.5 | Os números de Fibonacci gerados com um método recursivo.

A chamada para o método `fibonacci` (linha 25 da Figura 18.5) de `main` não é uma chamada recursiva, mas todas as chamadas subsequentes a `fibonacci` realizadas a partir das linhas 16–17 da Figura 18.5 são recursivas, porque nesse ponto as chamadas são iniciadas pelo próprio método `fibonacci`. Toda vez que `fibonacci` é chamado, ele testa imediatamente para os casos básicos — `number` igual a 0 ou `number` igual a 1 (Figura 18.5, linhas 12–13). Observe que utilizamos constantes `BigInteger` **`ZERO`** e `ONE` para representar os valores 0 e 1, respectivamente. Se a condição nas linhas 12–13 for `true`, `fibonacci` simplesmente retorna `number`, porque `fibonacci(0)` é 0 e `fibonacci(1)` é 1. Curiosamente, se `number` for maior que 1, o passo de recursão gera *duas* chamadas recursivas (linhas 16–17), cada uma para um problema ligeiramente menor que a chamada original a `fibonacci`. As linhas 16–17 utilizam os métodos `BigInteger` **`add`** e `subtract` para ajudar a implementar o passo recursivo. Também utilizamos uma constante do tipo `BigInteger` chamada `TWO` que definimos na linha 7.

Analisando as chamadas para o método `Fibonacci`

A Figura 18.6 mostra como o método `fibonacci` avalia `fibonacci(3)`. Observe que, na parte inferior da figura, resta-nos os valores 1, 0 e 1 — os resultados da avaliação dos casos básicos. Os primeiros dois valores de retorno (da esquerda para direita), 1 e 0, são retornados como os valores das chamadas `fibonacci(1)` e `fibonacci(0)`. A soma 1 mais 0 é retornada como o valor de `fibonacci(2)`. Isso é adicionado ao resultado (1) da chamada para `fibonacci(1)`, produzindo o valor 2. Esse valor final é então retornado como o valor de `fibonacci(3)`.

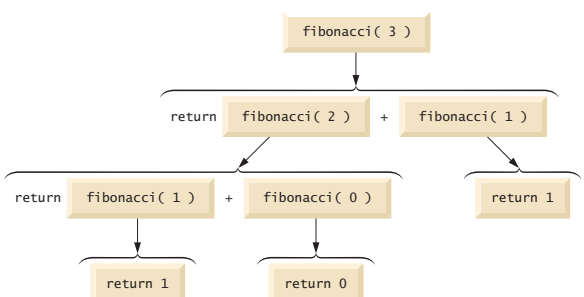


Figura 18.6 | Conjunto de chamadas recursivas para `fibonacci( 3 )`.

A Figura 18.6 levanta algumas questões interessantes sobre a ordem em que compiladores Java avaliam os operandos de operadores. Essa ordem é diferente daquela em que os operadores são aplicados a seus operandos — a saber, a ordem ditada pelas regras de precedência de operadores. A partir da Figura 18.6, parece que enquanto `fibonacci(3)` está sendo avaliada, duas chamadas recursivas serão feitas — `fibonacci(2)` e `fibonacci(1)`. Mas em que ordem elas serão feitas? A linguagem Java especifica que a ordem de avaliação dos operandos é da direita para a esquerda. Portanto, a chamada `fibonacci(2)` é feita primeiro e a `fibonacci(1)`, depois.

Uma palavra de cautela está em ordem sobre programas recursivos como os utilizamos aqui para gerar números de Fibonacci. Cada invocação do método `fibonacci` que não corresponder a um dos casos básicos (0 ou 1) resulta em *mais duas chamadas recursivas* ao método `fibonacci`. Por isso, esse conjunto de chamadas recursivas foge do controle rapidamente. Calcular o valor de Fibonacci de 20 com o pro-

grama na Figura 18.5 requer 21.891 chamadas para o método `fibonacci`; calcular o valor de Fibonacci de 30 requer 2.692.537 chamadas! À medida que tentar calcular valores de Fibonacci maiores, você notará que cada número de Fibonacci consecutivo que solicita ao aplicativo para calcular resulta em um aumento substancial no tempo de cálculo e no número de chamadas para o método `fibonacci`. Por exemplo, o valor de Fibonacci de 31 requer 4.356.617 chamadas e o valor de Fibonacci de 32 requer 7.049.155 chamadas! Como você pode ver, o número de chamadas para `fibonacci` aumenta rapidamente — 1.664.080 chamadas adicionais entre valores de Fibonacci de 30 e 31 e 2.692.538 chamadas adicionais entre valores de Fibonacci de 31 e 32! A diferença no número de chamadas feitas entre valores de Fibonacci de 31 e 32 é mais que 1,5 vez a diferença no número de chamadas por valores Fibonacci entre 30 e 31. Problemas dessa natureza podem humilhar até mesmo os computadores mais poderosos do mundo. [*Nota*: No campo da teoria da complexidade, os cientistas da computação estudam a rapidez com que os algoritmos trabalham para completar suas tarefas. As questões de complexidade são discutidas em detalhes no curso do currículo de ciência da computação de nível superior geralmente chamado "Algoritmos". Introduzimos várias questões de complexidade no Capítulo 19, Pesquisa, Classificação e Big O]. Nos exercícios, você será solicitado a aprimorar o programa Fibonacci da Figura 18.5 para que calcule o período de tempo aproximado necessário para realizar o cálculo. Para esse propósito, você chamará o método `static` `System` `currentTimeMillis`, que não aceita nenhum argumento e retorna a hora atual do computador em milissegundos.

Dica de desempenho 18.1

Evite programas recursivos no estilo Fibonacci, porque resultam em uma "explosão" exponencial de chamadas de método.

18.5 Recursão e a pilha de chamadas de método

No Capítulo 6, a estrutura de dados de pilha foi introduzida no contexto de entender como o Java realiza chamadas de método. Discutimos tanto a pilha de chamadas de método (também conhecida como a pilha de execução do programa) como os registros de ativação. Nesta seção, utilizaremos esses conceitos para demonstrar como a pilha de execução do programa trata as chamadas de método recursivo.

Vamos começar voltando para o exemplo do Fibonacci — especificamente, chamando o método `fibonacci` com o valor `3`, como na Figura 18.6. Para mostrar a ordem em que os registros de ativação das chamadas de método são colocados na pilha, marcamos as chamadas de método na Figura 18.7 com letras.

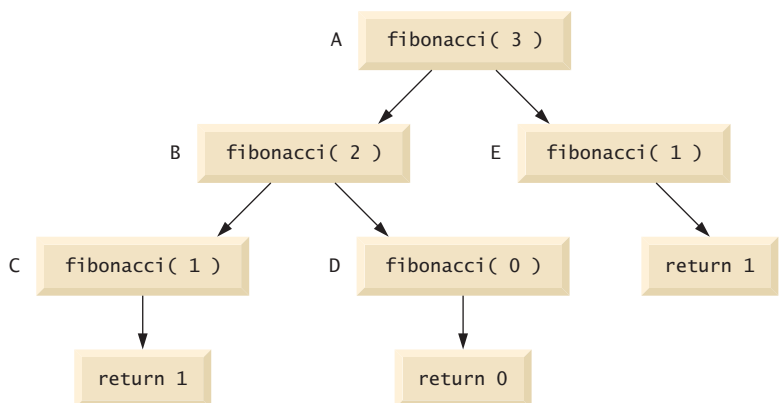


Figura 18.7 | Chamadas de método feitas dentro da chamada `fibonacci(3)`.

Quando a primeira chamada de método (`A`) é feita, um registro de ativação contendo o valor da variável local `number` (`3`, nesse caso) é adicionado à pilha de execução do programa. A pilha de execução do programa, incluindo o registro de ativação para chamada do método `A`, é ilustrada na parte (a) da Figura 18.8. [*Nota:* utilizamos uma pilha simplificada aqui. Uma pilha real de execução do programa e seus registros de ativação seriam mais complexos que na Figura 18.8, contendo informações como onde a chamada de método deve retornar ao completar a execução.]

Dentro da chamada de método `A`, as chamadas de método `B` e `E` são feitas. A chamada de método original ainda não foi concluída, então seu registro de ativação permanece na pilha. A primeira chamada de método a ser feita dentro de `A` é a chamada de método `B`, então o registro de ativação para a chamada de método `B` é adicionado à pilha na parte superior daquele para a chamada de método `A`. A chamada de método `B` deve executar e completar antes de a chamada de método `E` ser feita. Dentro da chamada de método `B`, as chamadas de método `C` e `D` serão feitas. A chamada de método `C` é feita primeiro e seu registro de ativação é adicionado à pilha [parte (b) da Figura 18.8]. A chamada de método `B` ainda não terminou e seu registro de ativação ainda está na pilha de chamada de método. Quando a chamada de método `C` executa, ela não faz mais chamadas de método, mas simplesmente retorna o valor `1`. Quando esse método retornar, seu registro de ativação é removido da parte superior da pilha. Agora a chamada de método na parte superior da pilha é `B`, que continua a executar realizando a chamada de método `D`. O registro de ativação para a chamada de método `D` é adicionado à pilha [parte (c) da Figura 18.8]. A chamada de

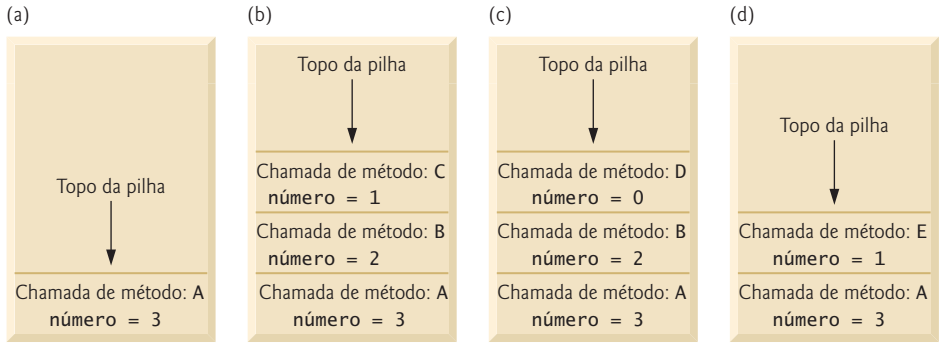


Figura 18.8 | Chamadas de método na pilha de execução do programa.

método `D` completa sem fazer mais nenhuma chamada de método e retorna o valor `0`. O registro de ativação dessa chamada de método é então removido da pilha. Agora, ambas as chamadas de método feitas de dentro da chamada de método `B` retornaram. A chamada de método `B` continua a executar, retornando o valor `1`. A chamada de método `B` é concluída e seu registro de ativação é removido da pilha. Nesse ponto, o registro de ativação para a chamada de método `A` está na parte superior da pilha e o método continua sua execução. Esse método faz a chamada de método `E`, cujo registro de ativação é agora adicionado à pilha (parte (d) da Figura 18.8). A chamada de método `E` completa e retorna o valor `1`. O registro de ativação para essa chamada de método é removido da pilha e mais uma vez a chamada de método `A` continua a executar. Nesse ponto, a chamada de método `A` não estará fazendo nenhuma outra chamada de método e pode terminar sua execução, retornando o valor `2` ao chamador de `A` (`fibonacci(3) = 2`). O registro de ativação de `A` é removido da pilha. Observe que o método em execução é sempre aquele cujo registro de ativação está na parte superior da pilha, e o registro de ativação desse método contém os valores das suas variáveis locais.

18.6 Recursão *vs.* iteração

Estudamos os métodos `factorial` e `fibonacci`, que podem ser facilmente implementados recursiva ou iterativamente. Nesta seção, comparamos duas abordagens e discutimos por que você talvez escolha uma abordagem sobre outra em uma situação particular.

Tanto iteração como recursão se baseiam em uma estrutura de controle: A iteração utiliza uma instrução de repetição (por exemplo, `for`, `while` ou `do...while`), enquanto a recursão usa uma instrução de seleção (por exemplo, `if`, `if...else` ou `switch`). Ambas envolvem repetição: a iteração utiliza explicitamente uma instrução de repetição, enquanto a recursão alcança a repetição por meio de repetidas chamadas de método. Tanto uma como outra envolvem um teste de terminação: A iteração termina quando a condição de continuação do loop falha, enquanto a recursão termina quando um caso básico é alcançado. A iteração com repetição controlada por contador e a recursão gradualmente se aproximam do término: A iteração continua modificando um contador até que o contador assume um valor que faz a condição de continuação do loop falhar; a recursão continua produzindo versões menores do problema original até que o caso básico seja alcançado. Tanto uma como outra podem ocorrer infinitamente: Ocorre um loop infinito com a iteração se o teste de continuação do loop nunca tornar-se falso, enquanto a recursão infinita ocorre se o passo de recursão não reduzir o problema sempre em uma maneira que convirja no caso básico, ou se o caso básico não for testado.

Para ilustrar as diferenças entre iteração e recursão, vamos examinar uma solução iterativa para o problema fatorial (Figura 18.9). Observe que uma instrução de repetição é utilizada (linhas 12–13) em vez da instrução de seleção da solução recursiva (linhas 9–12 da Figura 18.3). Observe que as duas soluções utilizam um teste de terminação. Na solução recursiva, a linha 9 testa quanto ao caso básico. Na solução iterativa, a linha 12 testa a condição de continuação do loop — se o teste falhar, o loop termina. Por fim, observe que em vez de produzir versões menores do problema original, a solução iterativa utiliza um contador que é modificado até a condição de continuação do loop tornar-se falsa.

```
1  // Figura 18.9: FactorialCalculator.java
2  // Método fatorial iterativo.
3
4  public class FactorialCalculator
5  {
6     // declaração recursiva de método factorial
7     public long factorial( long number )
8     {
9        long result = 1;
10
11       // declaração iterativa de método factorial
12       for ( long i = number; i >= 1; i-- )
13          result *= i;
```

```
14
15          return result;
16       } // fim do método factorial
17
18       // gera saída de fatoriais para valores 0-10
19       public static void main( String[] args )
20       {
21          // calcula o fatorial de 0 a 10
22          for ( int counter = 0; counter <= 10; counter++ )
23             System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
24       } // fim de main
25    } // fim da classe FactorialCalculator
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Figura 18.9 | Solução fatorial iterativa.

A recursão tem muitos negativas. Ela invoca repetidamente o mecanismo, e consequentemente o overhead, das chamadas de método. Essa repetição pode ser cara tanto em termos de tempo de processador como de espaço de memória. Cada chamada recursiva faz com que outra cópia do método (na verdade, somente as variáveis do método, armazenadas no registro de ativação) seja criada — esse conjunto de cópias pode consumir espaço considerável de memória. Visto que a iteração ocorre dentro de um método, as chamadas de método repetidas e a atribuição extra de memória são evitadas. Então por que escolher recursão?

Observação de engenharia de software 18.1

Qualquer problema que pode ser resolvido recursivamente também pode ser resolvido iterativamente (não recursivamente). Uma abordagem recursiva normalmente é preferida sobre uma abordagem iterativa quando a abordagem recursiva espelha mais naturalmente o problema e resulta em um programa mais fácil de entender e depurar. Uma abordagem recursiva pode ser frequentemente implementada com menos linhas de código. Outra razão de escolher uma abordagem recursiva é que uma iterativa talvez não seja aparente.

Dica de desempenho 18.2

Evite usar a recursão em situações que requerem alto desempenho. Chamadas recursivas levam tempo e consomem memória adicional.

Erro comum de programação 18.2

Ter acidentalmente um método não recursivo chamando a si próprio seja direta ou indiretamente por outro método pode causar recursão infinita.

18.7 Torres de Hanói

Estudamos neste capítulo os métodos que podem ser facilmente implementados tanto recursiva como iterativamente. Agora, apresentamos um problema cuja solução recursiva demonstra a elegância da recursão, e cuja solução iterativa pode não ser tão aparente.

As **Torres de Hanói** são um dos problemas clássicos com os quais todo cientista da computação deve lidar. Diz a lenda que, em um templo no Extremo Oriente, os sacerdotes tentavam mover uma pilha de discos dourados de um pino de diamante para outro (Figura 18.10). A pilha inicial tinha 64 discos sobrepostos em um pino e organizados de baixo para cima por tamanho decrescente. Os sacerdotes tentavam mover a pilha de um pino para outro sob as restrições de que exatamente um disco seria movido por vez e, em nenhuma circunstância, um disco maior poderia ser colocado em cima de um disco menor. Três pinos eram fornecidos e um deles era utilizado para armazenar discos temporariamente. Supostamente, o mundo acabará quando os sacerdotes completarem sua tarefa, portanto há pouco incentivo para nós facilitarmos seus esforços.

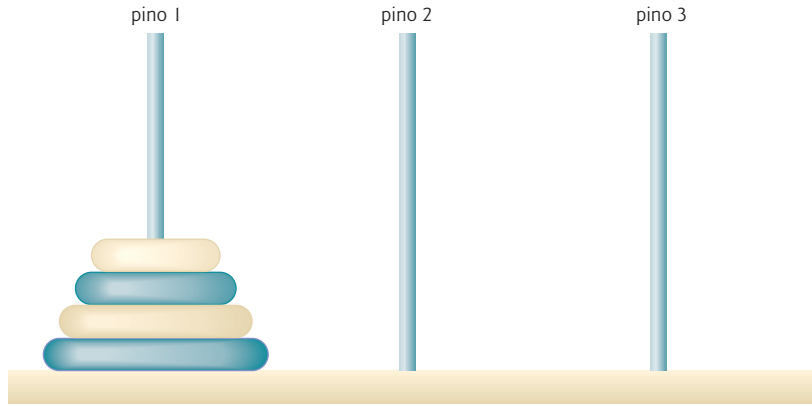


Figura 18.10 | As Torres de Hanói para o caso com quatro discos.

Vamos supor que os sacerdotes estejam tentando mover os discos do pino 1 para o pino 3. Desejamos desenvolver um algoritmo que imprima a sequência precisa de transferências de discos de um pino para outro.

Se tentarmos achar uma solução iterativa, é provável que nos encontremos irremediavelmente "amarrados" no gerenciamento dos discos. Em vez disso, atacar esse problema recursivamente com rapidez produz uma solução. Mover n discos pode ser visualizado em termos de mover somente $n - 1$ discos (daí a recursão) como a seguir:

1. Mova $n - 1$ disco do pino 1 para o pino 2, utilizando o pino 3 como área de armazenamento temporário.
2. Mova o último disco (o maior) doe pino 1 para o pino 3.
3. Mova $n - 1$ disco do pino 2 para o pino 3, utilizando o pino 1 como área de armazenamento temporário.

O processo termina quando a última tarefa envolve mover $n = 1$ disco (isto é, o caso básico). Essa tarefa é realizada movendo o disco, sem utilizar uma área de armazenamento temporária.

Na Figura 18.11, o método `solveTowers` (linhas 6–25) resolve o problema das Torres de Hanói, dado o número total de discos (nesse caso 3), o pino inicial, o pino final e o pino armazenado temporariamente como parâmetros. O caso básico (linhas 10–14) ocorre quando um único disco precisa ser movido do pino inicial para o pino final. No passo de recursão (linhas 18–24), a linha 18 move `disks - 1` discos do primeiro pino (`sourcePeg`) para o pino armazenado temporariamente (`tempPeg`). Quando todos exceto um dos discos foram movidos para o pino temporário, a linha 21 move o disco maior do pino de inicial para o pino de destino. A linha 24 conclui os demais movimentos chamando o método `solveTowers` para mover recursivamente os discos `disks - 1` do pino temporário (`tempPeg`) para o pino de destino (`destinationPeg`), dessa vez utilizando o primeiro pino (`sourcePeg`) como o pino temporário. A linha 35 em `main` chama o método recursivo `solveTowers`, que gera saída dos passos para o prompt de comando.

```
1  // Figura 18.11: TowersOfHanoi.java
2  // Solução do problema das Torres de Hanói com um método recursivo.
3  public class TowersOfHanoi
4  {
5     // move recursivamente os discos entre as torres
6     public static void solveTowers( int disks, int sourcePeg,
7        int destinationPeg, int tempPeg )
8     {
9        // caso básico -- somente um disco a ser movido
10       if ( disks == 1 )
11       {
12          System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );
13          return;
14       } // fim do if
15
16       // passo de recursão - move (disco - 1) discos de sourcePeg
17       // para tempPeg utilizando destinationPeg
18       solveTowers( disks - 1, sourcePeg, tempPeg, destinationPeg );
19
20       // move o último disco de sourcePeg para destinationPeg
21       System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );
22
23       // move ( disks  - 1 ) discos de tempPeg para destinationPeg
24       solveTowers( disks - 1, tempPeg, destinationPeg, sourcePeg );
25    } // fim do método solveTowers
```

```
26
27      public static void main( String[] args )
28      {
29         int startPeg = 1; // valor 1 utilizado para indicar startPeg na saída
30         int endPeg = 3; // valor 3 utilizado para indicar endPeg na saída
31         int tempPeg = 2; // valor 2 utilizado para indicar tempPeg na saída
32         int totalDisks = 3; // número de discos
33
34         // chamada não recursiva inicial: move todos os discos.
35         towersOfHanoi.solveTowers( totalDisks, startPeg, endPeg, tempPeg );
36      } // fim de main
37   } // fim da classe TowersOfHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Figura 18.11 | Solução do problema das Torres de Hanói com um método recursivo.

18.8 Fractais

Um **fractal** é uma figura geométrica que pode ser gerada de um padrão repetido recursivamente (Figura 18.12). A figura é modificada aplicando o padrão a cada segmento da figura original. Examinaremos algumas dessas aproximações nesta seção. [*Nota:* iremos nos referir às nossas figuras geométricas como fractais, ainda que sejam aproximações]. Embora essas figuras tenham sido estudadas antes do século XX, foi o matemático Benoit Mandelbrot que introduziu o termo "fractal" na década de 1970, com as especificidades de como um fractal é criado e de suas aplicações práticas. A geometria fractal de Mandelbrot fornece modelos matemáticos para muitas formas complexas encontradas na natureza, como montanhas, nuvens e litorais. Os fractais têm muitos usos na matemática e ciência. Eles podem ser utilizados para entender melhor os sistemas ou padrões que aparecem na natureza (por exemplo, ecossistemas), no corpo humano (por exemplo, nas circunvoluções cerebrais) ou no universo (por exemplo, grupos de galáxias). Nem todos os fractais se parecem com objetos na natureza. Desenhar fractais tornou-se uma forma popular de arte. Os fractais têm uma **propriedade autossimilar** — quando subdivididos em partes, cada parte parece uma cópia de tamanho reduzido do total. Muitos fractais produzem uma cópia exata do original quando uma parte do fractal é ampliada — diz-se que um fractal é **estritamente autossimilar**. Consulte nosso Recursion Resource Center (`www.deitel.com`) para sites web que demonstram fractais.

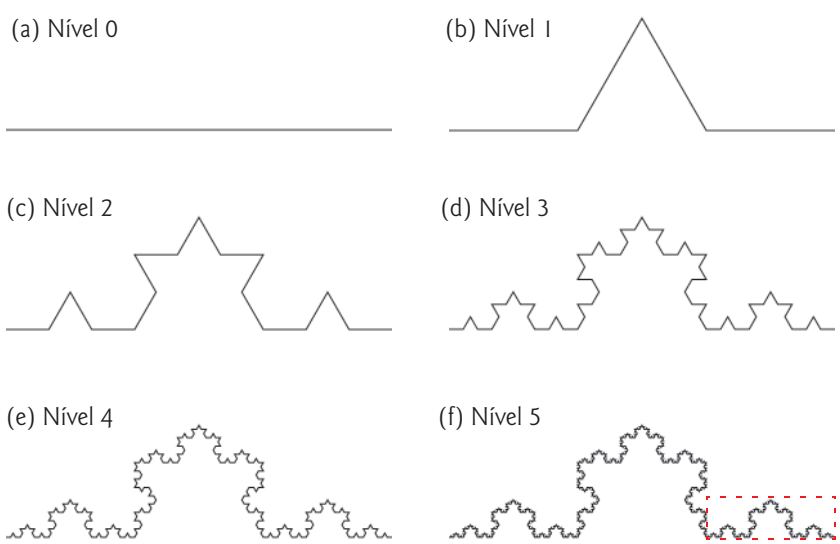


Figura 18.12 | Fractal Curva de Koch.

Como exemplo, vejamos o fractal **Curva de Koch** estritamente autossimilar (Figura 18.12). Ele é formado removendo o terço médio de cada linha no desenho e substituindo-o por duas linhas que formam um ponto, de tal modo que se esse terço médio permanecesse no meio da linha original, um triângulo equilateral seria formado. As formas para criar fractais costumam envolver a remoção de toda ou parte da imagem fractal anterior. Esse padrão já foi determinado para esse fractal — focalizamos aqui como usar essas fórmulas em uma solução recursiva.

Iniciamos com uma linha reta (Figura 18.12(a)) e aplicamos o padrão, criando um triângulo a partir do terço médio (Figura 18.12(b)). Então, aplicamos novamente o padrão a cada linha reta, resultando na Figura 18.12(c). Toda vez que o padrão for aplicado, dizemos que o fractal está em um novo **nível**, ou **profundidade** (às vezes, o termo **ordem** também é utilizado). Fractais podem ser exibidos em muitos níveis — por exemplo, um fractal no nível 3 teve três iterações do padrão aplicado (Figura 18.12(d)). Depois de apenas algumas iterações, esse fractal começa a parecer com uma parte de um floco de neve (Figura 18.12(e) e (f)). Visto que esse é um fractal estritamente autossimilar, cada parte dele contém uma cópia exata do fractal. Na Figura 18.12(f), por exemplo, destacamos uma parte do fractal com uma caixa vermelha tracejada. Se aumentássemos o tamanho da imagem nessa caixa, ela seria exatamente semelhante ao fractal inteiro da parte (f).

Um fractal semelhante, o **Floco de neve de Koch**, é o mesmo da Curva de Koch, mas inicia com um triângulo em vez de iniciar com uma linha. O mesmo padrão é aplicado a cada lado do triângulo, resultando em uma imagem semelhante a um floco de neve fechado. Para simplificar, escolhemos focalizar a Curva de Koch. Para aprender mais sobre a Curva de Koch e Floco de neve de Koch, consulte os links no nosso Recursion Resource Center (`www.deitel.com`).

"Fractal de Lo"

Agora demonstramos o uso da recursão para desenhar fractais escrevendo um programa para criar um fractal estritamente autossimilar. Chamamos esse fractal de "fractal de Lo," em homenagem a Sin Han Lo, um colega da Deitel & Associates que o criou. O fractal por fim será parecido com uma metade de uma pena (ver a saída na Figura 18.19). O caso básico, ou nível fractal de 0, inicia como uma linha entre dois pontos, `A` e `B` (Figura 18.13). Para criar o próximo nível mais alto, localizamos o ponto intermediário (`C`) da linha. Para calcular a localização do ponto `C`, utilizamos a seguinte fórmula:

```
xC = (xA + xB) / 2;
yC = (yA + yB) / 2;
```

[*Nota*: o `x` e `y` à esquerda de cada letra referem-se à coordenada *x* e à coordenada y desse ponto, respectivamente. Por exemplo, `xA` refere-se à coordenada *x* do ponto `A`, enquanto `yC` refere-se à coordenada *y* do ponto `C`. Em nossos diagramas denotamos o ponto por sua letra, seguido por dois números que representam as coordenadas *x* e *y*].

Para criar esse fractal, também devemos localizar um ponto `D` que resida à esquerda de segmento `AC` e cria um triângulo isósceles reto `ADC`. Para calcular a localização do ponto `D`, utilize as seguintes fórmulas:

```
xD = xA + (xC - xA) / 2 - (yC - yA) / 2;
yD = yA + (yC - yA) / 2 + (xC - xA) / 2;
```

Figura 18.13 | "Fractal de Lo" no nível 0.

Agora nos movemos do nível 0 para o nível 1 como mostrado a seguir: Inicialmente, adicionamos os pontos `C` e `D` (como na Figura 18.14). Então, removemos a linha original e adicionamos os segmentos `DA`, `DC` e `DB`. As linhas restantes se curvarão em um ângulo, fazendo com que nosso fractal se pareça com uma pena. Para o próximo nível do fractal, esse algoritmo é repetido em cada uma das três linhas no nível 1. Para cada linha, as fórmulas acima são aplicadas, nas quais o primeiro ponto `D` agora é considerado o ponto `A`, enquanto a outra extremidade de cada linha é considerada o ponto `B`. A Figura 18.15 contém a linha do nível 0 (agora uma linha tracejada) e três linhas adicionais do nível 1. Mudamos o ponto `D` para o ponto `A`, e os pontos originais `A`, `C` e `B` para `B1`, `B2` e `B3`, respectivamente. As fórmulas precedentes foram utilizadas para localizar os novos pontos `C` e `D` em cada linha. Esses pontos também são numerados 1–3 para monitorar que ponto está associado com cada linha. Os

pontos `C1` e `D1`, por exemplo, representam os pontos `C` e `D` associados com a linha formada a partir do ponto `A` para o ponto `B1`. Para alcançar o nível 2, as três linhas na Figura 18.15 são removidas e substituídas pelas novas linhas dos pontos `C` e `D` que acabaram de ser adicionados. A Figura 18.16 mostra as novas linhas (as linhas do nível 2 são mostradas como linhas tracejadas para sua conveniência). A Figura 18.17 mostra o nível 2 sem as linhas tracejadas do nível 1. Uma vez que esse processo foi repetido várias vezes, o fractal criado começará a parecer-se com metade de uma pena, como mostrado na saída da Figura 18.19. Apresentaremos o código para esse aplicativo em breve.

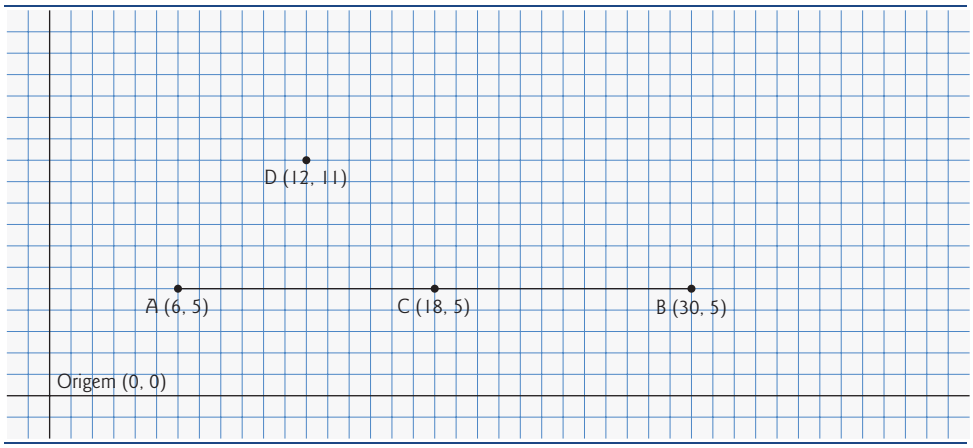


Figura 18.14 | Determinando os pontos C e D para o nível 1 do "Fractal de Lo".

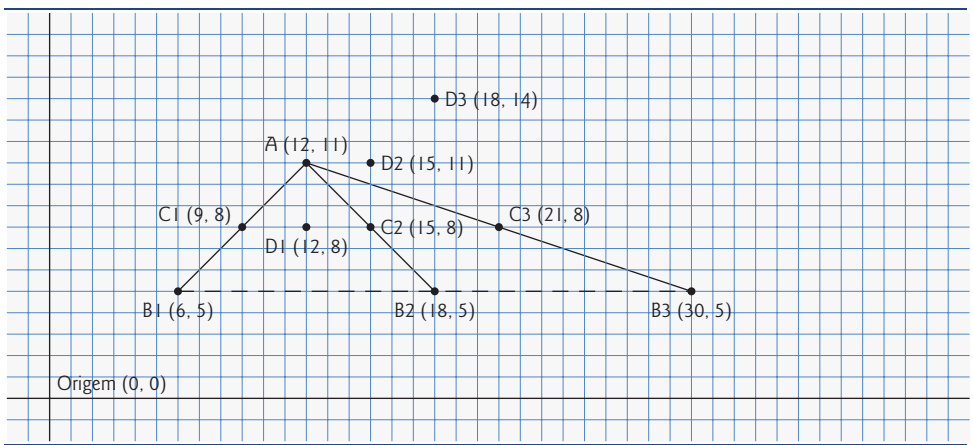


Figura 18.15 | "Fractal de Lo" no nível 1, com os pontos C e D determinados para o nível 2. [*Nota:* o fractal no nível 0 está incluído como uma linha tracejada como um lembrete de onde a linha foi localizada em relação ao fractal atual.]

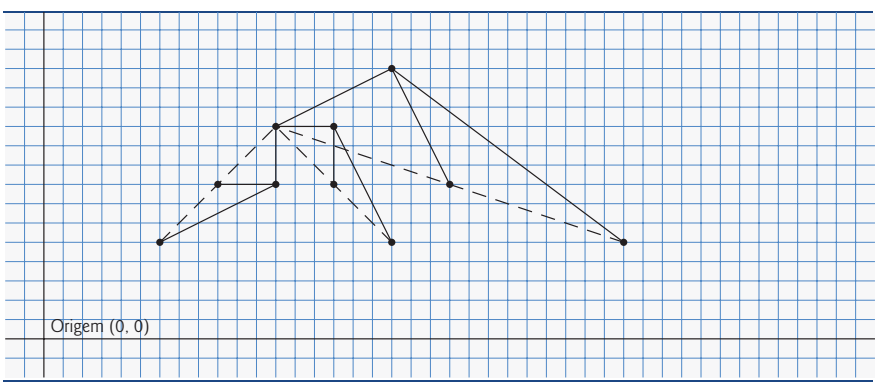


Figura 18.16 | "Fractal de Lo" no nível 2, com linhas tracejadas do nível 1 fornecido.

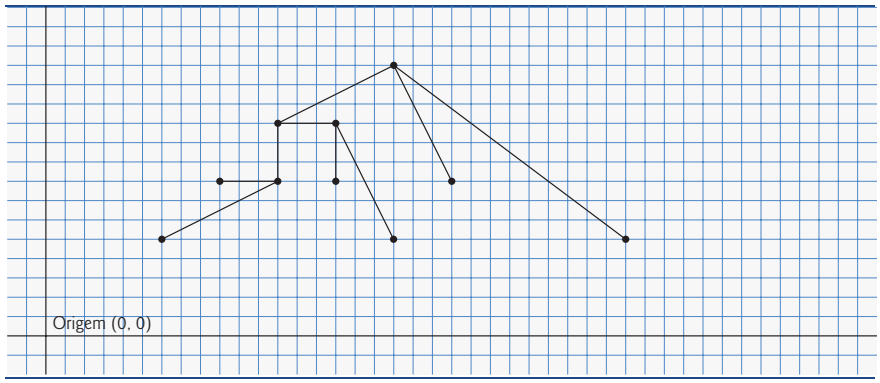


Figura 18.17 | "Fractal de Lo" no nível 2.

O aplicativo na Figura 18.18 define a interface com o usuário para desenhar esse fractal (mostrado no fim da Figura 18.19). A interface consiste em três botões — um para o usuário alterar a cor do fractal, um para aumentar o nível de recursão e um para diminuir o nível de recursão. Um `JLabel` monitora o nível atual de recursão, que é modificado chamando-se o método `setLevel`, a ser discutido em breve. As linhas 15–16 especificam que as constantes `WIDTH` e `HEIGHT` são `400` e `480` respectivamente, para o tamanho do `JFrame`. O usuário provoca um `ActionEvent` clicando no botão **Color**. O handler de evento para esse botão é registrado nas linhas 37–53. O método `actionPerformed` exibe um `JColorChooser`. Esse diálogo retorna o objeto `Color` selecionado ou azul (se o usuário pressionar **Cancel** ou fechar o diálogo sem pressionar **OK**). A linha 50 chama o método `setColor` na classe `FractalJPanel` para atualizar a cor.

```
1  // Figura 18.18: Fractal.java
2  // Interface com o usuário do fractal.
3  import java.awt.Color;
4  import java.awt.FlowLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JColorChooser;
12
13 public class Fractal extends JFrame
14 {
15    private static final int WIDTH = 400;  // define a largura de GUI
16    private static final int HEIGHT = 480; // define a altura de GUI
17    private static final int MIN_LEVEL = 0, MAX_LEVEL = 15;
18
19    private JButton changeColorJButton, increaseLevelJButton,
20       decreaseLevelJButton;
21    private JLabel levelJLabel;
22    private FractalJPanel drawSpace;
23    private JPanel mainJPanel, controlJPanel;
24
25    // configura a GUI
26    public Fractal()
27    {
28       super( "Fractal" );
29
30       // configura o painel de controle
31       controlJPanel = new JPanel();
32       controlJPanel.setLayout( new FlowLayout() );
33
34       // configura o botão de cor e registra o ouvinte
35       changeColorJButton = new JButton( "Color" );
36       controlJPanel.add( changeColorJButton );
37       changeColorJButton.addActionListener(
38          new ActionListener() // classe interna anônima
39          {
40             // processa o evento changeColorJButton
```

```
41             public void actionPerformed( ActionEvent event )
42             {
43                Color color = JColorChooser.showDialog(
44                   Fractal.this, "Choose a color", Color.BLUE );
45
46                // configura a cor padrão, se nenhuma cor for retornada
47                if ( color == null )
48                   color = Color.BLUE;
49
50                drawSpace.setColor( color );
51             } // fim do método actionPerformed
52          }  // fim da classe interna anônima
53       ); // fim de addActionListener
54
55       // configura o botão decrease level para adicionar o painel de controle
56       // ouvinte registrado
57       decreaseLevelJButton = new JButton( "Decrease Level" );
58       controlJPanel.add( decreaseLevelJButton );
59       decreaseLevelJButton.addActionListener(
60          new ActionListener() // classe interna anônima
61          {
62             // processa o evento decreaseLevelJButton
63             public void actionPerformed( ActionEvent event )
64             {
65                int level = drawSpace.getLevel();
66                --level; // diminui o nível por um
67
68                // modifica o nível se possível
69                if ( ( level >= MIN_LEVEL ) ) &&
70                   ( level <= MAX_LEVEL ) )
71                {
72                   levelJLabel.setText( "Level: " + level );
73                   drawSpace.setLevel( level );
74                   repaint();
75                } // fim do if
76             } // fim do método actionPerformed
77          } // fim da classe interna anônima
78       ); // fim de addActionListener
79
80       // configura o botão increase level para adicionar o painel de controle
81       // e registra o ouvinte
82       increaseLevelJButton = new JButton( "Increase Level" );
83       controlJPanel.add( increaseLevelJButton );
84       increaseLevelJButton.addActionListener(
85          new ActionListener() // classe interna anônima
86          {
87             // processa o evento increaseLevelJButton
88             public void actionPerformed( ActionEvent event )
89             {
90                int level = drawSpace.getLevel();
91                ++level; // aumenta nível por um
92
93                // modifica o nível se possível
94                if ( ( level >= MIN_LEVEL ) ) &&
95                   ( level <= MAX_LEVEL ) )
96                {
97                   levelJLabel.setText( "Level: " + level );
98                   drawSpace.setLevel( level );
99                   repaint();
100               } // fim do if
101            } // fim do método actionPerformed
102         } // fim da classe interna anônima
103      ); // fim de addActionListener
104
105      // configura levelJLabel para adicionar ao controlJPanel
106      levelJLabel = new JLabel( "Level: 0" );
107      controlJPanel.add( levelJLabel );
```

```
108
109         drawSpace = new FractalJPanel( 0 );
110
111         // cria mainJPanel para conter controlJPanel e drawSpace
112         mainJPanel = new JPanel();
113         mainJPanel.add( controlJPanel );
114         mainJPanel.add( drawSpace );
115
116         add( mainJPanel ); // adiciona JPanel ao JFrame
117
118         setSize( WIDTH, HEIGHT ); // configura o tamanho de JFrame
119         setVisible( true ); // exibe JFrame
120      } // fim do construtor Fractal
121
122      public static void main( String[] args )
123      {
124         Fractal demo = new Fractal();
125         demo.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
126      } // fim de main
127   } // fim da classe Fractal
```

Figura 18.18 | Interface com o usuário do fractal.

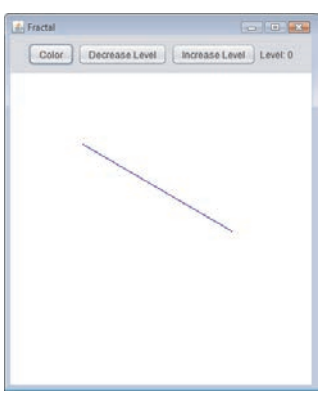
O handler de evento do botão Decrease Level é registrado nas linhas 59–78. O método `actionPerformed` recupera o nível atual da recursão e o decrementa pela 1 (linhas 65–66). As linhas 69–70 asseguram que o nível é maior ou igual a `MIN_LEVEL` e menor ou igual a `MAX_LEVEL` — o fractal não é definido pelos níveis de recursão menores que `MIN_LEVEL` e não é possível ver os detalhes adicionais acima de `MAX_LEVEL`. Você pode abordar de qualquer nível desejado, mas em torno do nível 10 e nível mais alto, a exibição de fractal torna-se lenta por causa dos detalhes a ser desenhados. As linhas 72–74 reconfiguram o rótulo de nível para refletir a alteração — o novo nível é configurado e o método `repaint` é chamado a fim de atualizar a imagem para mostrar o fractal ao novo nível.

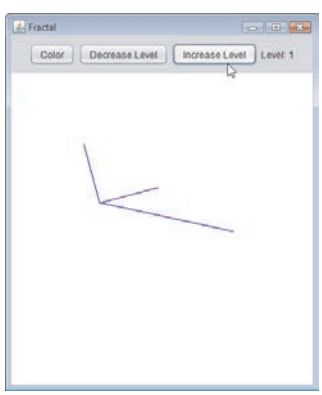
O `JButton` Increase Level funciona como o `JButton` Decrease Level, mas o nível é incrementado em vez de decrementado para mostrar mais detalhes do fractal (linhas 90–91). Quando o aplicativo é executado pela primeira vez, o nível será configurado como 0, o que exibirá uma linha azul entre dois pontos que foram especificados na classe `FractalJPanel`.

A classe `FractalJPanel` (Figura 18.19) especifica as dimensões do desenho `JPANEL` como `400` por `400` (linhas 13–14). O construtor `FractalJPanel` (linhas 18–24) aceita o nível atual como um parâmetro e o atribui à sua variável de instância `level`. A variável de instância `color` é configurada como azul por padrão. As linhas 22–23 mudam a cor de fundo do `JPANEL` para branco (portanto, é fácil de ver o fractal), e configuram as dimensões do desenho do `FractalJPanel`.

```
1  // Figura 18.19: FractalJPanel.java
2  // Desenhando o "fractal de Lo" com a recursão.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import java.awt.Dimension;
6  import javax.swing.JPanel;
7
8  public class FractalJPanel extends JPanel
9  {
10    private Color color; // armazena a cor utilizada para desenhar o fractal
11    private int level;   // armazena o nível atual do fractal
12
13    private static final int WIDTH = 400;  // define a largura do JPanel
14    private static final int HEIGHT = 400; // define a altura do JPanel
15
16    // configura o nível do fractal inicial com o valor especificado
17    // e configura as especificações do JPanel
18    public FractalJPanel( int currentLevel )
19    {
20       color = Color.BLUE;  // inicializa a cor desenho como azul
21       level = currentLevel; // configura o nível do fractal inicial
22       setBackground( Color.WHITE );
23       setPreferredSize( new Dimension( WIDTH, HEIGHT ) );
24    } // fim do construtor FractalJPanel
25
```

```
26       // desenha o fractal recursivamente
27       public void drawFractal( int level, int xA, int yA, int xB,
28          int yB, Graphics g )
29       {
30          // caso básico: desenha uma linha conectando dois pontos dados
31          if ( level == 0 )
32             g.drawLine( xA, yA, xB, yB );
33          else // passo de recursão: determina novos pontos desenha próximo nível
34          {
35             // calcula ponto intermediário entre (xA, yA) e (xB, yB)
36             int xC = ( xA + xB ) / 2;
37             int yC = ( yA + yB ) / 2;
38
39             // calcula o quarto ponto (xD, yD) que forma um
40             // triângulo isósceles reto entre (xA, yA) e (xC, yC)
41             // onde o ângulo direito está a (xD, yD)
42             int xD = xA + ( xC - xA ) / 2 - ( yC - yA ) / 2;
43             int yD = yA + ( yC - yA ) / 2 + ( xC - xA ) / 2;
44
45             // desenha recursivamente o Fractal
46             drawFractal( level - 1, xD, yD, xA, yA, g );
47             drawFractal( level - 1, xD, yD, xC, yC, g );
48             drawFractal( level - 1, xD, yD, xB, yB, g );
49          } // fim de else
50       } // fim do método drawFractal
51
52       // começa a desenhar o fractal
53       public void paintComponent( Graphics g )
54       {
55          super.paintComponent( g );
56
57          // desenha o padrão de fractal
58          g.setColor( color );
59          drawFractal( level, 100, 90, 290, 200, g );
60       } // fim do método paintComponent
61
62       // configura a cor de desenho como c
63       public void setColor( Color c )
64       {
65          color = c;
66       } // fim do método setColor
67
68       // configura o novo nível de recursão
69       public void setLevel( int currentLevel )
70       {
71          level = currentLevel;
72       } // fim do método setLevel
73
74       // retorna o nível de recursão
75       public int getLevel()
76       {
77          return level;
78       } // fim do método getLevel
79    } // fim da classe FractalJPanel
```





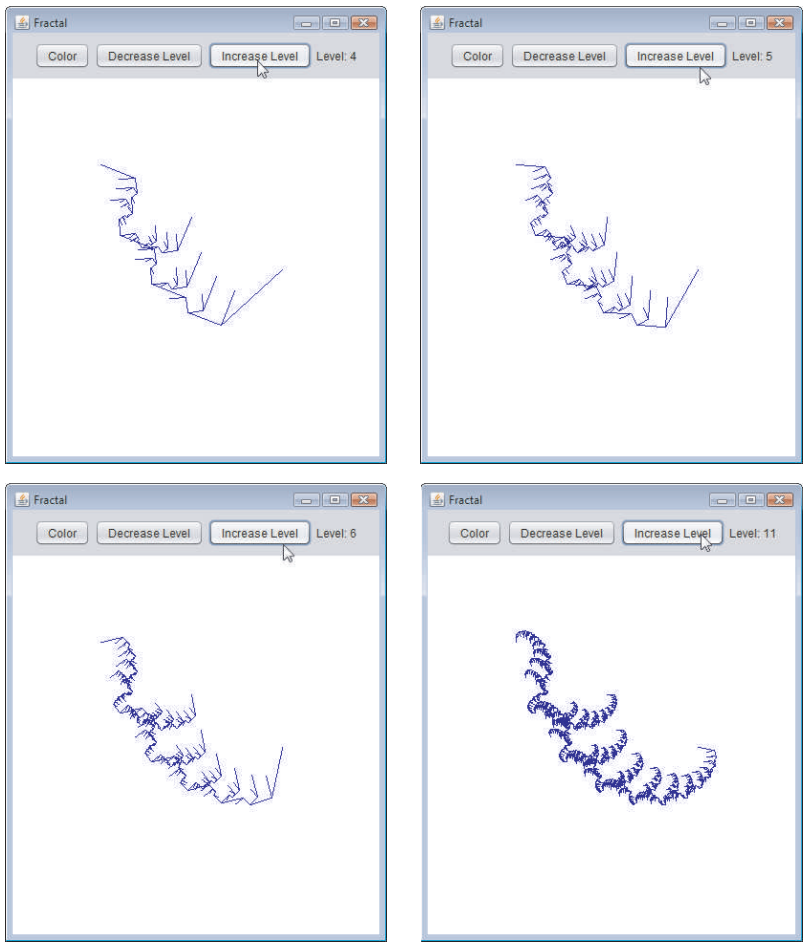


Figura 18.19 | Desenhando o "fractal de Lo" com a recursão.

As linhas 27–50 definem o método recursivo que cria o fractal. Esse método aceita seis parâmetros: o nível, quatro inteiros que especificam as coordenadas *x* e *y* de dois pontos e o objeto `Graphics` `g`. O caso básico para esse método (linha 31) ocorre quando `level` é igual a 0, momento em que uma linha será desenhada entre os dois pontos dados como parâmetros. As linhas 36–43 calculam (`xC`, `yC`), o ponto intermediário entre (`xA`, `yA`) e (`xB`, `yB`) e (`xD`, `yD`), o ponto que cria um triângulo isósceles reto com (`xA`, `yA`) e (`xC`, `yC`). As linhas 46–48 fazem três chamadas recursivas em três diferentes conjuntos de pontos.

No método `paintComponent`, a linha 59 faz a primeira chamada para o método `drawFractal` começar a desenhar. Essa chamada de método não é recursiva, mas todas as chamadas subsequentes para `drawFractal` realizadas a partir do corpo de `drawFractal` são. Visto que as linhas não serão desenhadas até que o caso básico seja alcançado, a distância entre os dois pontos diminui a cada chamada recursiva. Como o nível de recursão aumenta, o fractal torna-se mais suave e mais detalhado. A forma desse fractal estabiliza-se à medida que o nível aproxima-se de 11. Os fractais se estabilizarão em diferentes níveis com base na forma e tamanho do fractal.

A Figura 18.19 mostra o desenvolvimento do fractal de nível 0 a 6. A última imagem mostra a forma de definição do fractal no nível 11. Se focalizarmos um dos braços desse fractal, ele será idêntico à imagem inteira. Essa propriedade define o fractal como estritamente autossimilar. Consulte nosso Recursion Resource Center em `www.deitel.com` para recursos adicionais em fractal.

18.9 Retorno recursivo

Todos os nossos métodos recursivos têm uma arquitetura semelhante — se o caso básico for alcançado, retorna um resultado; se não for, faz uma ou mais chamadas recursivas. Esta seção explora uma técnica recursiva mais complexa que encontra um caminho por um labirinto, retornando verdadeira se houver uma possível solução para o labirinto. A solução envolve mover um passo por vez pelo labirinto, onde os movimentos podem ser feitos para baixo, para a direita, para cima ou para a esquerda (movimentos diagonais não são permitidos). A partir da localização atual no labirinto (iniciando com o ponto de entrada), os seguintes passos são dados: Para cada direção possível, o movimento é feito nessa direção e uma chamada recursiva é feita para resolver o restante do labirinto a partir da nova localização. Quando alcançamos um caminho sem saída (isto é, não podemos mais avançar sem se deparar com uma parede), voltamos à localização anterior e tentamos pegar uma direção diferente. Se não houver nenhuma outra direção, voltamos ao ponto de partida. Esse processo continua até encontrarmos um ponto no labirinto onde um movimento *possa* ser feito em outra direção. Uma vez que essa localização é encontrada, vamos para a nova direção e continuamos com outra chamada recursiva para resolver o restante do labirinto.

Para voltar à localização anterior no labirinto, nosso método recursivo simplesmente retorna falso, subindo a cadeia de chamadas de método até a chamada recursiva anterior (que referencia a localização anterior no labirinto). O uso da recursão para retornar a um ponto de decisão anterior é conhecido como **reversão recursiva**. Se um conjunto de chamadas recursivas não resultar em uma solução para o problema, o programa volta ao ponto de decisão e toma uma decisão diferente, resultando frequentemente em outro conjunto de chamadas recursivas. Nesse exemplo, o ponto de decisão anterior é a localização anterior no labirinto e a decisão a ser tomada é a direção que o próximo movimento deve tomar. Uma direção resultou em um caminho sem saída, então a pesquisa continua com uma direção diferente. A solução de reversão recursiva ao problema de labirinto utiliza a recursão para retroceder apenas parte da cadeia de chamadas de método e, então, tentar uma direção diferente. Se a reversão alcançar a localização de entrada do labirinto e os caminhos em todas as direções forem tentados, o labirinto não tem uma solução.

Os exercícios solicitam que você implemente soluções da reversão recursiva para o problema do labirinto (exercícios 18.20–18.22) e para o problema das Oito Rainhas (Exercício 18.15), que tenta encontrar uma forma de colocar oito rainhas em um tabuleiro vazio, de modo que nenhuma rainha esteja "atacando" qualquer outra rainha (isto é, duas rainhas não ficam na mesma linha, na mesma coluna ou na mesma diagonal). Consulte nosso Recursion Resource Center em `www.deitel.com/` para obter mais informações sobre reversão recursiva (recursive backtracking).

18.10 Conclusão

Neste capítulo, você aprendeu a criar métodos recursivos — isto é, os métodos que chamam a si próprios. Você aprendeu que os métodos recursivos em geral dividem um problema em duas partes conceituais — uma parte que o método sabe fazer (o caso básico) e, outra, que o método não sabe fazer (o passo de recursão). O passo de recursão é uma versão ligeiramente menor do problema original e é realizada por uma chamada de método recursivo. Você viu alguns exemplos populares de recursão, incluindo exemplos de como calcular fatoriais e produzir valores na série de Fibonacci. Você, então, aprendeu o funcionamento da recursão "sob o capô", incluindo a ordem em que as chamadas de método recursivo são adicionadas ou removidas da pilha de execução do programa. Depois, você comparou abordagens recursivas e iterativas (não recursivas). Aprendeu a resolver problemas mais complexos utilizando a recursão — as Torres do Hanói e a exibição de fractal. O capítulo concluiu com uma introdução à reversão recursiva, uma técnica para resolução de problemas que envolve voltar por meio de chamadas recursivas para tentar possíveis soluções diferentes. No próximo capítulo, você aprenderá numerosas técnicas para classificar listas de dados e pesquisar um item em uma lista de dados e explorará as circunstâncias sob as quais cada técnica de pesquisa e classificação deve ser utilizada.

Resumo

Seção 18.1 Introdução

- Um método recursivo chama a si próprio direta ou indiretamente por outro método.

Seção 18.2 Conceitos de recursão

- Quando um método recursivo é chamado para resolver um problema, na verdade o método só é capaz de resolver o(s) caso(s) mais simples(s), ou caso(s) de base. Se for chamado com um caso básico, o método retorna um resultado.
- Se um método recursivo é chamado com um problema mais complexo do que o de um caso básico, ele geralmente divide o problema em duas partes conceituais — uma parte que o método sabe como fazer e uma parte que ele não sabe.
- Para tornar a recursão realizável, a parte que o método não sabe fazer deve ser semelhante ao problema original, mas ser uma versão ligeiramente mais simples ou menor dele. Como esse novo problema se parece com o problema original, o método chama uma cópia nova de si próprio para trabalhar no problema menor — isso é chamado de passo de recursão.
- Para a recursão por fim terminar, toda vez que um método chamar a si próprio com uma versão mais simples do problema original, a sequência de problemas cada vez menores deve convergir para um caso básico. Quando o método reconhece o caso básico, ele retorna um resultado para a cópia anterior do método.
- Uma chamada recursiva pode ser uma chamada para outro método, que por sua vez faz uma chamada de volta ao método original. Esse processo ainda resulta em uma chamada recursiva ao método original. Isso é conhecido como uma chamada recursiva indireta ou recursão indireta.

Seção 18.3 Exemplo que utiliza recursão: fatoriais

- Omitir o caso básico ou escrever o passo de recursão incorretamente de modo que ele não convirja para o caso básico pode causar recursão infinita, esgotando por fim a memória. Isso é análogo ao problema de um loop infinito em uma solução iterativa (não recursiva).

Seção 18.4 Exemplo que utiliza recursão: série de Fibonacci

- A série de Fibonacci inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois anteriores.
- A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618…, um número que é chamado de relação áurea ou média áurea.
- Algumas soluções recursivas, como Fibonacci, resultam em uma "explosão" de chamadas de método.

Seção 18.5 Recursão e a pilha de chamadas de método

- O método de execução é sempre aquele cujo registro de ativação está na parte superior da pilha, e o registro de ativação desse método contém os valores das suas variáveis locais.

Seção 18.6 Recursão vs. iteração

- Tanto iteração como recursão se baseiam em uma estrutura de controle: a iteração utiliza uma instrução de repetição, recursão e uma instrução de seleção.
- Ambas envolvem repetição: a iteração utiliza explicitamente uma instrução de repetição, enquanto a recursão alcança a repetição por meio de repetidas chamadas de método.
- Tanto uma como outra envolvem um teste de terminação: A iteração termina quando a condição de continuação do loop falha e a recursão, quando um caso básico é reconhecido.
- A iteração com repetição controlada por contador e a recursão gradualmente se aproximam do término: a iteração continua modificando um contador até que o contador assume um valor que faz a condição de continuação do loop falhar; a recursão continua produzindo versões mais simples do problema original até que o caso básico seja alcançado.
- Tanto uma como outra podem ocorrer infinitamente: um loop infinito ocorre com a iteração se o teste de continuação do loop nunca se tornar falso, enquanto a recursão infinita ocorre se o passo de recursão não reduz o problema cada vez de uma maneira que convirja para o caso básico.
- A recursão invoca repetidamente o mecanismo e, consequentemente, o overhead das chamadas de método.
- Qualquer problema que possa ser resolvido recursivamente também pode ser resolvido iterativamente.
- Uma abordagem recursiva normalmente é preferida sobre uma abordagem iterativa quando ela espelha mais naturalmente o problema e resulta em um programa mais fácil de entender e depurar.
- Uma abordagem recursiva pode ser frequentemente implementada com poucas linhas de código, mas uma abordagem iterativa correspondente poderia exigir uma grande quantidade de código. Outra razão para escolher uma solução recursiva é que uma solução iterativa poderia não ser aparente.

Seção 18.8 Fractais

- Um fractal é uma figura geométrica que é gerada de um padrão repetido recursiva e infinitamente
- Os fractais têm uma propriedade de autossimilaridade — as subpartes são cópias de tamanho reduzido do todo.

Seção 18.9 Retorno recursivo

- O uso da recursão para retornar a um ponto de decisão anterior é conhecido como reversão recursiva. Se um conjunto de chamadas recursivas não resultar em uma solução para o problema, o programa volta ao ponto de decisão e toma uma decisão diferente, resultando frequentemente em outro conjunto de chamadas recursivas.

Terminologia

`add`, método da classe `BigInteger`, 595
`BigDecimal`, classe, 593
`BigInteger`, classe, 593
caso de base, 591
chamada recursiva, 591
chamada recursiva indireta, 591
`compareTo`, método da classe `BigInteger`, 594
Fibonacci, série de, 594
fractal, 600
fractal da Curva de Koch, 601
fractal de Floco de neve de Koch, 601
fractal estritamente autossimilar, 600
média áurea, 594
método recursivo, 590
`multiply`, método da classe `BigInteger`, 594
nível de um fractal, 601
`ONE`, constante da classe `BigInteger`, 594
ordem de um fractal, 601
profundidade de um fractal, 601
propriedade autossimilar de um fractal, 600
recursão indireta, 591
recursão infinita, 593
recursão, passo, 591
relação áurea, 594
retorno recursivo, 607
reversão recursiva, 608
`subtract`, método da classe `BigInteger`, 595
Torres de Hanói, 598
`ZERO`, constante da classe `BigInteger`, 595

Exercícios de autorrevisão

18.1 Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Um método que chama a si próprio indiretamente não é um exemplo de recursão.

b) A recursão pode ser eficiente em computação por causa da utilização reduzida do espaço de memória.

c) Quando um método recursivo é chamado para resolver um problema, na verdade, ele é capaz de resolver somente o(s) caso(s) mais simples(s), ou caso(s) de base.

d) Para tornar a recursão realizável, o passo de recursão em uma solução recursiva deve ser semelhante ao do problema original, mas ser uma versão ligeiramente maior dele.

18.2 Um(a) _____ é necessário(a) para terminar a recursão.

a) passo de recursão

b) instrução `break`
c) tipo de retorno `void`
d) caso básico

18.3 A primeira chamada para invocar um método recursivo é_____.
a) não recursivo(a)
b) recursiva(o)
c) passo de recursão
d) nenhuma das alternativas acima

18.4 Toda vez que o padrão de um fractal é aplicado, diz-se que o fractal está em um(a) novo(a) _____.
a) largura
b) altura
c) nível
d) volume

18.5 Tanto a iteração como a recursão envolvem um(a) _____.
a) instrução de repetição
b) teste de terminação
c) variável de contador
d) nenhuma das alternativas acima

18.6 Preencha as lacunas em cada uma das seguintes afirmações:
a) A relação de números de Fibonacci sucessivos converge em um valor constante de 1,618…, um número que foi chamado de _____ ou _____.
b) A iteração normalmente utiliza uma instrução de repetição, enquanto a recursão normalmente utiliza uma instrução _____.
c) Os fractais têm uma propriedade _____ — quando subdivididos em partes, cada uma das partes é uma cópia de tamanho reduzido do total.

Respostas dos exercícios de autorrevisão

18.1 a) Falsa. Um método que chama a si próprio dessa maneira é um exemplo de recursão indireta. b) Falsa. A recursão pode ser ineficiente no cálculo por causa de múltiplas chamadas de método e uso de espaço de memória. c) Verdadeira. d) Falsa. Para tornar a recursão realizável, o passo de recursão em uma solução recursiva deve ser semelhante ao do problema original, mas ser uma versão ligeiramente *menor* dele.

18.2 d

18.3 a

18.4 c

18.5 b

18.6 a) relação de ouro, média áurea. b) seleção. c) autossimilaridade.

Exercícios

18.7 O que o seguinte código faz?

```
1  public int mystery( int a, int b )
2  {
3     if ( b == 1 )
4        return a;
5     else
6        return a + mystery( a, b - 1 );
7  } // fim do método mystery
```

18.8 Localize o(s) erro(s) no seguinte método recursivo e explique como corrigi-lo(s). Esse método deve localizar a soma dos valores de 0 a `n`.

```
1  public int sum( int n )
2  {
3     if ( n == 0 )
4        return 0;
5     else
6        return n + sum( n );
7  } // fim do método sum
```

18.9 (***Método power recursivo***) Escreva um método recursivo `power(base, exponent)` que, quando chamado, retorna

$base^{exponente}$

Por exemplo, `power( 3,4 )` = 3 * 3 * 3 * 3. Assuma que `exponent` é um inteiro maior que ou igual a 1. [*Dica:* o passo de recursão deve utilizar o relacionamento

$base^{exponente} = base \cdot base^{exponente - 1}$

e a condição de terminação ocorre quando `exponent` é igual a `1` porque

$base^{1} = base$

Incorpore esse método em um programa que permita que o usuário insira a `base` e o `exponent`].

18.10 (***Visualizando a recursão***) É interessante observar a recursão "em ação". Modifique o método fatorial na Figura 18.3 para imprimir sua variável local e o parâmetro de chamada recursiva. Para cada chamada recursiva, exiba as saídas em uma linha separada e adicione um nível de recuo. Faça o melhor que você puder para tornar a saída limpa, interessante e significativa. Seu objetivo aqui é projetar e implementar um formato de saída que facilite o entendimento da recursão. Você pode querer adicionar essas capacidades de exibição a outros exemplos de recursão e a exercícios por todo o texto.

18.11 (***Máximo divisor comum***) O máximo divisor comum dos inteiros `x` e `y` é o maior inteiro que é divisível por `x` e `y`. Escreva um método recursivo `gcd` que retorna o máximo divisor comum de `x` e `y`. O `gcd` de `x` e `y` é definido recursivamente como segue: se `y` for igual a `0`, então `gcd( x, y )` é `x`; caso contrário, `gcd( x, y )` é `gcd( y, x % y )`, onde `%` é o operador de resto. Utilize esse método para substituir o que você escreveu no aplicativo do Exercício 6.27.

18.12 O que o seguinte programa faz?

```
1  // Solução do Exercício 18.12: MysteryClass.java
2  public class MysteryClass
3  {
4     public static int mystery( int[] array2, int size )
5     {
6      if ( size == 1 )
7           return array2[ 0 ];
8        else
9           return array2[ size - 1 ] + mystery( array2, size - 1 );
10    } // fim do método mystery
11
12    public static void main( String[] args )
13    {
14       int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15
16       int result = mystery( array, array.length );
17       System.out.printf( "Result is: %d\n", result );
18    } // fim do método main
19 } // fim da classe MysteryClass
```

18.13 O que o seguinte programa faz?

```
1  // Solução do Exercício 18.13: SomeClass.java
2  public class SomeClass
3  {
4     public static String someMethod( int[] array2, int x )
5     {
6        if ( x < array2.length )
7           return String.format(
8              "%s%d ", someMethod( array2, x + 1 ), array2[ x ] );
9        else
10          return "";
11    } // fim do método someMethod
12
13    public static void main( String[] args )
14    {
15       int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
16       String results = someMethod( array, 0 );
17       System.out.println( results );
18    } // fim de main
19 } // fim da classe SomeClass
```

18.14 (***Palíndromos***) Um palíndromo é uma string que é lida da mesma maneira, quer da esquerda para a direita, quer da direita para a esquerda. Alguns exemplos de palíndromos são "radar", "a cara rajada da jararaca" e (se espaços forem ignorados) "a bola da loba." Escreva um método recursivo `testPalindrome` que retorna o valor `boolean true` se a string armazenada no array for um palíndromo e `false`, caso contrário. O método deve ignorar espaços e pontuação na string.

18.15 (***Oito Rainhas***) Um enigma para os fãs de xadrez é o problema das Oito Rainhas, que exige: é possível colocar oito rainhas em um tabuleiro de xadrez vazio de modo que nenhuma rainha esteja "atacando" qualquer outra (isto é, sem que duas rainhas estejam na mesma linha, na mesma coluna ou na mesma diagonal)? Por exemplo, se uma rainha for colocada no canto superior esquerdo do tabuleiro, nenhuma outra rainha pode ser colocada em qualquer um dos quadrados marcados mostrados na Figura 18.20. Resolva o problema recursivamente. [*Dica:* sua solução deve iniciar com a primeira coluna e procurar uma localização nessa coluna em que uma rainha pode ser colocada — inicialmente, coloque a rainha na primeira linha. A solução deve então pesquisar recursivamente as colunas restantes. Nas primeiras poucas colunas, há várias localizações onde uma rainha pode ser colocada. Utilize a primeira localização disponível. Se uma coluna for alcançada sem nenhuma possível localização para uma rainha, o programa deve retornar à coluna anterior e mover a rainha nessa coluna para uma nova linha. Esse contínuo procedimento de voltar e tentar novas alternativas é um exemplo de reversão recursiva].

18.16 (***Imprima um array***) Escreva um método recursivo `printArray` que exibe todos os elementos em um array de inteiros, separados por espaços.

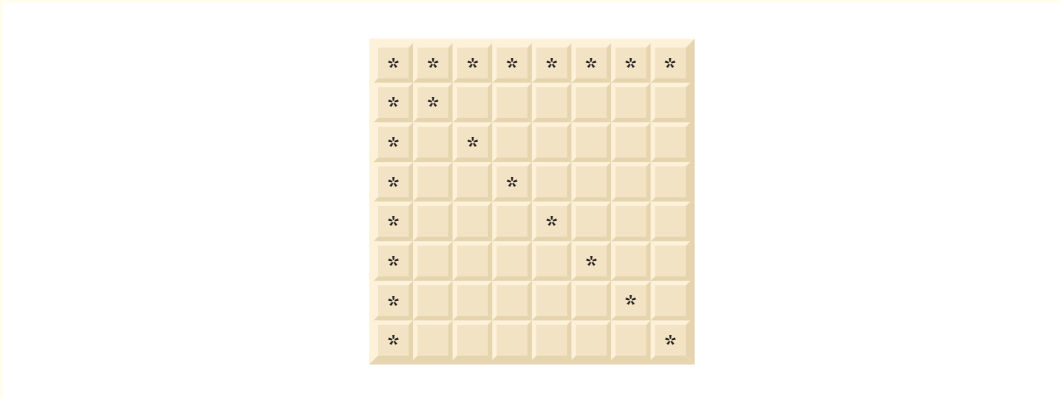


Figura 18.20 | Os quadrados eliminados pela colocação de uma rainha no canto superior esquerdo de um tabuleiro.

18.17 (***Imprima um array de trás para frente***) Escreva um método recursivo `stringReverse` que aceita um array de caracteres contendo uma string como um argumento e imprime a string de trás para frente. [*Dica:* utilize o método `String` `toCharArray`, que não aceita nenhum argumento, para obter um array `char` contendo os caracteres na `String`.]

18.18 (***Localize o valor mínimo em um array***) Escreva um método recursivo `recursiveMinimum` que determina o menor elemento em um array de inteiros. O método deve retornar quando ele receber um array de um elemento.

18.19 (***Fractais***) Repita o padrão de fractal na Seção 18.8 para formar uma estrela. Inicie com cinco linhas (ver a Figura 18.21), em vez de uma, em que cada linha é uma ponta diferente da estrela. Aplique o padrão "fractal de Lo" a cada ponta da estrela.

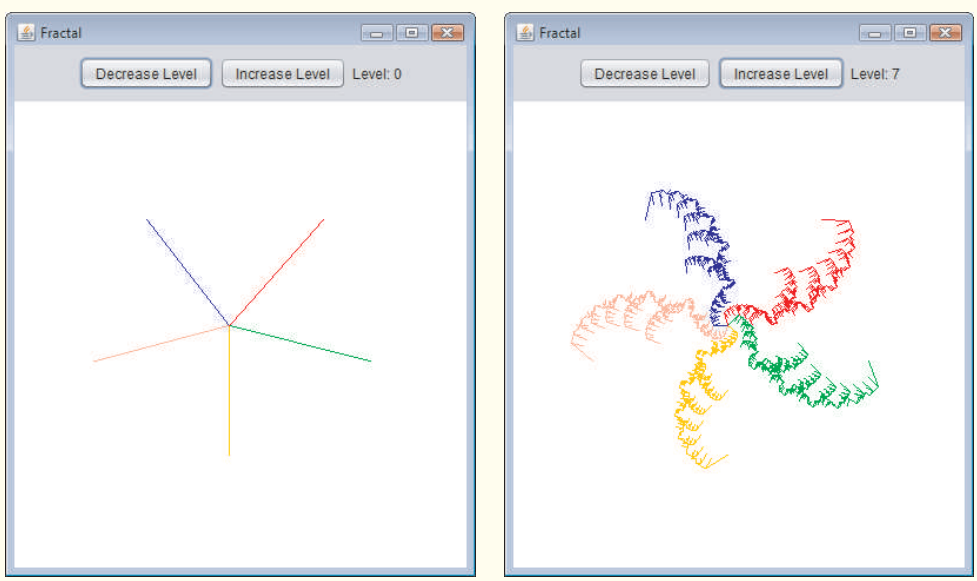


Figura 18.21 | Saídas de exemplo para o Exercício 18.19.

18.20 (***Percurso para sair de um labirinto utilizando reversão recursiva***) A grade de `#`s e pontos (`.`) na Figura 18.22 é uma representação bidimensional do array de um labirinto. Os `#`s representam as paredes do labirinto e os pontos representam as localizações nos possíveis caminhos pelo labirinto. Movimentos são permitidos apenas nas posições do array que contiverem um ponto.

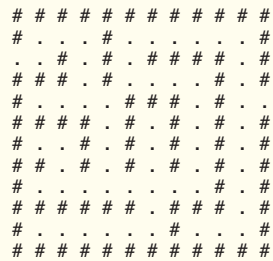


Figura 18.22 | Representação de array bidimensional de um labirinto.

Escreva um método recursivo (`mazeTraversal`) para percorrer labirintos como o da Figura 18.22. O método deve receber como argumentos um array de caractere 12 por 12 que representa o labirinto e a localização atual no labirinto (a primeira vez em que esse método for chamado, a localização atual deve ser o ponto de entrada do labirinto). À medida que `mazeTraversal` tenta localizar a saída, ele deve colocar o caractere `x` em cada quadrado no caminho. Há um algoritmo simples para percorrer um labirinto que garante a localização da saída (assumindo que exista uma saída). Se não houver nenhuma saída, você chegará à localização inicial novamente. O algoritmo é como segue: A partir da localização atual no labirinto, tente mover-se um espaço em qualquer uma das possíveis direções (para baixo, para a direita, para cima ou para a esquerda). Se for possível ir pelo menos para uma direção, chame `mazeTraversal` recursivamente, passando o novo local no labirinto como o local atual. Se não for possível ir para nenhuma direção, "retroceda" para uma localização anterior no labirinto e tente uma nova direção dessa localização. Programe o método para exibir o labirinto depois de cada movimento para que o usuário possa observar enquanto uma solução para o labirinto é procurada. A saída final do labirinto deve exibir somente o caminho necessário para resolver o labirinto — os outros caminhos não devem exibidos. [*Dica:* para exibir apenas o caminho final, pode ser útil marcar os locais que resultam em um caminho sem saída com outro caractere (como `'0'`).]

18.21 **(*Gerando labirintos aleatoriamente*)** Escreva um método `mazeGenerator` que recebe como um argumento um array bidimensional de 12 caracteres e produza aleatoriamente um labirinto. O método também deve fornecer as posições inicial e final do labirinto. Teste seu método `mazeTraversal` do Exercício 18.20, utilizando vários labirintos gerados aleatoriamente.

18.22 **(*Labirintos de qualquer tamanho*)** Generalize os métodos `mazeTraversal` e `mazeGenerator` do Exercício 18.20 e do Exercício 18.21 para processar labirintos de qualquer largura e altura.

18.23 **(*Tempo para calcular números de Fibonacci*)** Aprimore o programa de Fibonacci da Figura 18.5 para que ele calcule a quantidade aproximada de tempo necessário para efetuar o cálculo e o número de chamadas feitas para o método recursivo. Para esse fim, chame o método `static` `System` `currentTimeMillis`, que não aceita nenhum argumento e retorna a hora atual do computador em milissegundos. Chame esse método duas vezes — uma vez antes e uma depois da chamada para `fibonacci`. Salve cada valor e calcule a diferença em horas para determinar quantos milissegundos foram necessários para efetuar o cálculo. Então, adicione uma variável à classe `FibonacciCalculator` e utilize essa variável para determinar o número de chamadas feitas para o método `fibonacci`. Exiba seus resultados.

19

Entre lágrimas e soluços,
escolheu as de maior tamanho …
— **Lewis Carroll**

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
[Tente o último e nunca duvides;
Nada é tão difícil, mas a pesquisa o encontrará.]
— **Robert Herrick**

Guardei-o na memória,
e a chave a levas.
— **William Shakespeare**

É uma lei imutável nos negócios que palavras são palavras, explicações são explicações, promessas são promessas — mas somente desempenho é realidade.
— **Harold S. Green**

Pesquisa, classificação e Big O

Objetivos

Neste capítulo, você aprenderá:

- A procurar um dado valor em um array utilizando a pesquisa linear e pesquisa binária.
- A classificar arrays utilizando a seleção iterativa e algoritmos de classificação por inserção.
- A classificar arrays utilizando o algoritmo recursivo de classificação por intercalação.
- A determinar a eficiência dos algoritmos de pesquisa e de classificação.

Sumário

19.1 Introdução
19.2 Algoritmos de pesquisa
19.2.1 Pesquisa linear
19.2.2 Pesquisa binária
19.3 Algoritmos de classificação
19.3.1 Classificação por seleção
19.3.2 Classificação por inserção
19.3.3 Classificação por intercalação
19.4 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

19.1 Introdução

A **pesquisa** de dados envolve determinar se um valor (chamado **chave de pesquisa**) está presente nos dados e, se estiver, encontrar a sua localização. Dois algoritmos de pesquisa famosos são a pesquisa linear simples e a mais rápida, porém mais complexa, pesquisa binária. A **classificação** coloca os dados em uma ordem crescente ou decrescente, com base em uma ou mais **chaves de classificação**. Uma lista de nomes poderia ser classificada alfabeticamente, contas bancárias poderiam ser classificadas pelo número de conta, registros de folha de pagamento de funcionários poderiam ser classificados pelo CPF e assim por diante. Este capítulo introduz dois algoritmos simples de classificação; a classificação por seleção e a classificação por inserção, juntamente com a classificação por intercalação mais eficiente, porém mais complexa. A Figura 19.1 resume os algoritmos de pesquisa e classificação discutidos nos exemplos e exercícios deste livro.

Capítulo	Algoritmo	Posição
Algoritmos de pesquisa:		
19	Pesquisa linear	Seção 19.2.1
	Pesquisa binária	Seção 19.2.2
	Pesquisa linear recursiva	Exercícios 19.8
	Pesquisa binária recursiva	Exercícios 19.9
20	Método `binarySearch` da classe `Collections`	Figura 20.12
22	Pesquisa linear em uma `List`	Exercícios 22.21
	Pesquisa em árvore binária	Exercícios 22.23
Algoritmos de classificação:		
19	Classificação por seleção	Seção 19.3.1
	Classificação por inserção	Seção 19.3.2
	Classificação por intercalação recursiva	Seção 19.3.3
	Classificação por flutuação	Exercícios 19.5 e 19.6
	Classificação de bucket	Exercícios 19.7
	Quicksort recursivo	Exercícios 19.10
20	Método `sort` da classe `Collections`	Figuras 20.6 a 20.9
	Coleção `SortedSet`	Figura 20.17
22	Classificação em árvore binária	Seção 22.7

Figura 19.1 | Algoritmos de pesquisa e classificação neste texto.

19.2 Algoritmos de pesquisa

Pesquisar um número de telefone, localizar um site Web via um mecanismo de pesquisa e verificar a definição de uma palavra em um dicionário são operações que envolvem pesquisar grandes volumes de dados. As duas seções a seguir discutem dois algoritmos de pesquisa comuns — um fácil de programar, mas relativamente ineficiente e outro relativamente eficiente, mas mais complexo e difícil de programar.

19.2.1 Pesquisa linear

O **algoritmo de pesquisa linear** pesquisa cada elemento em um array sequencialmente. Se a chave de pesquisa não corresponder a um elemento no array, o algoritmo testa cada elemento e, quando alcança o fim do array, informa o usuário que a chave de pesquisa não está presente. Se a chave de pesquisa estiver no array, o algoritmo testa cada elemento até encontrar um que corresponda à chave de pesquisa e retorna o índice desse elemento.

Como um exemplo, considere um array que contém os valores a seguir

```
34   56   2   10   77   51   93   30   5   52
```

e um programa que pesquisa 51. Utilizando o algoritmo de pesquisa linear, o programa primeiro verifica se 34 corresponde à chave de pesquisa. Se não corresponder, o algoritmo então verifica se 56 corresponde à chave de pesquisa. O programa continua a percorrer o array sequencialmente, testando 2, 10 e então 77. Quando o programa testa 51, que corresponde à chave de pesquisa, o programa retorna o índice 5, que é a localização do 51 no array. Se, depois de verificar cada elemento do array, o programa determinar que a chave de pesquisa não corresponde a nenhum elemento no array, ele retorna um valor de sentinela (por exemplo, `-1`).

A Figura 19.2 declara a classe `LinearArray`. Essa classe tem duas variáveis de instância `private` — um array de `int`s chamado `data` e um objeto `static` `Random` para preencher o array com `int`s aleatoriamente gerados. Quando um objeto da classe `LinearArray` é instanciado, o construtor (linhas 13–20) cria e inicializa o array `data` com `int`s aleatórios no intervalo de `10–99`. Se houver valores duplicados no array, a pesquisa linear retorna o índice do primeiro elemento no array que corresponde à chave de pesquisa.

```
1  // Figura 19.2: LinearArray.java
2  // Classe que contém um array de inteiros aleatórios e um método
3  // que pesquisará esse array sequencialmente
4  import java.util.Random;
5  import java.util.Arrays;
6
7  public class LinearArray
8  {
9     private int[] data; // array de valores
10    private static final Random generator = new Random();
11
12    // cria um array de um dado tamanho e o preenche com números aleatórios
13    public LinearArray( int size )
14    {
15       data = new int[ size ]; // cria espaço para o array
16
17       // preenche o array com ints aleatórios no intervalo de 10-99
18       for ( int i = 0; i < size; i++ )
19          data[ i ] = 10 + generator.nextInt( 90 );
20    } // fim do construtor de LinearArray
21
22    // realiza uma pesquisa linear nos dados
23    public int linearSearch( int searchKey )
24    {
25       // faz um loop pelo array sequencialmente
26       for ( int index = 0; index < data.length; index++ )
27          if ( data[ index ] == searchKey )
28             return index; // retorna o índice de inteiros
29
30       return -1; // inteiro não foi localizado
31    } // fim do método linearSearch
32
33    // método para gerar saída de valores no array
34    public String toString()
35    {
36       return Arrays.toString( data );
37    } // fim do método toString
38 } // fim da classe LinearArray
```

Figura 19.2 | A classe que contém um array de números inteiros aleatórios e um método que pesquisará esse array sequencialmente.

As linhas 23–31 realizam a pesquisa linear. A chave de pesquisa é passada para o parâmetro `searchKey`. As linhas 26–28 fazem um loop pelos elementos no array. A linha 27 compara cada um com `searchKey`. Se os valores forem iguais, a linha 28 retornará o índice do elemento. Se o loop terminar sem encontrar o valor, a linha 30 retornará `-1`. As linhas 34–37 declaram o método `toString`, que retorna

uma representação de `String` do array para impressão. Utilizamos o método `toString` `Arrays` `static` para retornar uma representação `String` do array.

A Figura 19.3 cria um objeto `LinearArray` contendo um array de `10` `int`s (linha 16) e permite que o usuário pesquise no array. As linhas 20–22 solicitam ao usuário a chave de pesquisa e a armazenam. O array contém `int`s entre `10`–`99` (linha 18 da Figura 19.2). A linha 28 chama o método `linearSearch` para determinar se `searchInt` está no array. Se não estiver, `linearSearch` retornará `-1` e o programa notificará o usuário (linhas 31–32). Se `searchInt` estiver no array, `linearSearch` retorna a posição do elemento, para o qual o programa gera a saída nas linhas 34–35. As linhas 38–40 recuperam o próximo inteiro do usuário.

```
1  // Figura 19.3: LinearSearchTest.java
2  // Pesquisando sequencialmente um item em um array.
3  import java.util.Scanner;
4
5  public class LinearSearchTest
6  {
7     public static void main( String[] args )
8     {
9        // cria o objeto Scanner para inserir dados
10       Scanner input = new Scanner( System.in );
11
12       int searchInt; // chave de pesquisa
13       int position; // localização da chave de pesquisa no array
14
15       // cria um array e gera saída
16       LinearArray searchArray = new LinearArray( 10 );
17       System.out.println( searchArray + "\n" ); // imprime o array
18
19       // obtém a entrada de usuário
20       System.out.print(
21          "Please enter an integer value (-1 to quit): " );
22       searchInt = input.nextInt(); // lê o primeiro int de usuário
23
24       // insere repetidamente um inteiro; -1 termina o programa
25       while ( searchInt != -1 )
26       {
27          // realiza a pesquisa linear
28          position = searchArray.linearSearch( searchInt );
29
30          if ( position == -1 ) // inteiro não foi localizado
31             System.out.println( "The integer " + searchInt +
32                " was not found.\n" );
33          else // inteiro foi localizado
34             System.out.println( "The integer " + searchInt +
35                " was found in position " + position + ".\n" );
36
37          // obtém a entrada de usuário
38          System.out.print(
39             "Please enter an integer value (-1 to quit): " );
40          searchInt = input.nextInt(); // lê o próximo int de usuário
41       } // fim do while
42    } // fim de main
43 } // fim da classe LinearSearchTest
```

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]

Please enter an integer value (-1 to quit): 79
The integer 79 was found in position 4.

Please enter an integer value (-1 to quit): 61
The integer 61 was not found.

Please enter an integer value (-1 to quit): 51
The integer 51 was found in position 7.

Please enter an integer value (-1 to quit): -1
```

Figura 19.3 | Pesquisando sequencialmente um item em um array.

Eficiência da pesquisa linear

Todos os algoritmos de pesquisa têm o mesmo objetivo — localizar um elemento que corresponde a uma dada chave de pesquisa, se esse elemento, de fato, existir. Há, porém, alguns aspectos que diferenciam os algoritmos de pesquisa uns dos outros. A principal diferença é o esforço que eles exigem para completar a pesquisa. Uma maneira de descrever esse esforço é com a **notação Big O**, que indica o pior cenário de tempo de execução para um algoritmo, isto é, a dificuldade que um algoritmo terá para resolver um problema. Para algoritmos de pesquisa e de classificação, isso depende particularmente de quantos elementos de dados há.

Suponha que um algoritmo seja projetado para testar se o primeiro elemento de um array é igual ao segundo. Se o array tiver 10 elementos, esse algoritmo exigirá uma comparação. Se o array tiver 1.000 elementos, ele ainda exigirá uma comparação. Na realidade, o algoritmo é completamente independente do número de elementos no array. Diz-se que esse algoritmo tem um **tempo de execução constante**, representado na notação Big O como $O(1)$. Um algoritmo que é $O(1)$ não necessariamente exige somente uma comparação. $O(1)$ simplesmente significa que o número de comparações é *constante* — não aumenta à medida que o tamanho do array aumenta. Um algoritmo que testa se o primeiro elemento de um array é igual a qualquer um dos três próximos elementos ainda é $O(1)$ mesmo se exigir três comparações.

Um algoritmo que testa se o primeiro elemento do array é igual a *qualquer* um dos outros elementos do array exigirá no máximo comparações $n - 1$ comparações, onde n é o número de elementos do array. Se o array tiver 10 elementos, esse algoritmo exigirá até nove comparações. Se o array tiver 1.000 elementos, ele exigirá até 999 comparações. À medida que n aumenta, a parte n da expressão "predomina", e subtrair uma torna-se irrelevante. A notação Big O é projetada para destacar esses termos dominantes e ignorar termos que não têm importância à medida que n aumenta. Por essa razão, diz-se que um algoritmo que exige um total de $n - 1$ comparações (como aquele que descrevemos anteriormente) é $O(n)$. Diz-se que um algoritmo $O(n)$ tem um **tempo de execução linear**. $O(n)$ costuma ser pronunciado "na ordem de n" ou, simplesmente, "ordem n".

Agora suponha que você tem um algoritmo que testa se *qualquer* elemento de um array é duplicado em outra parte no array. O primeiro elemento deve ser comparado com elementos alternados no array. O segundo elemento deve ser comparado com elementos alternados, exceto o primeiro (já foi comparado com o primeiro). O terceiro elemento deve ser comparado com elementos alternados, exceto os dois primeiros. No final, esse algoritmo terminará fazendo $(n - 1) + (n - 2) + \ldots + 2 + 1$ ou $n^2/2 - n/2$ comparações. À medida que n aumenta, o termo n^2 predomina e o termo n torna-se irrelevante. Mais uma vez, a notação Big O destaca o termo n^2, deixando $n^2/2$. Mas, como veremos a seguir, fatores constantes são omitidos na notação Big O.

A notação Big O se preocupa como o tempo de execução de um algoritmo aumenta em relação ao número de itens processado. Suponha que um algoritmo exija n^2 comparações. Com quatro elementos, o algoritmo requer 16 comparações; com oito elementos, 64 comparações. Com esse algoritmo, dobrar o número de elementos quadruplica o número de comparações. Considere um algoritmo semelhante que exige $n^2/2$ comparações. Com quatro elementos, o algoritmo requer oito comparações; com oito elementos, 32 comparações. Mais uma vez, dobrar o número de elementos quadruplica o número de comparações. Esses dois algoritmos aumentam conforme o quadrado de n, assim o Big O ignora a constante e os dois algoritmos são considerados como $O(n^2)$, o que é chamado **tempo de execução quadrático**, pronunciado como "na ordem de n ao quadrado" ou, mais simplesmente, "ordem do quadrado de n".

Quando n *é* pequeno, algoritmos $O(n^2)$ (executando nos computadores de hoje em dia) não afetarão visivelmente o desempenho. Mas, à medida que n aumentar, você começará a observar a degradação de desempenho. Um algoritmo $O(n^2)$ executando em um array de um milhão de elementos exigiria um trilhão de "operações" (em que cada uma, na verdade, exigiria várias instruções de máquina para executar). Isso pode demorar várias horas. Um array de um bilhão de elementos exigiria um quintilhão de operações, um número tão grande que o algoritmo demoraria décadas para executar! Infelizmente, algoritmos $O(n^2)$ são fáceis de escrever, como você verá neste capítulo. Também verá algoritmos com medidas Big O mais favoráveis. Frequentemente, esses eficientes algoritmos demandam um pouco mais de perícia e trabalho para serem criados, mas seu desempenho superior recompensa muito bem o esforço extra, especialmente à medida que o n torna-se grande e algoritmos são combinados em programas maiores.

O algoritmo de pesquisa linear executa a $O(n)$ vezes. O pior caso nesse algoritmo é que cada elemento deve ser verificado para determinar se o item de pesquisa existe no array. Se o tamanho do array for dobrado, o número de comparações que o algoritmo deve realizar também será dobrado. Observe que a pesquisa linear pode fornecer um excelente desempenho se o elemento que corresponde à chave de pesquisa estiver próximo ou no início do array. Mas buscamos algoritmos que, em média, tenham um bom desempenho em todas as pesquisas, incluindo aqueles em que o elemento que corresponde à chave de pesquisa está próximo do fim do array.

A pesquisa linear é fácil de programar, mas pode ser lenta se comparada com outros algoritmos de pesquisa. Se um programa precisar realizar muitas pesquisas em grandes arrays, é melhor implementar um algoritmo mais eficiente, como a pesquisa binária que apresentaremos a seguir.

Dica de desempenho 19.1

Às vezes os algoritmos mais simples demonstram um pobre desempenho. Sua virtude é que são fáceis de programar, testar e depurar. Às vezes, algoritmos mais complexos são necessários para conseguir desempenho máximo.

19.2.2 Pesquisa binária

O **algoritmo de pesquisa binária** é mais eficiente que o de pesquisa linear, mas exige que o array seja classificado. A primeira iteração desse algoritmo testa o elemento no meio do array. Se isso corresponder à chave de pesquisa, o algoritmo termina. Supondo que o array seja classificado em ordem crescente, se a chave de pesquisa for menor que o elemento do meio, ela não poderá localizar nenhum elemento na segunda metade do array e o algoritmo continua com apenas a primeira metade do array (isto é, até o primeiro elemento, mas sem incluir o elemento do meio). Se a chave de pesquisa for maior que o elemento no meio, ela não poderá localizar nenhum elemento na primeira metade do array e o algoritmo continua apenas com a segunda metade (isto é, o elemento depois do elemento do meio até o último elemento). Cada iteração testa o valor do meio da parte restante do array. Se a chave de pesquisa não corresponder ao elemento, o algoritmo eliminará metade dos elementos restantes. O algoritmo termina localizando um elemento que corresponde à chave de pesquisa ou reduzindo o subarray ao tamanho zero.

Como um exemplo, considere o array de 15 elementos classificado

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

e uma chave de pesquisa de 65. Um programa que implementa o algoritmo de pesquisa binária primeiro verificaria se 51 é a chave de pesquisa (uma vez que 51 é o elemento no meio do array). A chave de pesquisa (65) é maior que 51, assim 51 é descartado com a primeira metade do array (todos os elementos menores que 51) deixando

```
56  65  77  81  82  93  99
```

Em seguida, o algoritmo verifica se 81 (o elemento no meio do restante do array) corresponde à chave de pesquisa. A chave de pesquisa (65) é menor que 81, portanto 81 é descartado com os elementos maiores que 81. Depois de apenas dois testes, o algoritmo reduziu a três o número de valores a verificar (56, 65 e 77). Ele, então, verifica 65 (que de fato corresponde à chave de pesquisa) e retorna o índice do elemento no array que contém 65. Esse algoritmo exigiu apenas três comparações para determinar se a chave de pesquisa localizou um elemento do array. Utilizando um algoritmo de pesquisa linear exigiria 10 comparações. [*Nota:* neste exemplo, optamos por utilizar um array com 15 elementos, de modo que sempre haverá um elemento óbvio no meio do array. Com um número par de elementos, o meio do array reside entre dois elementos. Implementamos o algoritmo para escolher o menor desses dois elementos.]

Implementação de pesquisa binária

A Figura 19.4 declara a classe `BinaryArray`. Essa classe é semelhante a `LinearArray` — ela tem duas variáveis de instância `private`, um construtor, um método de pesquisa (`binarySearch`), um método `remainingElements` e um método `toString`. As linhas 13–22 declaram o construtor. Depois de o array ser inicializado com `int`s aleatórios de `10-99` (linhas 18–19), a linha 21 chamada o método `Arrays.sort` no array `data`. O **método `sort`** é um método `static` da classe `Arrays` que classifica os elementos em um array em ordem crescente por padrão; uma versão sobrecarregada desse método permite alterar a ordem de classificação. Lembre-se de que o algoritmo de pesquisa binária só funcionará em um array classificado.

```
1  // Figura 19.4: BinaryArray.java
2  // Classe que contém um array de inteiros aleatórios e um método
3  // que utiliza a pesquisa binária para localizar um inteiro.
4  import java.util.Random;
5  import java.util.Arrays;
6
7  public class BinaryArray
8  {
9     private int[] data; // array de valores
10    private static final Random generator = new Random();
11
12    // cria um array de um dado tamanho e o preenche com inteiros aleatórios
13    public BinaryArray( int size )
14    {
15       data = new int[ size ]; // cria espaço para o array
16
17       // preenche o array com ints aleatórios no intervalo de 10-99
18       for ( int i = 0; i < size; i++ )
19          data[ i ] = 10 + generator.nextInt( 90 );
20
21       Arrays.sort( data );
22    } // fim do construtor de BinaryArray
23
24    // realiza uma pesquisa binária sobre os dados
25    public int binarySearch( int searchElement )
26    {
27       int low = 0; // extremidade baixa da área de pesquisa
```

```
28          int high = data.length - 1; // extremidade alta da área de pesquisa
29          int middle = ( low + high + 1 ) / 2; // elemento do meio
30          int location = -1; // valor de retorno; -1 se não localizado
31
32          do // faz um loop para procurar o elemento
33          {
34             // imprime os elementos remanescentes do array
35             System.out.print( remainingElements( low, high ) );
36
37             // gera espaços para alinhamento
38             for ( int i = 0; i < middle; i++ )
39                System.out.print( "   " );
40             System.out.println( " * " ); // indica o meio atual
41
42             // se o elemento for localizado no meio
43             if ( searchElement == data[ middle ] )
44                location = middle; // a localização é o meio atual
45
46             // elemento do meio é muito alto
47             else if ( searchElement < data[ middle ] )
48                high = middle - 1; // elimina a metade mais alta
49             else  // elemento do meio é muito baixo
50                low = middle + 1; // elimina a metade mais alta
51
52             middle = ( low + high + 1 ) / 2; // recalcula o meio
53          } while ( ( low <= high ) && ( location == -1 ) );
54
55          return location; // retorna a localização da chave de pesquisa
56       } // fim do método binarySearch
57
58       // método para gerar saída de certos valores no array
59       public String remainingElements( int low, int high )
60       {
61          StringBuilder temporary = new StringBuilder();
62
63          // gera espaços para alinhamento
64          for ( int i = 0; i < low; i++ )
65             temporary.append( "   " );
66
67          // gera a saída dos elementos que permanecem no array
68          for ( int i = low; i <= high; i++ )
69             temporary.append( data[ i ] + " " );
70
71          temporary.append( "\n" );
72          return temporary.toString();
73       } // remainingElements de método de fim
74
75       // método para gerar saída de valores no array
76       public String toString()
77       {
78          return remainingElements( 0, data.length - 1 );
79       } // fim do método toString
80    } // fim do método BinaryArray
```

Figura 19.4 | Classe que contém um array de números inteiros aleatórios e um método que utiliza a pesquisa binária para localizar um número inteiro.

As linhas 25–56 declaram o método `binarySearch`. A chave de pesquisa é passada para o parâmetro `searchElement` (linha 25). As linhas 27–29 calculam o índice da extremidade `low`, o índice da extremidade `high` índice `middle` da parte do array em que o programa está atualmente pesquisando. No início do método, a extremidade `low` é `0`, a extremidade `high` é o comprimento do array menos `1` e `middle` é a média desses dois valores. A linha 30 inicializa a `location` do elemento para `-1` — o valor que será retornado se o elemento não for localizado. As linhas 32–53 fazem um loop até que `low` seja maior que `high` (isso ocorre quando o elemento não é localizado) ou a `location` não é igual `-1` (indicando que a chave de pesquisa foi localizada). A linha 43 testa se o valor no elemento `middle` é igual a `searchElement`. Se isso for `true`, a linha 44 atribui `middle` a `location`. Então, o loop termina e `location` é retornada ao chamador. Cada iteração do loop testa um único valor (linha 43) e elimina metade dos valores restantes no array (linha 48 ou 50).

A Figura 19.5 realiza uma pesquisa binária usando a chave de pesquisa que o usuário insere para determinar se ela corresponde a um elemento no array. A primeira linha da saída desse programa é o array de `int`s, na ordem crescente. Quando o usuário instrui o

programa a procurar 23, o programa primeiro testa o elemento intermediário, que é 42 (como indicado por *). A chave de pesquisa é menor que 42, assim o programa elimina a segunda metade do array e testa o elemento do meio na primeira metade. A chave de pesquisa é menor que 34, portanto o programa elimina a segunda metade do array, deixando somente três elementos. Por fim, o programa verifica 23 (que corresponde à chave de pesquisa) e retorna o índice 1.

```
1  // Figura 19.5: BinarySearchTest.java
2  // Utiliza a pesquisa binária para localizar um item em um array.
3  import java.util.Scanner;
4
5  public class BinarySearchTest
6  {
7     public static void main( String[] args )
8     {
9        // cria o objeto Scanner para inserir dados
10       Scanner input = new Scanner( System.in );
11
12       int searchInt; // chave de pesquisa
13       int position; // localização da chave de pesquisa no array
14
15       // cria um array e gera saída
16       BinaryArray searchArray = new BinaryArray( 15 );
17       System.out.println( searchArray );
18
19       // obtém a entrada de usuário
20       System.out.print(
21          "Please enter an integer value (-1 to quit): " );
22       searchInt = input.nextInt(); // lê um int do usuário
23       System.out.println();
24
25       // insere repetidamente um inteiro; -1 termina o programa
26       while ( searchInt != -1 )
27       {
28          // utiliza a pesquisa binária para tentar localizar o inteiro
29          position = searchArray.binarySearch( searchInt );
30
31          // valor de retorno de -1 indica que o inteiro não foi localizado
32          if ( position == -1 )
33             System.out.println( "The integer " + searchInt +
34                " was not found.\n" );
35          else
36             System.out.println( "The integer " + searchInt +
37                " was found in position " + position + ".\n" );
38
39          // obtém a entrada de usuário
40          System.out.print(
41             "Please enter an integer value (-1 to quit): " );
42          searchInt = input.nextInt(); // lê um int do usuário
43          System.out.println();
44       } // fim do while
45    } // fim de main
46 } // fim da classe BinarySearchTest
```

```
13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
Please enter an integer value (-1 to quit): 23
13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                     *
13 23 24 34 35 36 38
         *
13 23 24
    *
The integer 23 was found in position 1.
Please enter an integer value (-1 to quit): 75
13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                     *
                     47 51 68 74 75 85 97
                                 *
```

```
                                    75 85 97
                                        *
                                    75
                                     *
The integer 75 was found in position 12.
Please enter an integer value (-1 to quit): 52
13 23 24 34 35 36 38 42 47 51 68 74 75 85 97
                         *
                           47 51 68 74 75 85 97
                                      *
                           47 51 68
                               *
                                  68
                                   *
The integer 52 was not found.
Please enter an integer value (-1 to quit): -1
```

Figura 19.5 | Utilize pesquisa binária para localizar um item em um array (* na saída marca o elemento no meio).

Eficiência da pesquisa binária

No cenário do pior caso, pesquisar um array classificado de 1.023 elementos leva apenas 10 comparações quando utilizamos uma pesquisa binária. Dividir repetidamente 1.023 por 2 (porque depois de cada comparação podemos eliminar metade do array) e arredondar para baixo (porque também removemos o elemento do meio) produz os valores 511, 255, 127, 63, 31, 15, 7, 3, 1 e 0. O número 1.023 (210 – 1) é dividido por 2 apenas 10 vezes para obter o valor 0, o que indica que não há mais elementos a testar. Dividir por 2 é equivalente a uma comparação no algoritmo de pesquisa binária. Portanto, um array de 1.048.575 ($2^{20} - 1$) elementos exige no máximo 20 comparações para localizar a chave e um array de mais de um bilhão de elementos demanda no máximo 30 comparações para localizar a chave. Isso é uma tremenda melhoria no desempenho em relação à pesquisa linear. Para um array de um bilhão de elementos, isso representa uma diferença entre uma média de 500 milhões de comparações para a pesquisa linear e no máximo apenas 30 comparações para a pesquisa binária! O número máximo de comparações necessárias para a pesquisa binária de qualquer array classificado é o expoente da primeira potência de 2 maior que o número de elementos no array, que é representado como $\log_2 n$. Todos os logaritmos crescem mais ou menos na mesma taxa, portanto na notação Big O a base pode ser omitida. Isso resulta em um Big O de $O(\log n)$ para uma pesquisa binária, que também é conhecida como **tempo de execução logarítmico**.

19.3 Algoritmos de classificação

Classificar dados (isto é, colocar os dados em alguma ordem particular como crescente ou decrescente) é um das aplicações mais importantes da computação. Um banco classifica todos os cheques pelo número de conta de modo que possa preparar extratos bancários individuais no fim de cada mês. As companhias telefônicas classificam suas listas de assinantes por sobrenome e, depois, pelo primeiro nome para facilitar a localização de números de telefone. Praticamente, todas as empresas devem classificar alguns dados e, frequentemente, volumes maciços deles. Classificar dados é um problema intrigante, que faz uso intensivo do computador e que atrai esforços intensos de pesquisa.

Um item importante a entender sobre a classificação é que o resultado final — o array classificado — será o mesmo independentemente do algoritmo que você utiliza para classificar o array. A escolha do algoritmo só afeta o tempo de execução e uso de memória do programa. O restante deste capítulo apresenta três algoritmos de classificação comuns. Os dois primeiros — classificação por seleção e classificação por inserção — são fáceis de programar, mas ineficientes. O último algoritmo — a classificação por intercalação — é muito mais rápido do que a classificação por seleção e a classificação por inserção, mas é mais difícil de programar. Focalizamos a classificação de arrays de dados de tipos primitivos, a saber `int`s. Também é possível classificar arrays de objetos de classes. Discutimos isso na Seção 20.7.1.

19.3.1 Classificação por seleção

Classificação por seleção é um algoritmo de classificação simples, mas ineficiente. Sua primeira iteração seleciona o menor elemento no array e troca-o pelo primeiro elemento. A segunda iteração seleciona o segundo menor item (que é o menor item dos elementos restantes) e troca-o pelo segundo elemento. O algoritmo continua até que a última iteração selecione o segundo maior elemento e permute-o pelo penúltimo índice, deixando o maior elemento no último índice. Depois da *i*-ésima iteração, *i* os menores itens do array serão classificados na ordem crescente nos primeiros *i* elementos do array.

Como um exemplo, considere o array

```
34   56   4   10   77   51   93   30   5   52
```

Um programa que implementa a classificação por seleção primeiro determina o menor elemento (4) desse array, que está contido no índice 2. O programa troca 4 por 34, resultando em

```
4   56   34   10   77   51   93   30   5   52
```

O programa então determina o menor valor dos elementos restantes (todos os elementos, exceto 4), que é 5, contido no índice 8. O programa troca 5 por 56, resultando em

```
4   5   34   10   77   51   93   30   56   52
```

Na terceira iteração, o programa determina o próximo menor valor (10) e o troca por 34.

```
4   5   10   34   77   51   93   30   56   52
```

O processo continua até que o array seja completamente classificado.

```
4   5   10   30   34   51   52   56   77   93
```

Observe que depois da primeira iteração, o menor elemento estará na primeira posição. Depois da segunda iteração, os dois menores elementos estarão na ordem nas duas primeiras posições. Depois da terceira iteração, os três menores elementos estarão na ordem nas três primeiras posições.

A Figura 19.6 declara a classe `SelectionSort`. Essa classe tem duas variáveis de instância `private` — um array de `int`s nomeado `data` e um objeto `static` `Random` para gerar inteiros aleatórios a fim de preencher o array. Quando um objeto da classe `SelectionSort` é instanciado, o construtor (linhas 13–20) cria e inicializa o array `data` com `int`s aleatório no intervalo de 10–99.

```
1  // Figura 19.6: SelectionSort.java
2  // Classe que cria um array preenchido com inteiros aleatórios.
3  // fornece um método para classificar o array com a classificação por seleção.
4  import java.util.Arrays;
5  import java.util.Random;
6
7  public class SelectionSort
8  {
9     private int[] data; // array de valores
10    private static final Random generator = new Random();
11
12    // cria um array de um dado tamanho e o preenche com inteiros aleatórios
13    public SelectionSort( int size )
14    {
15       data = new int[ size ]; // cria espaço para o array
16
17       // preenche o array com ints aleatórios no intervalo de 10-99
18       for ( int i = 0; i < size; i++ )
19          data[ i ] = 10 + generator.nextInt( 90 );
20    } // fim do construtor de SelectionSort
21
22    // classifica o array utilizando a classificação por seleção
23    public void sort()
24    {
25       int smallest; // índice do menor elemento
26
27       // faz um loop sobre data.length - 1 elementos
28       for ( int i = 0; i < data.length - 1; i++ )
29       {
30          smallest = i; // primeiro índice do array remanescente
31
32          // faz um loop para localizar o índice do menor elemento
33          for ( int index = i + 1; index < data.length; index++ )
34             if ( data[ index ] < data[ smallest ] )
35                smallest = index;
36
37          swap( i, smallest ); // troca o menor elemento na posição
38          printPass( i + 1, smallest ); // passagem de saída do algoritmo
39       } // fim do for externo
40    } // fim do método sort
41
42    // método auxiliar para trocar valores em dois elementos
43    public void swap( int first, int second )
44    {
45       int temporary = data[ first ]; // armazena o primeiro em temporário
46       data[ first ] = data[ second ]; // substitui o primeiro pelo segundo
47       data[ second ] = temporary; // coloca o temporário no segundo
48    } // fim do método swap
49
```

```
50      // imprime uma passagem do algoritmo
51      public void printPass( int pass, int index )
52      {
53         System.out.print( String.format( "after pass %2d: ", pass ) );
54
55         // saída de elementos até o item selecionado
56         for ( int i = 0; i < index; i++ )
57            System.out.print( data[ i ] + "   " );
58
59         System.out.print( data[ index ] + "* " ); // indica troca
60
61         // termina de gerar a saída do array
62         for ( int i = index + 1; i < data.length; i++ )
63            System.out.print( data[ i ] + "   " );
64
65         System.out.print( "\n               " ); // para alinhamento
66
67         // indica a quantidade do array que é classificado
68         for ( int j = 0; j < pass; j++ )
69            System.out.print( "--   " );
70         System.out.println( "\n" ); // adiciona fim de linha
71      } // fim do método printPass
72
73      // método para gerar saída de valores no array
74      public String toString()
75      {
76         return Arrays.toString( data );
77      } // fim do método toString
78   } // fim da classe SelectionSort
```

Figura 19.6 | Classe que cria um array preenchido com inteiros aleatórios. Fornece um método para classificar o array com a classificação por seleção.

As linhas 23–40 declaram o método `sort`. A linha 25 declara a variável `smallest`, que armazenará o índice do menor elemento no array restante. As linhas 28–39 fazem um loop `data.length - 1` vez. A linha 30 inicializa o índice do menor elemento como o item atual. As linhas 33–35 fazem um loop sobre os elementos restantes no array. Para cada um desses elementos, a linha 34 compara seu valor com o valor do menor elemento. Se o elemento atual for menor que o menor elemento, a linha 35 atribui o índice do elemento atual a `smallest`. Quando esse loop termina, `smallest` conterá o índice do menor elemento no array restante. A linha 36 chama o método `swap` (linhas 43–48) para colocar o menor elemento restante na próxima área ordenada do array.

A linha 9 da Figura 19.7 cria um objeto `SelectionSort` com 10 elementos. A linha 12 gera saída ao objeto não classificado. A linha 14 chama o método `sort` (linhas 22–39 da Figura 19.6), que classifica os elementos utilizando a classificação por seleção. Em seguida, as linhas 16–17 geram a saída do objeto classificado. A saída utiliza traços (linhas 67–68) para indicar a parte do array que é classificada depois de cada passagem. Um asterisco é colocado ao lado da posição do elemento que foi trocado pelo menor elemento nessa passagem. Em cada passagem, o elemento ao lado do asterisco (especificado na linha 58) e o elemento acima do conjunto de traços mais à direita foram permutados.

```
1   // Figura 19.7: SelectionSortTest.java
2   // Testando a classe de classificação por seleção.
3
4   public class SelectionSortTest
5   {
6      public static void main( String[] args )
7      {
8         // cria um objeto para realizar a classificação por seleção
9         SelectionSort sortArray = new SelectionSort( 10 );
10
11        System.out.println( "Unsorted array:" );
12        System.out.println( sortArray + "\n" ); // imprime um array não classificado
13
14        sortArray.sort(); // classifica o array
15
16        System.out.println( "Sorted array:" );
17        System.out.println( sortArray ); // imprime o array classificado
18     } // fim de main
19  } // fim da classe SelectionSortTest
```

```
Unsorted array:
[61, 87, 80, 58, 40, 50, 20, 13, 71, 45]
after pass  1: 13  87  80  58  40  50  20  61* 71  45
               --
after pass  2: 13  20  80  58  40  50  87* 61  71  45
               --  --
after pass  3: 13  20  40  58  80* 50  87  61  71  45
               --  --  --
after pass  4: 13  20  40  45  80  50  87  61  71  58*
               --  --  --  --
after pass  5: 13  20  40  45  50  80* 87  61  71  58
               --  --  --  --  --
after pass  6: 13  20  40  45  50  58  87  61  71  80*
               --  --  --  --  --  --
after pass  6: 13  20  40  45  50  58  87  61  71  80*
               --  --  --  --  --  --
after pass  7: 13  20  40  45  50  58  61  87* 71  80
               --  --  --  --  --  --  --
after pass  8: 13  20  40  45  50  58  61  71  87* 80
               --  --  --  --  --  --  --  --
after pass  9: 13  20  40  45  50  58  61  71  80  87*
               --  --  --  --  --  --  --  --  --
Sorted array:
[13, 20, 40, 45, 50, 58, 61, 71, 80, 87]
```

Figura 19.7 | Testando a classe de classificação por seleção.

A eficiência da classificação por seleção

O algoritmo de classificação por seleção executa no tempo $O(n^2)$. O método `sort` nas linhas 23–40 da Figura 19.6, que implementa o algoritmo de classificação por seleção, contém dois loops `for`. O loop `for` externo (linhas 28–39) itera pelos primeiros $n - 1$ elementos no array, colocando o menor item restante na sua posição classificada. O loop `for` interno (linhas 33–35) itera por cada item no array restante, procurando o menor elemento. Esse loop é executado $n - 1$ vez durante a primeira iteração do loop externo, $n - 2$ vezes durante a segunda iteração e, então, $n - 3, \ldots, 3, 2, 1$. Esse loop interno irá iterar um total de $n(n - 1)/2$ ou $(n^2 - n)/2$. Na notação Big O, os menores termos são descartados e as constantes são ignoradas, deixando um Big O final de $O(n^2)$.

19.3.2 Classificação por inserção

A **classificação por inserção** é um outro algoritmo de classificação simples, mas ineficiente. A primeira iteração desse algoritmo seleciona o segundo elemento no array e, se for menor que o primeiro elemento, troca-o pelo primeiro elemento. A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos estejam na ordem. Na i-ésima iteração desse algoritmo, os primeiros e elementos no array original serão classificados.

Considere como exemplo o array a seguir. [*Nota:* esse array é idêntico àquele utilizado nas discussões sobre classificação por seleção e classificação por intercalação.]

```
34   56   4   10   77   51   93   30   5   52
```

Um programa que implementa o algoritmo de classificação por inserção primeiro examinará os dois primeiros elementos do array, 34 e 56. Estes já estão na ordem, portanto o programa continua. (Se eles estivessem fora da ordem, o programa iria trocá-los.)

Na próxima iteração, o programa examina o terceiro valor, 4. Esse valor é menor que 56, portanto o programa armazena 4 em uma variável temporária e move o 56 um elemento para a direita. O programa então verifica e determina que 4 é menor que 34, assim move o 34 um elemento para a direita. O programa agora alcançou o começo do array, assim coloca 4 no zero-ésimo elemento. O array agora está

```
4   34   56   10   77   51   93   30   5   52
```

Na próxima iteração, o programa armazena 10 em uma variável temporária. Então, compara 10–56 e move o 56 um elemento para a direita porque ele é maior que 10. O programa então compara 10 com 34, movendo o 34 um elemento para a direita. Quando o programa compara 10–4, ele observa que 10 é maior que 4 e coloca 10 no elemento 1. O array agora está

```
4   10   34   56   77   51   93   30   5   52
```

Utilizando esse algoritmo, na i^o iteração, os primeiros i elementos do array original são classificados, mas talvez eles não estejam nas suas localizações finais — os menores valores talvez sejam localizados posteriormente no array.

A Figura 19.8 declara a classe `InsertionSort`. As linhas 23–47 declaram o método `sort`. A linha 25 declara a variável `insert`, que contém o elemento que você irá inserir enquanto move os outros elementos. As linhas 28–46 fazem um loop por itens `data.length - 1`

no array. Em cada iteração, a linha 31 armazena em `insert` o valor do elemento que será inserido na parte classificada do array. A linha 34 declara e inicializa a variável `moveItem`, que monitora onde inserir o elemento. As linhas 37–42 fazem um loop para localizar a posição correta onde o elemento deve ser inserido. O loop terminará quando o programa alcançar o início do array ou quando alcançar um elemento menor que o valor a ser inserido. A linha 40 move um elemento para a direita e a linha 41 decrementa a posição em que inserir o próximo elemento. Depois de o loop terminar, a linha 44 insere o elemento na posição. A Figura 19.9 é idêntica à Figura 19.7, exceto que ela cria e utiliza um objeto `InsertionSort`. A saída desse programa utiliza traços para indicar a parte do array que é classificada depois de cada passagem. Um asterisco é colocado ao lado do elemento que foi inserido na posição nessa passagem.

```
1  // Figura 19.8: InsertionSort.java
2  // Classe que cria um array preenchido com inteiros aleatórios.
3  // fornece um método para classificar o array com a classificação por inserção.
4  import java.util.Arrays;
5  import java.util.Random;
6
7  public class InsertionSort
8  {
9     private int[] data; // array de valores
10    private static final Random generator = new Random();
11
12    // cria um array de um dado tamanho e o preenche com inteiros aleatórios
13    public InsertionSort( int size )
14    {
15       data = new int[ size ]; // cria espaço para o array
16
17       // preenche o array com ints aleatórios no intervalo de 10-99
18       for ( int i = 0; i < size; i++ )
19          data[ i ] = 10 + generator.nextInt( 90 );
20    } // fim do construtor de InsertionSort
21
22    // classifica o array utilizando a classificação por inserção
23    public void sort()
24    {
25       int insert; // variável temporária para armazenar o elemento a inserir
26
27       // faz um loop sobre data.length - 1 elemento
28       for ( int next = 1; next < data.length; next++ )
29       {
30          // armazena o valor no elemento atual
31          insert = data[ next ];
32
33          // inicializa a localização para colocar elemento
34          int moveItem = next;
35
36          // procura o local para colocar o elemento atual
37          while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
38          {
39             // desloca o elemento direito um slot
40             data[ moveItem ] = data[ moveItem - 1 ];
41             moveItem--;
42          } // fim do while
43
44          data[ moveItem ] = insert; // posiciona o elemento inserido
45          printPass( next, moveItem ); // passagem de saída do algoritmo
46       } // for final
47    } // fim do método sort
48
49    // imprime uma passagem do algoritmo
50    public void printPass( int pass, int index )
51    {
52       System.out.print( String.format( "after pass %2d: ", pass ) );
53
54       // gera saída dos elementos até o item trocado
55       for ( int i = 0; i < index; i++ )
56          System.out.print( data[ i ] + "  " );
57
58       System.out.print( data[ index ] + "* " ); // indica troca
59
60       // termina de gerar a saída do array
```

```
61          for ( int i = index + 1; i < data.length; i++ )
62             System.out.print( data[ i ] + "   " );
63
64          System.out.print( "\n               " ); // para alinhamento
65
66          // indica quantidade do array que é classificado
67          for( int i = 0; i <= pass; i++ )
68             System.out.print( "--  " );
69          System.out.println( "\n" ); // adiciona fim de linha
70       } // fim do método printPass
71
72       // método para gerar saída de valores no array
73       public String toString()
74       {
75          return Arrays.toString( data );
76       } // fim do método toString
77    } // fim da classe InsertionSort
```

Figura 19.8 | Classe que cria um array preenchido com inteiros aleatórios. Fornece um método para classificar o array com a classificação por inserção.

```
1  // Figura 19.9: InsertionSortTest.java
2  // Testando a classe de classificação por inserção.
3
4  public class InsertionSortTest
5  {
6     public static void main( String[] args )
7     {
8        // cria o objeto para executar a classificação por inserção
9        InsertionSort sortArray = new InsertionSort( 10 );
10
11       System.out.println( "Unsorted array:" );
12       System.out.println( sortArray + "\n" ); // imprime um array não classificado
13
14       sortArray.sort(); // classifica o array
15
16       System.out.println( "Sorted array:" );
17       System.out.println( sortArray ); // imprime o array classificado
18    } // fim de main
19 } // fim da classe InsertionSortTest
```

```
Unsorted array:
[40, 17, 45, 82, 62, 32, 30, 44, 93, 10]
after pass  1: 17* 40  45  82  62  32  30  44  93  10
               --  --
after pass  2: 17  40  45* 82  62  32  30  44  93  10
               --  --  --
after pass  3: 17  40  45  82* 62  32  30  44  93  10
               --  --  --  --
after pass  4: 17  40  45  62* 82  32  30  44  93  10
               --  --  --  --  --
after pass  5: 17  32* 40  45  62  82  30  44  93  10
               --  --  --  --  --  --
after pass  6: 17  30* 32  40  45  62  82  44  93  10
               --  --  --  --  --  --  --
after pass  7: 17  30  32  40  44* 45  62  82  93  10
               --  --  --  --  --  --  --  --
after pass  8: 17  30  32  40  44  45  62  82  93* 10
               --  --  --  --  --  --  --  --  --
after pass  9: 10* 17  30  32  40  44  45  62  82  93
               --  --  --  --  --  --  --  --  --  --
Sorted array:
[10, 17, 30, 32, 40, 44, 45, 62, 82, 93]
```

Figura 19.9 | Testando a classe de classificação por inserção.

Eficiência da classificação por inserção

O algoritmo de classificação por inserção também é executado em tempo $O(n^2)$. Como ocorre com a classificação por seleção, a implementação da classificação por inserção (linhas 23–47 da Figura 19.8) contém dois loops. O loop `for` (linhas 28–46) itera `data.length - 1` vez, inserindo um elemento na posição apropriada nos elementos classificados até o momento. Para os propósitos desse aplicativo, `data.length - 1` é equivalente a $n - 1$ (uma vez que `data.length` é o tamanho do array). O loop `while` (linhas 37–42) itera pelos elementos precedentes no array. No pior caso, esse loop `while` exigirá $n - 1$ comparações. Cada loop individual é executado no tempo $O(n)$. Na notação Big O, loops aninhados significam que você deve multiplicar o número de comparações. Para cada iteração de um loop externo, haverá certo número de iterações do loop interno. Nesse algoritmo, para cada $O(n)$ iterações do loop externo, haverá $O(n)$ iterações do loop interno. Multiplicar esses valores resulta em uma Big O de $O(n^2)$.

19.3.3 Classificação por intercalação

A **classificação por intercalação** é um algoritmo de classificação eficiente, mas é conceitualmente mais complexo que a classificação por seleção e a classificação por inserção. O algoritmo de classificação por intercalação classifica um array dividindo-o em dois subarrays de igual tamanho, classificando cada subarray e, então, os mesclando em um array maior. Com um número ímpar de elementos, o algoritmo cria os dois subarrays de tal maneira que um deles tenha um elemento a mais que o outro.

A implementação da classificação por intercalação nesse exemplo é recursiva. O caso básico é um array com um único elemento, que está, naturalmente, classificado, e, por isso, nesse caso a classificação por intercalação retorna imediatamente. O passo de recursão divide o array em duas partes aproximadamente iguais, classifica-as recursivamente e, então, intercala os dois arrays classificados em um array classificado maior.

Suponha que o algoritmo já tenha intercalado arrays menores para criar os arrays classificados A:

```
4   10   34   56   77
```

e B:

```
5   30   51   52   93
```

A classificação por intercalação combina esses dois arrays em um array classificado maior. O menor elemento em A é 4 (localizado no zero-ésimo índice de A). O menor elemento em B é 5 (localizado no zero-ésimo índice de B). A fim de determinar o menor elemento no maior array, o algoritmo compara 4 e 5. O valor em A é menor, assim 4 torna-se o primeiro elemento no array intercalado. O algoritmo continua comparando 10 (o segundo elemento em A) com 5 (o primeiro elemento em B). O valor de B é menor, portanto 5 torna-se o segundo elemento no maior array. O algoritmo continua comparando 10–30, com 10 tornando-se o terceiro elemento no array e assim por diante.

As linhas 22–25 da Figura 19.10 declaram o método `sort`. A linha 24 chama o método `sortArray` com `0` e `data.length - 1` como os argumentos — correspondente aos índices inicial e final, respectivamente, do array a ser classificado. Esses valores informam ao método `sortArray` a operar no array inteiro.

O método `sortArray` é declarado nas linhas 28–49. A linha 31 testa o caso básico. Se o tamanho do array for `1`, o array já está classificado, assim o método retorna imediatamente. Se o tamanho do array for maior que `1`, o método divide o array em dois, chama recursivamente o método `sortArray` para classificar os dois subarrays e então os intercala. A linha 43 chama recursivamente o método `sortArray` na primeira metade do array e a linha 44 chama recursivamente o método `sortArray` na segunda metade. Depois que essas duas chamadas de método retornam, cada metade do array terá sido classificada. A linha 47 chama o método `merge` (linhas 52–91) nas duas metades do array para combinar os dois arrays classificados em um array classificado maior.

```
1  // Figura 19.10: MergeSort.java
2  // A classe cria um array preenchido com inteiros aleatórios.
3  // Fornece um método para classificar o array com a classificação por intercalação.
4  import java.util.Random;
5
6  public class MergeSort
7  {
8     private int[] data; // array de valores
9     private static final Random generator = new Random();
10
11    // cria um array de um dado tamanho e o preenche com inteiros aleatórios
12    public MergeSort( int size )
13    {
14       data = new int[ size ]; // cria espaço para o array
15
16       // preenche o array com ints aleatórios no intervalo de 10-99
17       for ( int i = 0; i < size; i++ )
18          data[ i ] = 10 + generator.nextInt( 90 );
19    } // fim do construtor de MergeSort
20
```

```
21       // chama o método split recursivo para iniciar a classificação por intercalação
22       public void sort()
23       {
24          sortArray( 0, data.length - 1 ); // divide o array inteiro
25       } // fim do método sort
26
27       // divide o array, classifica subarrays e intercala subarrays no array classificado
28       private void sortArray( int low, int high )
29       {
30          // caso básico de teste; tamanho do array é igual a 1
31          if ( ( high - low ) >= 1 ) // se não for o caso básico
32          {
33             int middle1 = ( low + high ) / 2; // calcula o meio do array
34             int middle2 = middle1 + 1; // calcula o próximo elemento
35
36             // gera uma saída do passo de divisão
37             System.out.println( "split:   " + subarray( low, high ) );
38             System.out.println( "         " + subarray( low, middle1 ) );
39             System.out.println( "         " + subarray( middle2, high ) );
40             System.out.println();
41
42             // divide o array pela metade; classifica cada metade (chamadas recursivas)
43             sortArray( low, middle1 ); // primeira metade do array
44             sortArray( middle2, high ); // segunda metade do array
45
46             // intercala dois arrays classificados depois que as chamadas de divisão retornam
47             merge ( low, middle1, middle2, high );
48          } // fim do if
49       } // fim do método sortArray
50
51       // intercala dois subarrays classificados em um subarray classificado
52       private void merge( int left, int middle1, int middle2, int right )
53       {
54          int leftIndex = left; // índice no subarray esquerdo
55          int rightIndex = middle2; // índice no subarray direito
56          int combinedIndex = left; // índice no array temporário funcional
57          int[] combined = new int[ data.length ]; // array funcional
58
59          // gera saída de dois subarrays antes de mesclar
60          System.out.println( "merge:   " + subarray( left, middle1 ) );
61          System.out.println( "         " + subarray( middle2, right ) );
62
63          // intercala arrays até alcançar o fim de um deles
64          while ( leftIndex <= middle1 && rightIndex <= right )
65          {
66             // coloca o menor dos dois elementos atuais no resultado
67             // e o move para o próximo espaço nos arrays
68             if ( data[ leftIndex ] <= data[ rightIndex ] )
69                combined[ combinedIndex++ ] = data[ leftIndex++ ];
70             else
71                combined[ combinedIndex++ ] = data[ rightIndex++ ];
72          } // fim do while
73
74          // se o array esquerdo estiver vazio
75          if ( leftIndex == middle2 )
76             // copia o restante do array direito
77             while ( rightIndex <= right )
78                combined[ combinedIndex++ ] = data[ rightIndex++ ];
79          else // o array direito está vazio
80             // copia o restante do array esquerdo
81             while ( leftIndex <= middle1 )
82                combined[ combinedIndex++ ] = data[ leftIndex++ ];
83
84          // copia os valores de volta ao array original
85          for ( int i = left; i <= right; i++ )
86             data[ i ] = combined[ i ];
87
88          // gera saída do array intercalado
89          System.out.println( "         " + subarray( left, right ) );
```

```
90          System.out.println();
91       } // fim do método merge
92
93       // método para gerar saída de certos valores no array
94       public String subarray( int low, int high )
95       {
96          StringBuilder temporary = new StringBuilder();
97
98          // gera espaços para alinhamento
99          for ( int i = 0; i < low; i++ )
100            temporary.append( "   " );
101
102         // gera a saída dos elementos que permanecem no array
103         for ( int i = low; i <= high; i++ )
104            temporary.append( " " + data[ i ] );
105
106         return temporary.toString();
107      } // fim do método subarray
108
109      // método para gerar saída de valores no array
110      public String toString()
111      {
112         return subarray( 0, data.length - 1 );
113      } // fim do método toString
114   } // fim da classe MergeSort
```

Figura 19.10 | Classe que cria um array preenchido com inteiros aleatórios. Fornece um método para classificar o array com a classificação por intercalação.

As linhas 64–72 no método `merge` fazem um loop até que o programa alcance o fim de um dos subarrays. A linha 68 testa qual elemento no começo dos arrays é o menor. Se o elemento no array esquerdo for o menor, a linha 69 coloca-o na posição no array combinado. Se o elemento no array direito for menor, a linha 71 coloca-o na posição no array combinado. Quando o loop `while` completar (linha 72), um subarray inteiro é colocado no array combinado, mas o outro subarray ainda contém dados. A linha 75 testa se o array esquerdo alcançou o fim. Se alcançou, as linhas 77–78 preenchem o array combinado com os elementos do array direito. Se o array esquerdo não alcançou o fim, o array direito deve então ter alcançado o fim e as linhas 81–82 preenchem o array combinado com os elementos do array esquerdo. Por fim, as linhas 85–86 copiam o array combinado para o array original. A Figura 19.11 cria e utiliza um objeto `MergeSort`. A saída desse programa exibe as divisões e intercalações realizadas pela classificação por intercalação, mostrando o progresso da classificação em cada passo do algoritmo. É importante parar e analisar essas saídas a fim de entender totalmente esse elegante algoritmo de classificação.

```
1   // Figura 16.11: MergeSortTest.java
2   // Testando a classe de classificação por intercalação.
3
4   public class MergeSortTest
5   {
6      public static void main( String[] args )
7      {
8         // cria o objeto para executar a classificação por intercalação
9         MergeSort sortArray = new MergeSort( 10 );
10
11        // imprime um array não classificado
12        System.out.println( "Unsorted:" + sortArray + "\n" );
13
14        sortArray.sort(); // classifica o array
15
16        // imprime o array classificado
17        System.out.println( "Sorted:  " + sortArray );
18     } // fim de main
19  } // fim da classe MergeSortTest
```

```
Unsorted: 75 56 85 90 49 26 12 48 40 47
split:    75 56 85 90 49 26 12 48 40 47
          75 56 85 90 49
                        26 12 48 40 47
```

```
split:    75 56 85 90 49
          75 56 85
                   90 49

split:    75 56 85
          75 56
                85
split:    75 56
          75
             56
merge:    75
             56
          56 75
merge:    56 75
                85
          56 75 85
split:              90 49
                    90
                       49
merge:              90
                       49
                    49 90
merge:    56 75 85
                    49 90
          49 56 75 85 90
split:                     26 12 48 40 47
                           26 12 48
                                    40 47
split:                     26 12 48
                           26 12
                                 48
split:                     26 12
                           26
                              12
merge:                     26
                              12
                           12 26
merge:                     12 26
                                 48
                           12 26 48
split:                              40 47
                                    40
                                       47
merge:                              40
                                       47
                                    40 47
merge:                     12 26 48
                                    40 47
                           12 26 40 47 48
merge:    49 56 75 85 90
                           12 26 40 47 48
          12 26 40 47 48 49 56 75 85 90
Sorted:   12 26 40 47 48 49 56 75 85 90
```

Figura 19.11 | Testando a classe de classificação por intercalação.

Eficiência da classificação por intercalação

A classificação por intercalação é um algoritmo muito mais eficiente do que a classificação por inserção ou por seleção. Considere a primeira chamada (não recursiva) ao método `sortArray`. Esta resulta em duas chamadas recursivas ao método `sortArray` e cada subarray com aproximadamente metade do tamanho do array original e uma única chamada ao método `merge`. Essa chamada para `merge` exige, no pior dos casos, $n - 1$ comparações para preencher o array original, que é $O(n)$. (Lembre-se de que cada elemento no array pode ser escolhido comparando um elemento de cada um dos subarrays). As duas chamadas ao método `sortArray` resultam em quatro outras chamadas recursivas ao método `sortArray`, cada uma com um subarray com aproximadamente um quarto do tamanho do array original

e duas chamadas ao método `merge`. Cada uma dessas duas chamadas ao método `merge` exige, no pior dos casos, $n/2 - 1$ comparações para um número total de comparações de $O(n)$. Esse processo continua, com cada chamada a `sortArray` gerando duas chamadas adicionais a `sortArray` e uma chamada para `merge`, até que o algoritmo tenha dividido o array em subarrays de um elemento. Em cada nível, $O(n)$ comparações são exigidas para intercalar os subarrays. Cada nível divide o tamanho dos arrays pela metade, portanto dobrar o tamanho do array exige mais um nível. Quadruplicar o tamanho do array exige mais dois níveis. Esse padrão é logarítmico e resulta em $\log_2 n$ níveis. Isso resulta em uma eficiência total de $\mathbf{O(n \log n)}$.

A Figura 19.12 resume os algoritmos de pesquisa e classificação cobertos neste capítulo com a notação Big O para cada um. A Figura 19.13 lista os valores de Big O que abordamos neste capítulo com alguns valores para n a fim de destacar as diferenças nas taxas de crescimento.

Algoritmo	Posição	Big O
Algoritmos de pesquisa:		
Pesquisa linear	Seção 19.2.1	$O(n)$
Pesquisa binária	Seção 19.2.2	$O(\log n)$
Pesquisa linear recursiva	Exercícios 19.8	$O(n)$
Pesquisa binária recursiva	Exercício 19.9	$O(\log n)$
Algoritmos de classificação:		
Classificação por seleção	Seção 19.3.1	$O(n^2)$
Classificação por inserção	Seção 19.3.2	$O(n^2)$
Classificação por intercalação	Seção 19.3.3	$O(n \log n)$
Classificação por flutuação	Exercícios 19.5 e 19.6	$O(n^2)$

Figura 19.12 | Algoritmos de pesquisa e classificação com valores na notação Big O.

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10.000
1000	3	1000	3000	10^6
1.000.000	6	1.000.000	6.000.000	10^{12}
1.000.000.000	9	1.000.000.000	9.000.000.000	10^{18}

Figura 19.13 | Número de comparações para notações Big O comuns.

19.4 Conclusão

Este capítulo fez uma introdução à pesquisa e classificação. Discutimos dois algoritmos de pesquisa — a pesquisa linear e a pesquisa binária — e três algoritmos de classificação — classificação por seleção, classificação por inserção e classificação por intercalação. Introduzimos a notação Big O, que ajuda a analisar a eficiência de um algoritmo. Os três capítulos a seguir continuam nossa discussão das estruturas de dados dinâmicas que podem aumentar ou diminuir de tamanho em tempo de execução. O Capítulo 20 apresenta os algoritmos predefinidos da API do Java (como pesquisa e classificação) e coleções genéricas. O Capítulo 21 demonstra como usar as capacidades genéricas do Java para implementar classes e métodos genéricos. O Capítulo 22 discute os detalhes da implementação das estruturas de dados genéricas.

Resumo

Seção 19.1 Introdução

- Pesquisar dados envolve determinar se uma chave de pesquisa está nos dados e, se estiver, encontrar sua localização.
- Classificar envolve organizar os dados em uma ordem.

Seção 19.2 Algoritmos de pesquisa

- O algoritmo de pesquisa linear pesquisa cada elemento no array sequencialmente até encontrar o elemento correto, ou até atingir o fim do array sem localizar o elemento.
- Uma diferença importante entre algoritmos de pesquisa é o esforço que eles exigem a fim de retornar um resultado.
- A notação Big O descreve a eficiência de um algoritmo em termos do trabalho exigido para resolver um problema. Para os algoritmos de pesquisa e classificação isso costuma depender do número de elementos nos dados.
- Um algoritmo que é $O(1)$ não necessariamente exige somente uma comparação. Significa apenas que o número de comparações não aumenta à medida que o tamanho do array aumenta.
- Diz-se que um algoritmo $O(n)$ tem um tempo de execução linear.
- A notação Big O é projetada para destacar fatores dominantes e ignorar termos sem importância com valores de n altos.
- A notação Big O se preocupa com a taxa de crescimento dos tempos de execução do algoritmo, portanto constantes são ignoradas.
- O algoritmo de pesquisa linear executa a $O(n)$ vezes.
- O pior caso na pesquisa linear é que cada elemento deve ser verificado para determinar se o item de pesquisa existe. Isso ocorre se a chave de pesquisa for o último elemento no array ou não estiver presente.
- O algoritmo de pesquisa binária é mais eficiente que o algoritmo de pesquisa linear, mas exige que o array seja classificado.
- A primeira iteração da pesquisa binária testa o elemento do meio no array. Se este for a chave de pesquisa, o algoritmo retornará sua localização. Se a chave de pesquisa for menor que o elemento do meio, a pesquisa continuará com a primeira metade do array. Se a chave de pesquisa for maior que o elemento do meio, a pesquisa continuará com a segunda metade do array. Cada iteração testa o valor do meio do array restante e, se o elemento não for localizado, elimina metade dos elementos restantes.
- A pesquisa binária é um algoritmo de pesquisa mais eficiente do que a pesquisa linear, pois cada comparação elimina metade dos elementos no array.
- A pesquisa binária executa em $O(\log n)$ vezes, pois cada passo remove metade dos elementos restantes.
- Se o tamanho do array for dobrado, a pesquisa binária só requer uma comparação extra.

Seção 19.3 Algoritmos de classificação

- Classificação por seleção é um algoritmo de classificação simples, mas ineficiente.
- A primeira iteração da classificação por seleção seleciona o menor item no array e troca-o pelo primeiro elemento. A segunda iteração da classificação por seleção seleciona o segundo menor item (que é o menor item restante) e troca-o pelo segundo elemento. A classificação por seleção continua até que a última iteração selecione o segundo maior elemento e permute-o pelo penúltimo elemento, deixando o maior elemento no último índice. Na i-ésima iteração da classificação por seleção, os menores i itens do array inteiro são classificados nos primeiros i índices.
- O algoritmo de classificação por seleção executa no tempo $O(n^2)$.
- A primeira iteração da classificação por inserção seleciona o segundo elemento no array e, se for menor que o primeiro elemento, troca-o pelo primeiro elemento. A segunda iteração desse tipo examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos. Depois da i-ésima iteração de classificação por inserção, os primeiro i elementos no array original são classificados.
- O algoritmo de classificação por inserção executa a $O(n^2)$ vezes.
- A classificação por intercalação é um algoritmo de classificação mais rápido, mas mais complexo de implementar, que a classificação por seleção e classificação por inserção.
- O algoritmo de classificação por intercalação classifica um array dividindo-o em dois subarrays de igual tamanho, classificando cada subarray recursivamente e mesclando os subarrays em um array maior.
- O caso básico da classificação por intercalação é um array com um único elemento. O array de um elemento já está classificado, assim a classificação por intercalação retorna imediatamente quando é chamada com um array de um elemento. A parte da combinação da classificação por intercalação recebe dois arrays classificados e combina-os em um array classificado maior.
- A classificação por intercalação realiza a intercalação examinando o primeiro elemento em cada array, que também é o menor elemento no array. A classificação por intercalação seleciona o menor destes e os coloca no primeiro elemento do maior array. Se ainda houver elementos no subarray, a classificação por intercalação examina o segundo deles (que agora é o menor elemento remanescente) e o compara ao primeiro elemento no outro subarray. A classificação por intercalação continua esse processo até que o maior array seja preenchido.
- No pior caso, a primeira chamada à classificação por intercalação tem de fazer $O(n)$ comparações para preencher os n slots no array final.
- A parte da intercalação do algoritmo de classificação por intercalação é realizada em dois subarrays, cada um com aproximadamente o tamanho $n/2$. Criar cada um desses subarrays exige $n / 2 - 1$ comparações para cada subarray ou o total de $O(n)$ comparações. Esse padrão continua à medida que cada nível constrói duas vezes o número de arrays, mas cada um tem a metade do tamanho do array anterior.
- Semelhante à pesquisa binária, essa divisão em metades resulta em $\log n$ níveis para uma eficiência total de $O(n \log n)$.

Terminologia

algoritmo de classificação por inserção, 625
algoritmo de classificação por intercalação, 628
algoritmo de classificação por seleção, 622
algoritmo de pesquisa binária, 619
chave de classificação, 615
chave de pesquisa, 615
classificação de dados, 615
classificação por inserção, 625
classificação por intercalação, 628
classificação por seleção, 622
Notação Big O, 618
$O(1)$, tempo, 618
$O(\log n)$, tempo, 622
$O(n)$, tempo, 618
$O(n^2)$, tempo, 618
$O(n \log n)$, tempo, 632
pesquisa de dados, 615
`sort`, método da classe `Arrays`, 619
tempo de execução constante, 618
tempo de execução linear, 618
tempo de execução logarítmico, 622
tempo de execução quadrático, 618

Exercícios de autorrevisão

19.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um aplicativo de classificação por seleção demoraria aproximadamente _____ vezes a mais para ser executado em um array de 128 elementos do que em um array de 32 elementos.

b) A eficiência da classificação por intercalação é _____.

19.2 Qual aspecto -have da pesquisa binária e da classificação por intercalação é responsável pela parte logarítmica das suas respectivas Big Os?

19.3 Em que sentido a classificação por inserção é superior à classificação por intercalação? Em que sentido a classificação por intercalação é superior à classificação por inserção?

19.4 No texto, dizemos que depois que a classificação por intercalação divide o array em dois subarrays, ela então classifica esses dois subarrays e os intercala. Por que alguém ficaria intrigado com a nossa afirmação de que "ele então classifica esses dois subarrays"?

Respostas dos exercícios de autorrevisão

19.1 a) 16, porque um algoritmo $O(n^2)$ demora 16 vezes mais para classificar quatro vezes o mesmo número de informações. b) $O(n \log n)$.

19.2 Esses dois algoritmos incorporam a "divisão por metades" — de algum modo reduzindo algo pela metade. A pesquisa binária elimina uma metade do array depois de cada comparação. A classificação por intercalação divide o array pela metade toda vez que é chamada.

19.3 A classificação por inserção é mais fácil de entender e programar do que a classificação por intercalação. A classificação por intercalação é muito mais eficiente $[O(n \log n)]$ do que a classificação por inserção $[O(n^2)]$.

19.4 Em certo sentido, ela na verdade não classifica esses dois subarrays. Simplesmente continua a dividir o array original pela metade até que ele forneça um subarray de um elemento, que está, naturalmente, classificado. Ela, então, cria os dois subarrays originais intercalando esses arrays de um elemento para formar subarrays maiores, que são então intercalados e assim por diante.

Exercícios

19.5 (***Classificação por flutuação [bubble sort]***) Implemente a classificação por flutuação — outra técnica simples, mas ineficiente de classificação. É chamada classificação por flutuação ou classificação por afundamento porque os menores valores gradualmente "borbulham" ou "flutuam" no seu caminho para a parte superior do array (isto é, na direção do primeiro elemento) como bolhas de ar que emergem na superfície, enquanto os maiores valores afundam na parte inferior (final) do array. A técnica utiliza loops aninhados para fazer várias passagens pelo array. Cada passagem compara pares sucessivos de elementos. Se um par estiver na ordem crescente (ou os valores forem iguais), a classificação por flutuação deixa os valores como estão. Se um par estiver na ordem decrescente, a classificação por flutuação troca seus valores no array.
A primeira passagem compara os dois primeiros elementos do array e troca seus valores se necessário. Ela então compara o segundo e terceiros elementos no array. O fim dessa passagem compara os dois últimos elementos no array e troca-os se necessário. Depois de uma passagem, o maior elemento estará no último índice. Depois de duas passagens, os dois maiores elementos estarão nos dois últimos índices. Explique por que a classificação por flutuação é um algoritmo $O(n^2)$.

19.6 (***Classificação por flutuação aprimorada***) Faça as modificações simples a seguir para melhorar o desempenho da classificação por flutuação que você desenvolveu no Exercício 19.5:

a) Depois da primeira passagem, garante-se que o número maior está no elemento de número mais alto do array; após a segunda passagem, os dois números mais altos estão "no lugar"; e assim por diante. Em vez de fazer nove comparações em cada passagem para um array de 10 elementos, modifique a classificação por flutuação para fazer oito comparações na segunda passagem, sete na terceira passagem e assim por diante.

b) Os dados no array já podem estar na ordem adequada ou quase adequada, então por que fazer nove passagens se menos seriam suficientes? Modifique a classificação para verificar no fim de cada passagem se alguma troca foi feita. Se nenhuma troca tiver sido feita, os dados já devem estar na ordem apropriada, então o programa deve terminar. Se trocas foram feitas, pelo menos mais uma passagem é necessária.

19.7 (***Bucket sort***) Uma classificação do tipo *bucket sort* inicia com um array unidimensional de inteiros positivos a ser classificado e um array bidimensional de inteiros com linhas indexadas de 0–9 e colunas indexadas de 0 a $n - 1$, onde n é o número dos valores a ser classificado. Cada linha do array bidimensional é chamada *bucket*. Escreva uma classe chamada `BucketSort` que contém um método chamado `sort` que opera desta maneira:

a) Coloque cada valor do array unidimensional em uma linha do array de bucket, com base nas "unidades" (mais à direita) do dígito. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3 e 100 é colocado na linha 0. Esse procedimento é chamado de *passagem de distribuição*.

b) Realize um loop pelo array de bucket linha por linha e copie os valores de volta para o array original. Esse procedimento é chamado *passagem de coleta*. A nova ordem dos valores precedentes no array unidimensional é 100, 3 e 97.

c) Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares etc.) Na segunda passagem (dígitos das dezenas), 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem nenhum dígito de dezena) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no array unidimensional é 100, 3 e 97. Na terceira passagem (dígitos das centenas), 100 é colocado na linha 1, 3 é colocado na linha 0 e 97 é colocado na linha 0 (depois do 3). Depois dessa última passagem de coleta, o array original está na ordem classificada.

Observe que o array bidimensional dos buckets tem 10 vezes o comprimento do array de inteiros sendo classificado. Essa técnica de classificação fornece um melhor desempenho do que uma classificação por flutuação, mas exige muito mais memória — a classificação por flutuação exige espaço para somente um elemento adicional de dados. Essa comparação é um exemplo da relação de troca espaço/tempo: A bucket sort utiliza mais memória que a classificação por flutuação, mas seu desempenho é melhor. Essa versão da bucket sort requer cópia de todos os dados de volta para o array original a cada passagem. Outra possibilidade é criar um segundo array de bucket bidimensional e permutar os dados repetidamente entre os dois arrays de bucket.

19.8 **(*Pesquisa linear recursiva*)** Modifique a Figura 19.2 para utilizar o método recursivo `recursiveLinearSearch` a fim de realizar uma pesquisa linear do array. O método deve receber a chave de pesquisa e o índice inicial como argumentos. Se a chave de pesquisa for encontrada, seu índice no array é retornado; caso contrário, `-1` é retornado. Cada chamada ao método recursivo deve verificar um índice no array.

19.9 **(*Pesquisa binária recursiva*)** Modifique a Figura 19.4 para utilizar o método recursivo `recursiveBinarySearch` a fim de realizar uma pesquisa binária do array. O método deve receber a chave de pesquisa, o índice inicial e o índice final como argumentos. Se a chave de pesquisa for encontrada, seu índice no array é retornado. Se a chave de pesquisa não for encontrada, é retornado `-1`.

19.10 **(*Quicksort*)** A técnica de classificação recursiva chamada quicksort utiliza o seguinte algoritmo básico para um array unidimensional de valores:

a) *Passo de partição:* selecione o primeiro elemento do array não classificado e determine sua localização final no array classificado (isto é, todos os valores à esquerda do elemento no array são menores que o elemento e todos os valores à direita do elemento no array são maiores que o elemento — mostramos como fazer isso a seguir). Agora temos um elemento em sua posição adequada e dois subarrays não classificados.

b) *Passo recursivo:* realize o *Passo 1* em cada subarray não classificado. Toda vez que o *Passo 1* for realizado em um subarray, outro elemento é colocado em sua posição final no array classificado e dois subarrays não classificados são criados. Quando um subarray consiste em apenas um elemento, esse elemento está na sua localização final (porque o array de um elemento já está classificado).

O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subarray? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de partição — ele será colocado em sua localização final no array classificado) :

37 2 6 4 89 8 10 12 68 45

Iniciando a partir do elemento mais à direita do array, compare cada elemento com **37** até um elemento menor que **37** ser encontrado; então permute **37** e esse elemento. O primeiro elemento menor que **37** é 12, então **37** e 12 são permutados. O novo array é

12 2 6 4 89 8 10 *37* 68 45

O elemento 12 está em itálico para indicar que acabou ser permutado com **37**.

Iniciando a partir da esquerda do array, mas começando com o elemento depois de 12, compare cada elemento com **37** até um elemento maior que **37** ser encontrado, então permute **37** e esse elemento. O primeiro elemento maior que **37** é 89, então **37** e 89 foram permutados. O novo array é

12 2 6 4 *37* 8 10 *89* 68 45

Iniciando da direita, mas começando com o elemento antes de 89, compare cada elemento com **37** até um elemento menor que **37** ser encontrado — então permute **37** e esse elemento. O primeiro elemento menor que **37** é 10, então **37** e 10 são permutados. O novo array é

12 2 6 4 *10* 8 *37* 89 68 45

Iniciando da esquerda, mas começando com o elemento depois de 10, compare cada elemento para **37** até um elemento maior que **37** ser encontrado — então permute **37** e esse elemento. Não há mais elementos maiores que **37**, então, quando comparamos **37** com ele mesmo, sabemos que **37** foi colocado na sua localização final no array classificado. Cada valor à esquerda do **37** é menor que ele e cada valor à direita do **37** é maior que ele.

Uma vez que a partição foi aplicada no array anterior, há dois subarrays não classificados. O subarray com valores menores que 37 contém 12, 2, 6, 4, 10 e 8. O subarray com valores maiores que 37 contém 89, 68 e 45. A classificação continua recursivamente com ambos os subarrays sendo particionados da mesma maneira que o array original.

Com base na discussão precedente, escreva o método recursivo `quickSortHelper` para classificar um array unidimensional de inteiros. O método deve receber como argumentos um índice inicial e um índice final no array original sendo classificado.

20

Acho que essa é a coleção mais extraordinária de talento e conhecimento humano já reunida na Casa Branca — com a possível exceção de quando Thomas Jefferson jantava sozinho.
— John F. Kennedy

As formas que um contêiner brilhante pode conter!
— Theodore Roethke

Viaje por todo o universo em um mapa.
— Miguel de Cervantes

Não é pela idade mas pela capacidade que se adquire a sabedoria.
— Tito Macio Plauto

Uma charada envolvida em um mistério, dentro de um enigma.
— Winston Churchill

Coleções genéricas

Objetivos

Neste capítulo, você aprenderá:

- O que são coleções.
- A utilizar a classe `Arrays` para manipulações de array.
- A formar estruturas de dados encadeadas utilizando referências, classes autorreferenciais e recursão.
- A reconhecer as classes empacotadoras de tipo que permitem aos programas processar valores de dados primitivos como objetos.
- A utilizar implementações de estrutura de coleções (estrutura de dados pré-construída).
- A utilizar métodos de estrutura de coleções (como `search`, `sort` e `fill`) para manipular coleções.
- A utilizar as interfaces de estrutura de coleções para programar com coleções polimorficamente.
- A utilizar iteradores para "percorrer" uma coleção.
- A utilizar tabelas de hash persistentes manipuladas com objetos da classe `Properties`.
- A utilizar empacotadores de sincronização e de modificabilidade.

Sumário

20.1 Introdução
20.2 Visão geral das coleções
20.3 Classes empacotadoras de tipo para tipos primitivos
20.4 Autoboxing e auto-unboxing
20.5 Interface `Collection` e classe `Collections`
20.6 Listas
20.6.1 `ArrayList` e `Iterator`
20.6.2 `LinkedList`
20.7 Métodos de coleções
20.7.1 Método `sort`
20.7.2 Método `shuffle`
20.7.3 Métodos `reverse`, `fill`, `copy`, `max` e `min`
20.7.4 Método `binarySearch`
20.7.5 Métodos `addAll`, `frequency` e `disjoint`
20.8 Classe `Stack` do pacote `java.util`
20.9 Classe `PriorityQueue` e `Interface Queue`
20.10 Conjuntos
20.11 Mapas
20.12 Classe `Properties`
20.13 Coleções sincronizadas
20.14 Coleções não modificáveis
20.15 Implementações abstratas
20.16 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

20.1 Introdução

Neste capítulo, consideramos a **estrutura de coleções** do Java, que contém estruturas de dados predefinidas, interfaces e métodos para manipular essas estruturas de dados. Alguns exemplos de coleções são as cartas de um jogo de cartas, suas músicas favoritas armazenadas no seu computador, os jogadores de um time e os registros de bens imobiliários no cartório local (que mapeiam números de livro e de páginas para proprietários de bens imóveis).

Com as coleções, você utiliza estruturas de dados existentes, sem se preocupar com a maneira como são implementadas. Esse é um exemplo maravilhoso de reutilização de código. Você pode codificar mais rápido e esperar excelente desempenho, maximização da velocidade de execução e minimização do consumo de memória. Neste capítulo, discutimos as interfaces de estrutura de coleções que declaram as capacidades de cada tipo de coleção, as classes de implementação, os métodos que processam as coleções e os chamados **iteradores** que servem para "percorrer" as coleções. Este capítulo fornece uma introdução à estrutura de coleções. Para obter todos os detalhes, visite `java.sun.com/javase/6/docs/technotes/guides/collections/index.html`.

A estrutura de coleções do Java fornece componentes reutilizáveis prontos para utilização — você não precisa escrever suas próprias classes de coleção, mas pode se quiser. As coleções são padronizadas de modo que aplicativos possam compartilhá-las facilmente sem a preocupação com os detalhes de sua implementação. A estrutura de coleções encoraja ainda mais a capacidade de reutilização. À medida que se desenvolvem novas estruturas de dados e métodos que se encaixam nesse framework, uma grande base de programadores já estará familiarizada com as interfaces e métodos implementados por essas estruturas.

20.2 Visão geral das coleções

Uma **coleção** é uma estrutura de dados na realidade, um objeto que pode armazenar referências a outros objetos. Normalmente, as coleções contêm referências a objetos que são inteiramente do mesmo tipo. As interfaces de estrutura de coleções declaram as operações a ser realizadas genericamente em vários tipos de coleções. A Figura 20.1 lista algumas interfaces da estrutura de coleções. Várias implementações dessas interfaces são fornecidas dentro da estrutura. Você também pode fornecer implementações específicas para seus próprios requisitos.

Interface	Descrição
`Collection`	A interface-raiz na hierarquia de coleções a partir da qual as interfaces `Set`, `Queue` e `List` são derivadas.
`Set`	Uma coleção que não contém duplicatas.
`List`	Uma coleção ordenada que pode conter elementos duplicados.
`Map`	Associa chaves a valores e não pode conter chaves duplicadas.
`Queue`	Em geral, uma coleção primeiro a entrar, primeiro a sair que modela uma fila de espera; outras ordens podem ser especificadas.

Figura 20.1 | Algumas interfaces da estrutura de coleções.

As classes e interfaces da estrutura de coleções são membros do pacote `java.util`. Na próxima seção, iniciamos nossa discussão examinando as capacidades de estrutura de coleções para manipulação de array.

Nas primeiras versões do Java, as classes na estrutura de coleções armazenavam e manipulavam referências `Object`, permitindo que armazenássemos *qualquer* objeto em uma coleção. Um aspecto inconveniente de armazenar referências `Object` ocorre ao recuperá-las de uma coleção. Em geral, um programa precisa processar tipos específicos de objetos. Como resultado, as referências `Object` obtidas de uma coleção em geral precisam sofrer coerção (ser convertidas) em um tipo apropriado para permitirem que o programa processe os objetos corretamente.

No Java SE 5, a estrutura de coleções foi aprimorada com as capacidades dos genéricos que introduzimos no Capítulo 7 discutindo `ArrayList`s genéricas. Isso significa que é possível especificar o tipo exato que será armazenado em uma coleção. Você também obtém os benefícios da verificação de tipos em tempo de compilação — o compilador assegura que você utiliza tipos apropriados com sua coleção e, se não estiver utilizando, emite mensagens de erro em tempo de compilação. Além disso, uma vez que você especifica o tipo armazenado em uma coleção, qualquer referência que recuperar da coleção terá o tipo especificado. Isso elimina a necessidade de coerções de tipo explícitas que podem lançar `ClassCastException`s se o objeto referenciado não for do tipo apropriado. Além disso, as coleções genéricas são compatíveis com o código Java escrito antes de os genéricos serem introduzidos no Java SE 5.

20.3 Classes empacotadoras de tipo para tipos primitivos

Todo tipo primitivo (listado no Apêndice D, Tipos primitivos) tem uma **classe empacotadora de tipo** correspondente (no pacote `java.lang`). Essas classes chamam-se **`Boolean`**, **`Byte`**, **`Character`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** e **`Short`**. Toda classe empacotadora de tipo permite manipular valores de tipo primitivo como objetos. As estruturas de dados que reutilizamos ou desenvolvemos nos Capítulos 20–22 manipulam e compartilham objetos — elas não podem manipular variáveis de tipos primitivos. Contudo, podem manipular objetos das classes empacotadoras de tipos, porque toda classe, em última instância, deriva de `Object`.

Cada uma das classes empacotadoras de tipos numéricos — `Byte`, `Short`, `Integer`, `Long`, `Float` e `Double` — estende a classe `Number`. Além disso, as classes empacotadoras de tipo são classes `final`, então não é possível estendê-las.

Os tipos primitivos não têm métodos, então os métodos relacionados a um tipo primitivo estão localizados na classe empacotadora de tipo correspondente (por exemplo, o método `parseInt`, que converte uma `String` em um valor `int`, está localizado na classe `Integer`). Se precisar manipular um valor primitivo em seu programa, primeiro consulte a documentação para as classes empacotadoras de tipo — o método que você precisa pode já estar declarado.

20.4 Autoboxing e auto-unboxing

Antes do Java SE 5, se você quisesse inserir um valor primitivo em uma estrutura de dados, tinha de criar um novo objeto da classe empacotadora de tipos correspondente e depois inseri-lo na coleção. De maneira semelhante, se quisesse recuperar um objeto de uma classe empacotadora de tipo a partir de uma coleção e manipular seu valor primitivo, você teria de invocar um método no objeto para obter seu valor de tipo primitivo correspondente. Por exemplo, suponha que queira adicionar um `int` a um array que só armazena referências em objetos `Integer`. Antes do Java SE 5, seria necessário "empacotar" um valor `int` em um objeto `Integer` antes de adicionar o inteiro ao array e "desempacotar" o valor `int` do objeto `Integer` para recuperar o valor do array, como em

```
Integer[] integerArray = new Integer[ 5 ]; // cria integerArray


// atribui Integer 10 a integerArray[0]
integerArray[ 0 ] = new Integer( 10 );


// obtém valor int de Integer
int value = integerArray[ 0 ].intValue();
```

Observe que o valor primitivo `int` 10 é utilizado para inicializar um objeto `Integer`. Isso alcança o resultado desejado, mas exige código extra e é incômodo. Então precisamos invocar o método `intValue` da classe `Integer` para obter o valor `int` no objeto `Integer`.

O Java SE 5 introduziu duas novas conversões — boxing e unboxing — para simplificar a conversão entre valores de tipo primitivo e objetos empacotadores de tipo sem codificação adicional por parte do programador. Uma **conversão boxing** converte um valor de um tipo primitivo em um objeto da classe empacotadora de tipo correspondente. Uma **conversão unboxing** converte um objeto de uma classe empacotadora de tipo em um valor do tipo primitivo correspondente. Essas conversões podem ser realizadas automaticamente (chamadas **autoboxing** e **auto-unboxing**). Por exemplo, as instruções anteriores podem ser reescritas como

```
Integer[] integerArray = new Integer[ 5 ]; // cria integerArray
integerArray[ 0 ] = 10; // atribui Integer 10 a integerArray[0]
int value = integerArray[ 0 ]; // obtém valor int de Integer
```

Nesse caso, o autoboxing ocorre ao atribuir-se um valor `int` (10) a `integerArray[0]` `integerArray ]`, porque `integerArray` armazena referências a objetos `Integer`, não valores `int`. O auto-unboxing ocorre ao atribuir-se `integerArray[0]` à variável `int` `value`,

porque a variável `value` armazena um valor `value`, não uma referência a um objeto `Integer`. As conversões boxing também ocorrem em condições, que podem ser avaliadas como valores `boolean` primitivos ou objetos `Boolean`. Muitos dos exemplos nos Capítulos 20–22 utilizam essas conversões para armazenar valores primitivos em estruturas de dados e recuperá-los dessas estruturas.

20.5 Interface `Collection` e classe `Collections`

A interface **`Collection`** é a interface-raiz na hierarquia de coleções a partir da qual as interfaces `Set`, `Queue` e `List` são derivadas. A interface **`Set`** define uma coleção que não contém duplicatas. A interface **`Queue`** define uma coleção que representa uma fila de espera — em geral, as inserções são feitas na parte de trás de uma fila e as exclusões, na parte da frente, embora outras ordens possam ser especificadas. Discutimos `Queue` e `Set` nas seções 20.9–20.10. A interface `Collection` contém **operações de volume** (isto é, operações realizadas na coleção inteira) para operações como adicionar, limpar e comparar objetos (ou elementos) em uma coleção. Uma `Collection` também pode ser convertida em um array. Além disso, a interface `Collection` fornece um método que retorna um objeto **`Iterator`**, que permite a um programa percorrer a coleção e remover elementos da coleção durante a iteração. Discutimos a classe `Iterator` na Seção 20.6.1. Outros métodos da interface `Collection` permitem a um programa determinar o tamanho de uma coleção e se uma coleção está ou não vazia.

Observação de engenharia de software 20.1

`Collection` é comumente utilizada como um tipo de parâmetro nos métodos para permitir processamento polimórfico de todos os objetos que implementam a interface `Collection`.

Observação de engenharia de software 20.2

A maioria das implementações de coleção fornece um construtor que aceita um argumento `Collection`, permitindo, assim, que uma nova coleção a ser construída contenha os elementos da coleção especificada.

A classe **`Collections`** fornece os métodos `static` que pesquisam, classificam e realizam outras operações em coleções. A Seção 20.7 discute sobre os algoritmos que estão disponíveis na classe `Collections`. Discutimos também os **métodos empacotadores** (wrapper methods) da classe `Collections`, que permitem tratar uma coleção como uma coleção sincronizada (Seção 20.13) ou uma coleção não modificável (Seção 20.14). Essas coleções são úteis quando um cliente de uma classe precisa visualizar os elementos de uma coleção, mas elas não devem ter permissão de modificar a coleção pela adição ou remoção de elementos. As coleções sincronizadas são para utilização com uma poderosa capacidade chamada multithreading (discutido no Capítulo 26). O multithreading permite aos programas realizar operações paralelamente. Quando dois ou mais threads de um programa compartilham uma coleção, há a possibilidade de ocorrerem problemas. Como uma breve analogia, considere um cruzamento de trânsito. Não podemos permitir que todos os carros passem por um cruzamento ao mesmo tempo — se permitíssemos, aconteceriam acidentes. Por essa razão, são instalados semáforos para controlar o acesso ao cruzamento. De maneira semelhante, podemos sincronizar o acesso a uma coleção para assegurar que apenas um thread manipule a coleção por vez. Os métodos empacotadores de sincronização da classe `Collections` retornam as versões sincronizadas de coleções que podem ser compartilhadas entre threads em um programa.

20.6 Listas

Uma `List` (às vezes chamada de **sequência**) é uma `Collection` ordenada que pode conter elementos duplicados. Como os arrays, índices de `List` tem base em zero (isto é, o índice do primeiro elemento é zero). Além dos métodos herdados de `Collection`, `List` fornece métodos para manipular elementos via seus índices, manipular um intervalo especificado de elementos, procurar elementos e obter um **`ListIterator`** para acessar os elementos.

A interface `List` é implementada por várias classes, inclusive as classes **`ArrayList`**, **`LinkedList`** e **`Vector`**. O autoboxing ocorre quando você adiciona valores de tipo primitivo a objetos dessas classes, porque eles armazenam apenas referências a objetos. A classe `ArrayList` e a classe `Vector` são implementações de arrays redimensionáveis de `List`. Inserir um elemento entre elementos existentes de um `ArrayList` ou `Vector` é uma operação ineficiente — todos os elementos depois do elemento novo devem ser movidos de lugar, o que poderia ser uma operação cara em uma coleção com um grande número de elementos. Uma `LinkedList` permite inserção (ou remoção) eficiente de elementos no meio de uma coleção. Discutimos a arquitetura de listas encadeadas (vinculadas) no Capítulo 22.

As classes `ArrayList` e `Vector` têm comportamentos quase idênticos. A principal diferença entre elas é que `Vectors` são sincronizados por padrão, enquanto `ArrayLists` não o são. Além disso, a classe `Vector` é do Java 1.0, antes de a estrutura de coleções ser adicionada ao Java. Assim, `Vector` tem vários métodos que não fazem parte da interface `List` e que não são implementados na classe `ArrayList`, mas realizam tarefas idênticas. Por exemplo, os métodos `Vector` `addElement` e `add` acrescentam um elemento a um `Vector`, mas somente o método `add` é especificado na interface `List` e implementado por `ArrayList`. As coleções não sincronizadas fornecem melhor desempenho que as sincronizadas. Por essa razão, `ArrayList` em geral é preferida a `Vector` em programas que não compartilham uma coleção entre threads. Separadamente, a API de coleções do Java fornece os chamados empacotadores de sincronização (Seção 20.13) que podem ser utilizados para adicionar sincronização a coleções não sincronizadas, e várias coleções sincronizadas poderosas são disponibilizadas nas Java APIs.

Dica de desempenho 20.1

`ArrayLists` comportam-se como `Vectors` sem sincronização e, portanto, executam mais rápido que `Vectors` porque `ArrayLists` não tem o overhead de sincronização de thread.

Observação de engenharia de software 20.3

`LinkedLists` podem ser utilizadas para criar pilhas, filas e deques (double-ended queues — filas com dupla terminação). A estrutura de coleções fornece implementações de algumas dessas estruturas de dados.

As três subseções a seguir demonstram as capacidades de `List` e `Collection` com vários exemplos. A Seção 20.6.1 focaliza a remoção de elementos de uma `ArrayList` com um `Iterator`. A Seção 20.6.2 focaliza `ListIterator` e vários métodos específicos a `List` e `LinkedList`.

20.6.1 `ArrayList` e `Iterator`

A Figura 20.2 utiliza uma `ArrayList` (introduzida na Seção 7.14) para demonstrar várias capacidades da interface `Collection`. O programa coloca dois arrays `Color` em `ArrayLists` e utiliza um `Iterator` para remover elementos na segunda coleção `ArrayList` da primeira coleção `ArrayList`.

```
1  // Figura 20.2: CollectionTest.java
2  // A interface Collection demonstrada via um objeto ArrayList.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collection;
6  import java.util.Iterator;
7
8  public class CollectionTest
9  {
10    public static void main( String[] args )
11    {
12       // adiciona elementos no array colors a listar
13       String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
14       List< String > list = new ArrayList< String >();
15
16       for ( String color : colors )
17          list.add( color ); // adiciona a cor ao fim da lista
18
19       // adiciona elementos no array removeColors em removeList
20       String[] removeColors = { "RED", "WHITE", "BLUE" };
21       List< String > removeList = new ArrayList< String >();
22
23       for ( String color : removeColors )
24          removeList.add( color );
25
26       // gera a saída do conteúdo da lista
27       System.out.println( "ArrayList: " );
28
29       for ( int count = 0; count < list.size(); count++ )
30          System.out.printf( "%s ", list.get( count ) );
31
32       // remove da lista as cores contidas em removeList
33       removeColors( list, removeList );
34
35       // gera a saída do conteúdo da lista
36       System.out.println( "\n\nArrayList after calling removeColors: " );
37
38       for ( String color : list )
39          System.out.printf( "%s ", color );
40    } // fim de main
41
42    // remove cores especificadas em collection2 a partir de collection1
43    private static void removeColors(Collection< String > collection1,
44       Collection< String > collection2 )
45    {
```

```
46          // obtém o iterador
47          Iterator< String > iterator = collection1.iterator();
48
49          // loop enquanto a coleção tiver itens
50          while (iterator.hasNext() )
51          {
52             if ( collection2.contains( iterator.next() ) )
53                iterator.remove(); // remove Color atual
54          } // fim do while
55       } // fim do método removeColors
56    } // fim da classe CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Figura 20.2 | A interface `Collection` demonstrada via um objeto `ArrayList`.

As linhas 13 e 20 declaram e inicializam os arrays `String colors` e `removeColors`. As linhas 14 e 21 criam objetos `ArrayList<String>` e atribuem suas referências a variáveis `List<String> list` e `removeList`, respectivamente. Lembre-se de que `ArrayList` é uma classe genérica, portanto podemos especificar um argumento de tipo (`String` nesse caso) para indicar o tipo dos elementos em cada lista. Observe que referenciamos as `ArrayLists` nesse exemplo por meio de variáveis `List`. Isso torna nosso código mais flexível e mais fácil de modificar. Se decidíssemos mais tarde que as `LinkedLists` seriam mais apropriadas, precisaríamos modificar apenas as linhas 14 e 21 nas quais criamos os objetos `ArrayList`.

As linhas 16–17 preenchem `list` com `Strings` armazenadas no array `colors`, e as linhas 23–24 preenchem `removeList` com `Strings` armazenadas no array `removeColors` utilizando o **método `List add`**. As linhas 29–30 imprimem cada elemento de `list`. A linha 29 chama o **método `List size`** para obter o número de elementos na `ArrayList`. A linha 30 utiliza o **método `List get`** para recuperar valores individuais do elemento. Observe que as linhas 29–30 também podem ter utilizado a instrução `for` aprimorada (que demonstraremos com coleções em outros exemplos).

A linha 33 chama o método `removeColors` (linhas 43–55), passando `list` e `removeList` como argumentos. O método `removeColors` exclui as `Strings` em `removeList` a partir das `Strings` em `list`. As linhas 38–39 imprimem os elementos de `list` depois de `removeColors` completar sua tarefa.

O método `removeColors` declara dois parâmetros `Collection<String>` (linhas 43–44) que permitem que quaisquer das duas `Collections` contendo strings sejam passadas como argumentos para esse método. O método acessa os elementos da primeira `Collection` (`collection1`) via um `Iterator`. A linha 47 chama o **método `Collection iterator`** para obter um `Iterator` para `Collection`. Observe que as interfaces `Collection` e `Iterator` são tipos genéricos. A condição de continuação do loop (linha 50) chama o **método `Iterator hasNext`** para determinar se `Collection` contém mais elementos. O método `hasNext` retorna `true` se houver outro elemento e, `false`, caso contrário.

A condição `if` na linha 52 chama o **método `Iterator next`** para obter uma referência ao próximo elemento e, então, utiliza o método **`contains`** da segunda `Collection` (`collection2`) para determinar se `collection2` contém o elemento retornado por `next`. Se contiver, a linha 53 chama o **método `Iterator remove`** para remover o elemento da `Collection collection1`.

Erro comum de programação 20.1

Se uma coleção for modificada por um dos seus métodos depois que um iterador é criado para essa coleção, o iterador torna-se imediatamente inválido — as operações realizadas com o iterador depois desse ponto lançam `ConcurrentModificationExceptions`. Por essa razão, diz-se que os iteradores "respondem rápido".

20.6.2 LinkedList

A Figura 20.3 demonstra várias operações em `LinkedLists`. O programa cria duas `LinkedLists` de `Strings`. Os elementos de uma `List` são adicionados a outra. Então todas as `Strings` são convertidas em letras maiúsculas e um intervalo de elementos é excluído.

```
1  // Figura 20.3: ListTest.java
2  // Lists, LinkedLists e ListIterators.
3  import java.util.List;
4  import java.util.LinkedList;
5  import java.util.ListIterator;
```

```
6
7  public class ListTest
8  {
9     public static void main( String[] args )
10    {
11       // adiciona elementos colors a list1
12       String[] colors =
13          { "black", "yellow", "green", "blue", "violet", "silver" };
14       List< String > list1 = new LinkedList< String >();
15
16       for ( String color : colors )
17          list1.add( color );
18
19       // adiciona elementos colors2 à list2
20       String[] colors2 =
21          { "gold", "white", "brown", "blue", "gray", "silver" };
22       List< String > list2 = new LinkedList< String >();
23
24       for ( String color : colors2 )
25          list2.add( color );
26
27       list1.addAll( list2 ); // concatena as listas
28       list2 = null; // libera recursos
29       printList( list1 ); // imprime elementos list1
30
31       convertToUppercaseStrings( list1 ); // converte para string maiúscula
32       printList( list1 ); // imprime elementos list1
33
34       System.out.print( "\nDeleting elements 4 to 6..." );
35       removeItems( list1, 4, 7 ); // remove itens 4-6 da lista
36       printList( list1 ); // imprime elementos list1
37       printReversedList( list1 ); // imprime lista na ordem inversa
38    } // fim de main
39
40    // gera saída do conteúdo de List
41    private static void printList(List< String > list )
42    {
43       System.out.println( "\nlist: " );
44
45       for ( String color : list )
46          System.out.printf( "%s ", color );
47
48       System.out.println();
49    } // fim do método printList
50
51    // localiza objetos String e converte em letras maiúsculas
52    private static void convertToUppercaseStrings(List< String > list )
53    {
54       ListIterator< String > iterator = list.listIterator();
55
56       while (iterator.hasNext() )
57       {
58          String color = iterator.next(); // obtém o item
59          iterator.set( color.toUpperCase() ); // converte em letras maiúsculas
60       } // fim do while
61    } // fim do método convertToUppercaseStrings
62
63    // obtém sublista e utiliza método clear para excluir itens da sublista
64    private static void removeItems(List< String > list,
65       int start, int end )
66    {
67       list.subList( start, end ).clear(); // remove os itens
68    } // fim do método removeItems
69
70    // imprime lista invertida
71    private static void printReversedList(List< String > list )
72    {
73       ListIterator< String > iterator = list.listIterator( list.size() );
74
```

```
75          System.out.println( "\nReversed List:" );
76
77          // imprime lista na ordem inversa
78          while (iterator.hasPrevious() )
79             System.out.printf( "%s ", iterator.previous() );
80       } // fim do método printReversedList
81    } // fim da classe ListTest
```

```
list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Figura 20.3 | `Lists`, `LinkedLists` e `ListIterators`.

As linhas 14 e 22 criam as `LinkedLists` `list1` e `list2` do tipo `String`. `LinkedList` é uma classe genérica que tem um único parâmetro de tipo para o qual especificamos o argumento de tipo `String` nesse exemplo. As linhas 16–17 e 24–25 chamam o método `List` `add` para acrescentar elementos dos arrays `colors` e `colors2` no fim de `list1` e `list2`, respectivamente.

A linha 27 chama o **método `List` `addAll`** para acrescentar todos os elementos de `list2` ao final de `list1`. A linha 28 configura `list2` como `null`, então `LinkedList` que `list2` referenciou pode sofrer coleta de lixo. A linha 29 chama o método `printList` (linhas 41–49) para gerar saída do conteúdo de `list1`. A linha 31 chama o método `convertToUppercaseStrings` (linhas 52–61) para converter cada elemento `String` em letras maiúsculas, então a linha 32 chama novamente `printList` para exibir as `Strings` modificadas. A linha 35 chama o método `removeItems` (linhas 64–68) para remover os elementos que iniciam no índice `4` até, mas não incluindo, o índice `7` da lista. A linha 37 chama o método `printReversedList` (linhas 71–80) para imprimir a lista em ordem inversa.

Método `convertToUppercaseStrings`

O método `convertToUppercaseStrings` (linhas 52–61) converte de letras minúsculas em letras maiúsculas os elementos `String` do seu argumento `List`. A linha 54 chama o **método `List` `listIterator`** para obter o **iterador bidirecional** da `List` (isto é, um que pode percorrer um `List` para trás ou para frente). `ListIterator` também é uma classe genérica. Nesse exemplo, o `List-Iterator` referencia objetos `String`, porque o método `listIterator` é chamado em uma `List` de `Strings`. A linha 56 chama o método `hasNext` para determinar se a `List` contém outro elemento. A linha 58 obtém a próxima `String` na `List`. A linha 59 chama o **método `String` `toUpperCase`** para obter uma versão em letras maiúsculas da `String` e chama o **método `ListIterator` `set`** para substituir a `String` atual que `iterator` referencia pela `String` retornada pelo método `toUpperCase`. Como o método `toUpperCase`, o **método `String` `toLowerCase`** retorna uma versão em letras minúsculas da `String`.

Método `removeItems`

O método `removeItems` (linhas 64–68) remove um intervalo de itens da lista. A linha 67 chama o **método `List` `subList`** para obter uma parte da `List` (chamada de **sublista**). Esse é chamado **método de visualização de intervalo**, que permite ao programa examinar uma parte da lista. A sublista é simplesmente uma visualização na `List` em que `subList` é chamada. O método `subList` aceita como argumentos o índice inicial e o índice final para a sublista. O índice final não faz parte do intervalo da sublista. Nesse exemplo, a linha 35 passa `4` para o índice inicial e `7` para o índice final da `subList`. A sublista retornada é o conjunto de elementos com os índices de `4` a `6`. Em seguida, o programa chama o **método `List` `clear`** na sublista para remover os elementos da sublista da `List`. Qualquer alteração feita em uma sublista também será feita na `List` original.

Método `printReversedList`

O método `printReversedList` (linhas 71–80) imprime a lista de trás para frente. A linha 73 chama o método `List` `listIterator` com a posição inicial como um argumento (no nosso caso, o último elemento na lista) para obter um iterador bidirecional para a lista. O **método `List` `size`** retorna o número de itens na `List`. A condição `while` (linha 78) chama o **método `hasPrevious`** de **`ListIterator`** para determinar se há mais elementos ao percorrer a lista de trás para frente. A linha 79 chama o **método `previous`** de **`ListIterator`** a fim de obter o elemento anterior da lista e o envia para o fluxo de saída padrão.

Visualizações em coleções e o método `Arrays` `asList`

Um recurso importante da estrutura de coleções é a capacidade de manipular os elementos de um tipo de coleção (como um conjunto) por um tipo diferente de coleção (como uma lista), independentemente da implementação interna da coleção. O conjunto de métodos `public` pelo qual as coleções são manipuladas é chamado de **visualização**.

A classe `Arrays` fornece método `static` **`asList`** para examinar um array (às vezes chamado **array de apoio**) como uma coleção **`List`**. Uma visualização `List` permite manipular o array como se ele fosse uma lista. Isso é útil para adicionar os elementos em um array a uma coleção (por exemplo, um `LinkedList`) e para classificar elementos de array. O próximo exemplo demonstra como criar uma `LinkedList` com uma visualização `List` de um array, porque não podemos passar o array para um construtor `LinkedList`. A classificação de elementos de array com uma visualização `List` é demonstrada na Figura 20.7. Qualquer modificação feita pela visualização `List` altera o array, e qualquer modificação feita no array altera a visualização `List`. A única operação permitida na visualização retornada por `asList` é *set*, o que altera o valor da visualização e o array de apoio. Qualquer outra tentativa de alterar a visualização (como adicionar ou remover elementos) resulta em uma **`UnsupportedOperationException`**.

Visualizando arrays como `Lists` e convertendo `Lists` em arrays

A Figura 20.4 utiliza método `Arrays` `asList` para visualizar um array como uma `List` e utiliza o **método `List` `toArray`** para obter um array de uma coleção `LinkedList`. O programa chama o método `asList` para criar uma visualização `List` de um array, que é utilizada para inicializar um objeto `LinkedList`, então adiciona uma série de strings a uma `LinkedList` e chama o método `toArray` para obter um array contendo referências às `String`s.

```
1  // Figura 20.4: UsingToArray.java
2  // Visualizando arrays como Lists e convertendo Lists em arrays.
3  import java.util.LinkedList;
4  import java.util.Arrays;
5
6  public class UsingToArray
7  {
8     // cria uma LinkedList, adiciona elementos e converte em array
9     public static void main( String[] args )
10    {
11       String[] colors = { "black", "blue", "yellow" };
12
13       LinkedList< String > links =
14          new LinkedList< String >( Arrays.asList( colors ) );
15
16       links.addLast( "red" ); // adiciona como o último item
17       links.add( "pink" ); // adiciona ao final
18       links.add( 3, "green" ); // adiciona no terceiro índice
19       links.addFirst( "cyan" ); // adiciona como primeiro item
20
21       // obtém elementos LinkedList como um array
22       colors = links.toArray( new String[ links.size() ] );
23
24       System.out.println( "colors: " );
25
26       for ( String color : colors )
27          System.out.println( color );
28    } // fim de main
29 } // fim da classe UsingToArray
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Figura 20.4 | Visualizando arrays como `Lists` e convertendo `Lists` em arrays.

As linhas 13–14 constroem uma `LinkedList` de `String`s contendo os elementos do array `colors`. A linha 14 utiliza o método `Arrays` `asList` para retornar uma visualização `List` do array, em seguida utiliza isso para inicializar a `LinkedList` com seu construtor que recebe um `Collection` como um argumento (uma `List` *é uma* `Collection`). A linha 16 chama o **método `LinkedList` `addLast`** para adicionar `"red"` ao fim de `links`. As linhas 17–18 chamam o **método `LinkedList` `add`** para adicionar `"pink"` como o último elemento e `"green"` como o elemento no índice `3` (isto é, o quarto elemento). O método `addLast` (linha 16) funciona de modo idêntico ao método `add` (linha 17). A linha 19 chama o **método `LinkedList` `addFirst`** para adicionar `"cyan"` como o novo primeiro item na `LinkedList`. As operações `add` são permitidas porque operam no objeto `LinkedList`, não na visualização retornada por `asList`. [*Nota*: Quando `"cyan"` é adicionado como o primeiro elemento, `"green"` torna-se o quinto elemento na `LinkedList`.]

A linha 22 chama o método `toArray` da interface `List` para obter um array `String` a partir de `links`. O array é uma cópia dos elementos da lista — modificar o conteúdo do array não modifica a lista. O array passado para o método `toArray` é do mesmo tipo que você gostaria que o método `toArray` retornasse. Se o número de elementos no array for maior ou igual ao número de elementos na `LinkedList`, `toArray` copia os elementos da lista em seu argumento de array e retorna esse array. Se a `LinkedList` tiver mais elementos que o número de elementos no array passado para `toArray`, `toArray` aloca um novo array do mesmo tipo que ele recebe como um argumento, copia os elementos da lista no novo array e retorna o novo array.

Erro comum de programação 20.2

Passar um array que contém dados para `toArray` pode causar erros de lógica. Se o número de elementos no array for menor que o número de elementos na lista em que `toArray` é chamado, um novo array é alocado para armazenar os elementos da lista — sem preservar os elementos do argumento de array. Se o número de elementos no array for maior que o número de elementos na lista, os elementos do array (iniciando no índice zero) serão sobrescritos pelos elementos da lista. Os elementos do array que não são sobrescritos retêm seus valores.

20.7 Métodos de coleções

A classe `Collection` fornece vários algoritmos de alto desempenho para manipular elementos de coleção. Os algoritmos (Figura 20.5) são implementados como métodos `static`. Os métodos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operam em `Lists`. Os métodos `min`, `max`, `addAll`, `frequency` e `disjoint` operam em `Collections`.

Método	Descrição
`sort`	Classifica os elementos de uma `List`.
`binarySearch`	Localiza um objeto em uma `List`.
`reverse`	Inverte os elementos de uma `List`.
`shuffle`	Ordena aleatoriamente os elementos de uma `List`.
`fill`	Configura todo elemento `List` para referir-se a um objeto especificado.
`copy`	Copia referências de uma `List` em outra.
`min`	Retorna o menor elemento em uma `Collection`.
`max`	Retorna o maior elemento em uma `Collection`.
`addAll`	Acrescenta todos os elementos em um array a uma `Collection`.
`frequency`	Calcula quantos elementos da coleção são iguais ao elemento especificado.
`disjoint`	Determina se duas coleções não têm nenhum elemento em comum.

Figura 20.5 | Métodos `Collections`.

Observação de engenharia de software 20.4

Os métodos da estrutura de coleções são polimórficos. Isto é, cada um deles pode operar em objetos que implementam interfaces específicas, independentemente da implementação subjacente.

20.7.1 Método `sort`

O **método `sort`** classifica os elementos de uma `List`, que deve implementar a **interface `Comparable`**. A ordem é determinada pela ordem natural do tipo dos elementos como implementado por um método `compareTo`. O método `compareTo` é declarado na interface `Comparable` e às vezes é chamado **método natural de comparação**. A chamada `sort` pode especificar como um segundo argumento um objeto **`Comparator`** que determina uma ordem alternativa dos elementos.

Classificando na ordem crescente

A Figura 20.6 utiliza o método `Collections` `sort` para ordenar os elementos de uma `List` em ordem crescente (linha 17). Lembre-se de que `List` é um tipo genérico e aceita como argumento o tipo de elemento da lista — a linha 14 cria `list` como uma `List` de `String`s. Observe que as linhas 15 e 20 utilizam, cada uma, uma chamada implícita para o método `toString` da `list` para gerar saída do conteúdo da lista no formato mostrado na segunda e quarta linhas da saída.

```
1  // Figura 20.6: Sort1.java
2  // Método Collections sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort1
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Cria e exibe uma lista que contém os elementos do array de ternos
14       List< String > list = Arrays.asList( suits ); // cria List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       Collections.sort( list ); // classifica ArrayList
18
19       // gera a saída da lista
20       System.out.printf( "Sorted array elements: %s\n", list );
21    } // fim de main
22 } // fim da classe Sort1
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Figura 20.6 | Método `Collections sort`.

Classificando em ordem decrescente

A Figura 20.7 classifica a mesma lista das strings utilizadas na Figura 20.6 em ordem decrescente. O exemplo introduz a interface `Comparator`, que é utilizado para classificar elementos de uma `Collection` em uma ordem diferente. A linha 18 chama o método `sort` de `Collections` para ordenar a `List` em ordem decrescente. O **método** `static` **`Collections reverseOrder`** retorna um objeto `Comparator` que ordena os elementos da coleção na ordem inversa.

```
1  // Figura 20.7: Sort2.java
2  // Utilizando um objeto Comparator com o método sort.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Sort2
8  {
9     public static void main( String[] args )
10    {
11       String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13       // Cria e exibe uma lista que contém os elementos do array de ternos
14       List< String > list = Arrays.asList( suits ); // cria List
15       System.out.printf( "Unsorted array elements: %s\n", list );
16
17       // classifica em ordem decrescente utilizando um comparador
18       Collections.sort( list, Collections.reverseOrder() );
19
20       // gera a saída de elementos List
21       System.out.printf( "Sorted list elements: %s\n", list );
22    } // fim de main
23 } // fim da classe Sort2
```

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted list elements: [Spades, Hearts, Diamonds, Clubs]
```

Figura 20.7 | O método `Collections sort` com um objeto `Comparator`.

Classificando com um Comparator

A Figura 20.8 cria uma classe `Comparator` personalizada, chamada `TimeComparator`, que implementa a interface `Comparator` para comparar dois objetos `Time2`. A classe `Time2`, declarada na Figura 8.5, representa o tempo com horas, minutos e segundos.

```
1  // Figura 20.8: TimeComparator.java
2  // Classe Comparator personalizada que compara dois objetos Time2.
3  import java.util.Comparator;
4
5  public class TimeComparator implements Comparator< Time2 >
6  {
7     public int compare(Time2 time1, Time2 time2 )
8     {
9        int hourCompare = time1.getHour() - time2.getHour(); // compara hora
10
11       // testa a primeira hora
12       if ( hourCompare != 0 )
13          return hourCompare;
14
15       int minuteCompare =
16          time1.getMinute() - time2.getMinute(); // compara minuto
17
18       // então testa o minuto
19       if ( minuteCompare != 0 )
20          return minuteCompare;
21
22       int secondCompare =
23          time1.getSecond() - time2.getSecond(); // compara segundo
24
25       return secondCompare; // retorna o resultado da comparação de segundos
26    } // fim do método compare
27 } // fim da classe TimeComparator
```

Figura 20.8 | Classe `Comparator` personalizada que compara dois objetos `Time2`.

A classe `TimeComparator` implementa a interface `Comparator`, um tipo genérico que aceita um argumento (nesse caso `Time2`). Uma classe que implementa `Comparator` deve declarar um método `compare` que recebe dois argumentos e retorna um número inteiro negativo se o primeiro argumento for menor que o segundo, 0 se os argumentos forem iguais ou um número inteiro positivo se o primeiro argumento for maior do que o segundo. O método `compare` (linhas 7–26) realiza comparações entre objetos `Time2`. A linha 9 compara as duas horas dos objetos `Time2`. Se as horas forem diferentes (linha 12), então retornamos esse valor. Se esse valor for positivo, então a primeira hora é maior que a segunda e o primeiro tempo é maior que o segundo. Se esse valor for negativo, então a primeira hora é menor que a segunda e o primeiro tempo é menor que o segundo. Se esse valor for zero, as horas serão as mesmas e devemos testar os minutos (e talvez os segundos) para determinar que tempo é maior.

A Figura 20.9 classifica uma lista que utiliza a classe `Comparator` `TimeComparator` personalizada. A linha 11 cria uma `ArrayList` de objetos `Time2`. Lembre-se de que `ArrayList` e `List` são tipos genéricos e aceitam um argumento de tipo que especifica o tipo de elemento da coleção. As linhas 13–17 criam cinco objetos `Time2` e os adicionam a essa lista. A linha 23 chama o método `sort`, passando para ele um objeto de nossa classe `TimeComparator` (Figura 20.8).

```
1  // Figura 20.9: Sort3.java
2  // Método Collections sort com um objeto Comparator personalizado.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public static void main( String[] args )
10    {
11       List< Time2 > list = new ArrayList< Time2 >(); // cria List
12
13       list.add( new Time2(  6, 24, 34 ) );
14       list.add( new Time2( 18, 14, 58 ) );
15       list.add( new Time2(  6, 05, 34 ) );
```

```
16          list.add( new Time2( 12, 14, 58 ) );
17          list.add( new Time2(  6, 24, 22 ) );
18
19          // gera a saída de elementos List
20          System.out.printf( "Unsorted array elements:\n%s\n", list );
21
22          // classifica em ordem utilizando um comparador
23          Collections.sort( list, new TimeComparator() );
24
25          // gera a saída de elementos List
26          System.out.printf( "Sorted list elements:\n%s\n", list );
27       } // fim de main
28    } // fim da classe Sort3
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Figura 20.9 | Método `Collections sort` com um objeto `Comparator` personalizado.

20.7.2 Método shuffle

O método **`shuffle`** ordena aleatoriamente os elementos de uma `List`. No Capítulo 7, apresentamos uma simulação do embaralhamento e distribuição de cartas que utiliza um loop para embaralhar as cartas do baralho. Na Figura 20.10, utilizamos o método `shuffle` para embaralhar as cartas do baralho de objetos `Card` que poderiam ser utilizados em um simulador de jogo de cartas.

A classe `Card` (linhas 8–41) representa uma carta do baralho. Cada `Card` tem uma face e um naipe. As linhas 10–12 declaram dois tipos `enum`— `Face` e `Suit` — que representam a face e o naipe da carta, respectivamente. O método `toString` (linhas 37–40) retorna uma `String` que contém a face e o naipe da `Card` separados pela string `" of "`. Quando uma constante `enum` for convertida em uma string, o identificador da constante é utilizado como a representação de string. Normalmente utilizaríamos todas as letras maiúsculas para constantes `enum`. Nesse exemplo, escolhemos utilizar letras maiúsculas apenas para a letra inicial de cada constante `enum` porque queremos que a carta seja exibida com letras iniciais maiúsculas para a face e naipe (por exemplo, `"Ace of Spades"`).

```
1  // Figura 20.10: DeckOfCards.java
2  // Embaralhando e distribuindo cartas com o método Collections shuffle.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  // classe para representar uma carta de um baralho
8  class Card
9  {
10    public static enum Face { Ace, Deuce, Three, Four, Five, Six,
11       Seven, Eight, Nine, Ten, Jack, Queen, King  };
12    public static enum Suit { Clubs, Diamonds, Hearts, Spades };
13
14    private final Face face; // face da carta
15    private final Suit suit; // naipe da carta
16
17    // construtor de dois argumentos
18    public Card( Face cardFace, Suit cardSuit )
19    {
20        face = cardFace; // inicializa face da carta
21        suit = cardSuit; // inicializa naipe da carta
22    } // fim do construtor Card de dois argumentos
23
24    // retorna a face da carta
25    public Face getFace()
26    {
27       return face;
28    } // fim do método getFace
29
30    // retorna o naipe da carta
31    public Suit getSuit()
```

```
32       {
33          return suit;
34       } // fim do método getSuit
35
36       // retorna a representação String de Card
37       public String toString()
38       {
39          return String.format( "%s of %s", face, suit );
40       } // fim do método toString
41    } // fim da classe Card
42
43    // declaração da classe DeckOfCards
44    public class DeckOfCards
45    {
46       private List< Card > list; // declara List que armazenará cartas
47
48       // configura o baralho de cartas e embaralha
49       public DeckOfCards()
50       {
51          Card[] deck = new Card[ 52 ];
52          int count = 0; // número de cartas
53
54          // preenche baralho com objetos Card
55          for (Card.Suit suit : Card.Suit.values() )
56          {
57             for (Card.Face face : Card.Face.values() )
58             {
59                deck[ count ] = new Card( face, suit );
60                ++count;
61             } // for final
62          } // for final
63
64          list = Arrays.asList( deck ); // obtém List
65          Collections.shuffle( list );  // embaralha as cartas
66       } // fim do construtor DeckOfCards
67
68       // gera a saída de baralho
69       public void printCards()
70       {
71          // exibe 52 cartas em duas colunas
72          for ( int i = 0; i < list.size(); i++ )
73             System.out.printf( "%-19s%s", list.get( i ),
74                ( ( i + 1 ) % 4 == 0 ) ? "\n" : "" );
75       } // fim do método printCards
76
77       public static void main( String[] args )
78       {
79          DeckOfCards cards = new DeckOfCards();
80          cards.printCards();
81       } // fim de main
82    } // fim da classe DeckOfCards
```

```
Deuce of Clubs     Six of Spades      Nine of Diamonds   Ten of Hearts
Three of Diamonds  Five of Clubs      Deuce of Diamonds  Seven of Clubs
Three of Spades    Six of Diamonds    King of Clubs      Jack of Hearts
Ten of Spades      King of Diamonds   Eight of Spades    Six of Hearts
Nine of Clubs      Ten of Diamonds    Eight of Diamonds  Eight of Hearts
Ten of Clubs       Five of Hearts     Ace of Clubs       Deuce of Hearts
Queen of Diamonds  Ace of Diamonds    Four of Clubs      Nine of Hearts
Ace of Spades      Deuce of Spades    Ace of Hearts      Jack of Diamonds
Seven of Diamonds  Three of Hearts    Four of Spades     Four of Diamonds
Seven of Spades    King of Hearts     Seven of Hearts    Five of Diamonds
Eight of Clubs     Three of Clubs     Queen of Clubs     Queen of Spades
Six of Clubs       Nine of Spades     Four of Hearts     Jack of Clubs
Five of Spades     King of Spades     Jack of Spades     Queen of Hearts
```

Figura 20.10 | Embaralhamento e distribuição de cartas com o método `Collections shuffle`.

As linhas 55–62 preenchem o array `deck` com cartas que têm combinações únicas de face e de naipe. Tanto `Face` como `Suit` são os tipos `public static enum` da classe `Card`. Para utilizar esses tipos `enum` fora da classe `Card`, você deve qualificar o nome de tipo de cada `enum` com o nome da classe em que ele reside (isto é, `Card`) e um ponto (`.`) separador. Por isso, as linhas 55 e 57 utilizam `Card.Suit` e `Card.Face` para declarar as variáveis de controle das instruções `for`. Lembre-se de que o método `values` de um tipo `enum` retorna um array que contém todas as constantes do tipo `enum`. As linhas 55–62 utilizam instruções `for` aprimoradas para construir 52 novas `Card`s.

O embaralhamento ocorre na linha 65, que chama o método `static` `shuffle` da classe `Collections`- para embaralhar os elementos do array. O método `shuffle` exige um argumento `List`, assim, devemos obter uma visualização `List` do array antes que possamos embaralhá-lo. A linha 64 invoca o método `static` `asList` da classe `Arrays` para obter uma visualização `List` do array `deck`.

O método `printCards` (linhas 69–75) exibe o baralho de cartas em quatro colunas. Em cada iteração do loop, as linhas 73–74 geram saída de uma carta alinhada à esquerda em um campo de 19 caracteres seguido por um caractere de nova linha ou uma string vazia com base no número de cartas enviado para saída até agora. Se o número de cartas for divisível por 4, o programa imprime uma nova linha; caso contrário, uma string vazia.

20.7.3 Métodos `reverse`, `fill`, `copy`, `max` e `min`

A classe `Collections` fornece métodos para inverter, preencher e copiar `List`s. O **método `Collections` `reverse`** inverte a ordem dos elementos em uma `List` e o **método `fill`** sobrescreve elementos em uma `List` com um valor especificado. A operação `fill` é útil para reinicializar uma `List`. O **método `copy`** recebe dois argumentos — uma `List` de destino e uma `List` de origem. Cada elemento da `List` de origem é copiado para a `List` de destino. A `List` de destino deve ser pelo menos tão longa quanto a `List` de origem; caso contrário, uma `IndexOutOfBoundsException` ocorre. Se a `List` de destino é mais longa, os elementos não sobrescritos permanecem inalterados.

Todos os métodos que vimos até aqui operam em `List`s. Os métodos **`min`** e **`max`**, cada um, operam em qualquer `Collection`. O método `min` retorna o menor elemento. Esses dois métodos podem ser chamados com um objeto `Comparator` como um segundo argumento para realizar comparações personalizadas de objetos, como o `TimeComparator` na Figura 20.9. A Figura 20.11 demonstra os métodos `reverse`, `fill`, `copy`, `min` e `max`.

```
1  // Figura 20.11: Algorithms1.java
2  // Métodos Collections reverse, fill, copy, max e min.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6
7  public class Algorithms1
8  {
9     public static void main( String[] args )
10    {
11       // cria e exibe uma List< Character >
12       Character[] letters = { 'P', 'C', 'M' };
13       List< Character > list = Arrays.asList( letters ); // obtém List
14       System.out.println( "list contains: " );
15       output( list );
16
17       // inverte e exibe a List< Character >
18       Collections.reverse( list ); // inverte a ordem dos elementos
19       System.out.println( "\nAfter calling reverse, list contains: " );
20       output( list );
21
22       // cria copyList de um array de 3 caracteres
23       Character[] lettersCopy = new Character[ 3 ];
24       List< Character > copyList = Arrays.asList( lettersCopy );
25
26       // copia os conteúdos de list para copyList
27       Collections.copy( copyList, list );
28       System.out.println( "\nAfter copying, copyList contains: " );
29       output( copyList );
30
31       // preenche a lista com Rs
32       Collections.fill( list, 'R' );
33       System.out.println( "\nAfter calling fill, list contains: " );
34       output( list );
35    } // fim de main
36
37    // envia informações de List para saída
38    private static void output( List< Character > listRef )
39    {
```

```
40          System.out.print( "The list is: " );
41
42          for ( Character element : listRef )
43             System.out.printf( "%s ", element );
44
45          System.out.printf( "\nMax: %s", Collections.max( listRef ) );
46          System.out.printf( "  Min: %s\n", Collections.min( listRef ) );
47       } // fim do método output
48    } // fim da classe Algorithms1
```

```
list contains:
The list is: P C M
Max: P  Min: C

After calling reverse, list contains:
The list is: M C P
Max: P  Min: C

After copying, copyList contains:
The list is: M C P
Max: P  Min: C

After calling fill, list contains:
The list is: R R R
Max: R  Min: R
```

Figura 20.11 | Os métodos `Collections` `reverse`, `fill`, `copy`, `max` e `min`.

A linha 13 cria a variável `List<Character>` `list` e a inicializa com uma visualização `List` do array `Character` `letters`. As linhas 14–15 imprimem o conteúdo atual da `List`. A linha 18 chama o método `Collections` `reverse` para inverter a ordem de `list`. O método `reverse` aceita um argumento `List`. Como `list` é uma visualização `List` do array `letters`, os elementos do array estão agora em ordem inversa. A saída do conteúdo invertido é gerada nas linhas 19–20. A linha 27 utiliza o método `Collections` `copy` para copiar os elementos de `list` para `copyList`. As alterações na `copyList` não alteram `letters`, porque `copyList` é uma `List` separada que não é uma visualização `List` do array `letters`. O método `copy` exige dois argumentos `List`— a `List` de destino e a `List` de origem. A linha 32 chama o método `Collections` `fill` para colocar o caractere `'R'` em cada elemento `list`. Como `list` é uma visualização `List` do array `letters`, essa operação transforma cada elemento em `letters` para `'R'`. O método `fill` exige uma `List` do primeiro argumento e um `Object` para o segundo argumento — nesse caso, o `Object` é a versão do caractere `'R'` após a operação de boxing. As linhas 45–46 chamam os métodos `Collections` `max` e `min` para localizar o maior e o menor elemento de uma `Collection`, respectivamente. Lembre-se de que a interface `List` estende a interface `Collection`, portanto, uma `List` *é uma* `Collection`.

20.7.4 Método `binarySearch`

Na Seção 19.2.2, estudamos o algoritmo de pesquisa binária de alta velocidade. Esse algoritmo é incorporado na estrutura de coleções do Java como um método `static` **`Collections binarySearch`**, que localiza um objeto em uma `List` (por exemplo, uma `LinkedList` ou uma `ArrayList`). Se o objeto for encontrado, seu índice é retornado. Se o objeto não for localizado, `binarySearch` retorna um valor negativo. O método `binarySearch` determina esse valor negativo primeiro calculando o ponto de inserção e tornando seu sinal negativo. Então, `binarySearch` subtrai 1 do ponto de inserção para obter o valor de retorno, que garante que o método `binarySearch` retorna números positivos (>=0) se e somente se o objeto for localizado. Se múltiplos elementos na lista corresponderem à chave de pesquisa, não é garantido que um será localizado primeiro. A Figura 20.12 utiliza o método `binarySearch` para procurar uma série de strings em uma `ArrayList`.

```
1  // Figura 20.12: BinarySearchTest.java
2  // Método Collections binarySearch.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.ArrayList;
7
8  public class BinarySearchTest
9  {
10    public static void main( String[] args )
11    {
12       // cria um ArrayList< String > do conteúdo do array colors
```

```
13          String[] colors = { "red", "white", "blue", "black", "yellow",
14             "purple", "tan", "pink" };
15          List< String > list =
16             new ArrayList< String >( Arrays.asList( colors ) );
17
18          Collections.sort( list ); // classifica a ArrayList
19          System.out.printf( "Sorted ArrayList: %s\n", list );
20
21          // pesquisa vários valores na lista
22          printSearchResults( list, colors[ 3 ] ); // primeiro item
23          printSearchResults( list, colors[ 0 ] ); // item do meio
24          printSearchResults( list, colors[ 7 ] ); // último item
25          printSearchResults( list, "aqua" ); // abaixo do mais baixo
26          printSearchResults( list, "gray" ); // não existe
27          printSearchResults( list, "teal" ); // não existe
28       } // fim de main
29
30       // realiza pesquisa e exibe o resultado
31       private static void printSearchResults(
32          List< String > list, String key )
33       {
34          int result = 0;
35
36          System.out.printf( "\nSearching for: %s\n", key );
37          result = Collections.binarySearch( list, key );
38
39          if ( result >= 0 )
40             System.out.printf( "Found at index %d\n", result );
41          else
42             System.out.printf( "Not Found (%d)\n",result );
43       } // fim do método printSearchResults
44    } // fim da classe BinarySearchTest
```

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]

Searching for: black
Found at index 0

Searching for: red
Found at index 4

Searching for: pink
Found at index 2
```

```
Searching for: aqua
Not Found (-1)

Searching for: gray
Not Found (-3)

Searching for: teal
Not Found (-7)
```

Figura 20.12 | Método `Collections` `binarySearch`.

As linhas 15–16 inicializam `list` com uma `ArrayList` contendo uma cópia dos elementos no array `colors`. O método `Collections` `binarySearch` espera que os elementos do argumento `List` sejam classificados em ordem crescente, então, a linha 18 utiliza o método `Collections` `sort` para classificar a lista. Se os elementos do argumento `List` não forem classificados, o resultado de utilizar `binarySearch` é indefinido. A linha 19 gera saída da lista classificada. As linhas 22–27 chamam o método `printSearchResults` (linhas 31–43) para realizar pesquisas e gerar saída dos resultados. A linha 37 chama o método `Collections` `binarySearch` para procurar pela `key` especificada em `list`. O método `binarySearch` aceita uma `List` como o primeiro argumento e um `Object` como o segundo argumento. As linhas 39–42 geram saída dos resultados da pesquisa. Uma versão sobrecarregada de `binarySearch` aceita um objeto `Comparator` como seu terceiro argumento, que especifica como `binarySearch` deve comparar a chave de pesquisa com os elementos da `List`

20.7.5 Métodos `addAll`, `frequency` e `disjoint`

A classe `Collections` também fornece os métodos `addAll`, `frequency` e `disjoint`. O **método `Collections` `addAll`** aceita dois argumentos — uma `Collection` na qual inserir o(s) novo(s) elemento(s) e um array que fornece elementos a ser inseridos. O **método**

`Collections frequency` aceita dois argumentos — um `Collection` a ser pesquisado e um `Object` a ser procurado na coleção. O método `frequency` retorna o número de vezes que o segundo argumento aparece na coleção. O **método `Collections disjoint`** aceita duas `Collections` e retorna `true` se elas não tiverem nenhum elemento em comum. A Figura 20.13 demonstra o uso de métodos `addAll`, `frequency` e `disjoint`.

```
1  // Figura 20.13: Algorithms2.java
2  // Métodos Collections addAll, frequency e disjoint.
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Arrays;
6  import java.util.Collections;
7
8  public class Algorithms2
9  {
10    public static void main( String[] args )
11    {
12       // inicializa list1 e list2
13       String[] colors = { "red", "white", "yellow", "blue" };
14       List< String > list1 = Arrays.asList( colors );
15       ArrayList< String > list2 = new ArrayList< String >();
16
17       list2.add( "black" ); // adiciona "black" ao fim de list2
18       list2.add( "red" ); // adiciona "red" ao fim de list2
19       list2.add( "green" ); // adiciona "green" ao fim de list2
20
21       System.out.print( "Before addAll, list2 contains: " );
22
23       // exibe elementos em list2
24       for ( String s : list2 )
25          System.out.printf( "%s ", s );
26
27       Collections.addAll( list2, colors ); // adiciona Strings colors a list2
28
29       System.out.print( "\nAfter addAll, list2 contains: " );
30
31       // exibe elementos em list2
32       for ( String s : list2 )
33          System.out.printf( "%s ", s );
34
35       // obtém frequência de "red"
36       int frequency = Collections.frequency( list2, “red” );
37       System.out.printf(
38          “\nFrequency of red in list2: %d\n”, frequency );
39
40       // verifica se list1 e list2 têm elementos em comum
41       boolean disjoint = Collections.disjoint( list1, list2 );
42
43       System.out.printf( "list1 and list2 %s elements in common\n",
44          ( disjoint ? "do not have" : "have" ) );
45    } // fim de main
46 } // fim da classe Algorithms2
```

```
Before addAll, list2 contains: black red green
After addAll, list2 contains: black red green red white yellow blue
Frequency of red in list2: 2
list1 and list2 have elements in common
```

Figura 20.13 | Métodos `Collections addAll`, `frequency` e `disjoint`.

A linha 14 inicializa `list1` com elementos no array `colors`, e as linhas 17–19 adiciona `Strings` `"black"`, `"red"` e `"green"` à `list2`. A linha 27 invoca o método `addAll` para adicionar elementos no array `colors` para `list2`. A linha 36 obtém a frequência de `String` `"red"` em `list2` para utilizar o método `frequency`. A linha 41 invoca o método `disjoint` para testar se `Collections` `list1` e `list2` têm elementos em comum, o que, nesse exemplo, elas apresentam.

20.8 Classe Stack do pacote java.util

Introduzimos o conceito de uma pilha na Seção 6.6 quando discutimos a pilha de chamada do método. No Capítulo 22, Estruturas de dados genéricos personalizados, aprenderemos como construir estruturas de dados, incluindo listas encadeadas, pilhas, filas e árvores. Em um mundo de reutilização de software, em vez de construir estruturas de dados à medida que precisamos delas, costumamos tirar proveito de estruturas de dados existentes. Nesta seção, investigamos a classe **Stack** do pacote de utilitários Java (`java.util`).

A classe `Stack` estende a classe `Vector` para implementar uma estrutura de dados de pilha. Como a classe `Stack` estende a classe `Vector`, a interface `public` inteira da classe `Vector` está disponível para clientes da classe `Stack`. A Figura 20.14 demonstra vários métodos `Stack`. Para os detalhes da classe `Stack`, visite `java.sun.com/javase/6/docs/api/java/util/Stack.html`.

Dica de prevenção de erro 20.1

Como `Stack` estende `Vector`, todos os métodos `public` `Vector` podem ser chamados em objetos `Stack`, mesmo se os métodos não representarem operações de pilha convencionais. Por exemplo, o método `Vector` `add` pode ser utilizado para inserir um elemento em qualquer lugar em uma pilha — uma operação que poderia "corromper" a pilha. Ao manipular uma `Stack`, somente os métodos `push` e `pop` devem ser utilizados para adicionar elementos à `Stack` e remover elementos dela, respectivamente.

```
1  // Figura 20.14: Stacktest.java
2  // Classe Stack do pacote java.util.
3  import java.util.Stack;
4  import java.util.EmptyStackException;
5
6  public class StackTest
7  {
8     public static void main( String[] args )
9     {
10       Stack< Number > stack = new Stack< Number >(); // cria uma pilha
11
12       // utiliza método push
13       stack.push( 12L ); // adiciona valor long 12L
14       System.out.println( "Pushed 12L" );
15       printStack( stack );
16       stack.push( 34567 ); // adiciona valor int 34567
17       System.out.println( "Pushed 34567" );
18       printStack( stack );
19       stack.push( 1.0F ); // adiciona valor float 1.0F
20       System.out.println( "Pushed 1.0F" );
21       printStack( stack );
22       stack.push( 1234.5678 ); // adiciona valor double 1234.5678
23       System.out.println( "Pushed 1234.5678 " );
24       printStack( stack );
25
26       // remove itens de pilha
27       try
28       {
29          Number removedObject = null;
30
31          // remove elementos da pilha
32          while ( true )
33          {
34             removedObject = stack.pop(); // utiliza método pop
35             System.out.printf( "Popped %s\n", removedObject );
36             printStack( stack );
37          } // fim do while
38       } // fim do try
39       catch ( EmptyStackException emptyStackException )
40       {
41          emptyStackException.printStackTrace();
42       } // fim do catch
43    } // fim de main
44
45    // exibe o conteúdo de Stack
46    private static void printStack( Stack< Number > stack )
47    {
```

```
48          if (stack.isEmpty() )
49             System.out.println( "stack is empty\n" ); // a pilha está vazia
50          else // a pilha não está vazia
51             System.out.printf( "stack contains: %s (top)\n", stack );
52       } // fim do método printStack
53    } // fim da classe StackTest
```

```
Pushed 12L
stack contains: [12] (top)
Pushed 34567
stack contains: [12, 34567] (top)
Pushed 1.0F
stack contains: [12, 34567, 1.0] (top)
Pushed 1234.5678
stack contains: [12, 34567, 1.0, 1234.5678] (top)
Popped 1234.5678
stack contains: [12, 34567, 1.0] (top)
Popped 1.0
stack contains: [12, 34567] (top)
Popped 34567
stack contains: [12] (top)
Popped 12
stack is empty

java.util.EmptyStackException
        at java.util.Stack.peek(Unknown Source)
        at java.util.Stack.pop(Unknown Source)
        at StackTest.main(StackTest.java:34)
```

Figura 20.14 | Classe `Stack` do pacote `java.util`.

A linha 10 cria uma `Stack` vazia de `Numbers`. A classe `Number` (no pacote `java.lang`) é a superclasse das classes empacotadoras de tipo de tipos numéricos primitivos (por exemplo, `Integer`, `Double`). Criando uma `Stack` de `Numbers`, os objetos de qualquer classe que estenda `Number` podem ser adicionados à `Stack`. As linhas 13, 16, 19 e 22 chamam o método `Stack` **`push`** para adicionar um objeto `Number` na parte superior da pilha. Observe os literais `12L` (linha 13) e `1.0F` (linha 19). Qualquer literal inteiro que tenha o **sufixo `L`** é um valor `long`. Um literal de inteiro sem sufixo é um valor `int`. De maneira semelhante, qualquer literal de ponto flutuante que tenha o **sufixo `F`** é um valor `float`. Um literal de ponto flutuante sem sufixo é um valor `double`. Você pode aprender mais sobre literais numéricos na *Java Language Specification* em `java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.8.1`.

Um loop infinito (linhas 32–37) chama o **método `Stack` `pop`** para remover o elemento na parte superior da pilha. O método retorna uma referência `Number` ao elemento removido. Se não houver nenhum elemento na `Stack`, o método `pop` lança uma **`EmptyStackException`**, que termina o loop. A classe `Stack` também declara o **método `peek`**. Esse método retorna o elemento na parte superior da pilha sem remover o elemento da pilha.

O método `printStack` (linhas 46–52) exibe o conteúdo da pilha. A parte superior atual da pilha (o último valor adicionado à pilha) é o primeiro valor impresso. A linha 48 chama o **método `Stack` `isEmpty`** (herdado por `Stack` da classe `Vector`) para determinar se a pilha está vazia. Se estiver vazia, o método retorna `true`; caso contrário, `false`.

20.9 Classe `PriorityQueue` e interface `Queue`

Lembre-se de que uma fila é uma coleção que representa uma fila de espera — em geral, as inserções são feitas na parte posterior da fila e as exclusões na da frente. Na Seção 22.6, discutiremos e implementaremos uma estrutura de dados de fila. Nesta seção, investigamos a interface **`Queue`** e a classe **`PriorityQueue`** do Java do pacote `java.util`. A interface `Queue` estende a interface `Collection` e fornece operações adicionais para inserir, remover e inspecionar elementos em uma fila. `PriorityQueue`, que implementa a interface `Queue`, ordena elementos por sua ordem natural como especificado pelo método `compareTo` dos elementos `Comparable` ou por um objeto `Comparator` que é fornecido pelo construtor.

A classe `PriorityQueue` fornece funcionalidades que permitem inserções na ordem de classificação na estrutura de dados subjacente e exclusões a partir da frente da estrutura de dados subjacente. Ao adicionar elementos a uma `PriorityQueue`, os elementos são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade (isto é, o maior valor) será o primeiro elemento removido da `PriorityQueue`.

As operações `PriorityQueue` comuns são **`offer`**, para inserir um elemento na posição apropriada com base na ordem de prioridade, **`poll`** para remover o elemento de mais alta prioridade da fila de prioridade (isto é, a cabeça da fila), **`peek`** para obter uma referência ao

elemento de mais alta prioridade da fila de prioridade (sem remover esse elemento), **clear** para remover todos os elementos da fila de prioridade e **size**, para obter o número de elementos da fila de prioridade. A Figura 20.15 demonstra a classe PriorityQueue.

```
1  // Figura 20.15: PriorityQueueTest.java
2  // Programa de teste PriorityQueue.
3  import java.util.PriorityQueue;
4
5  public class PriorityQueueTest
6  {
7     public static void main( String[] args )
8     {
9        // fila de capacidade 11
10       PriorityQueue< Double > queue = new PriorityQueue< Double >();
11
12       // insere elementos na fila
13       queue.offer( 3.2 );
14       queue.offer( 9.8 );
15       queue.offer( 5.4 );
16
17       System.out.print( "Polling from queue: " );
18
19       // exibe elementos na fila
20       while (queue.size() > 0 )
21       {
22          System.out.printf( "%.1f ", queue.peek() ); // visualiza elemento superior
23          queue.poll(); // remove elemento superior
24       } // fim do while
25    } // fim de main
26 } // fim da classe PriorityQueueTest
```

```
Polling from queue: 3.2 5.4 9.8
```

Figura 20.15 | Programa de teste PriorityQueue.

A linha 10 cria uma PriorityQueue que armazena Doubles com uma capacidade inicial de 11 elementos e os ordena de acordo com a ordem natural do objeto (os padrões para uma PriorityQueue). Observe que PriorityQueue é uma classe genérica e que a linha 10 instancia uma PriorityQueue com um argumento de tipo Double. A classe PriorityQueue fornece cinco construtores adicionais. Um desses construtores aceita um int e um objeto Comparator para criar uma PriorityQueue com a capacidade inicial especificada pelo int e a ordem pelo Comparator. As linhas 13–15 utilizam o método offer para adicionar elementos à fila de prioridade. O método offer lança uma NullPointerException se o programa tentar adicionar um objeto null à fila. O loop nas linhas 20–24 utiliza o método size para determinar se a fila de prioridade está vazia (linha 20). Enquanto houver mais elementos, a linha 22 utiliza o método PriorityQueue peek para recuperar o elemento de prioridade mais alta na fila para saída (sem realmente remover o elemento da fila). A linha 23 remove o elemento de prioridade mais alta na fila com o método poll, que retorna o elemento removido.

20.10 Conjuntos

Um **Set** é uma Collection não ordenada de elementos únicos (isto é, sem elementos duplicados). A estrutura de coleções contém diversas implementações de Set, incluindo **HashSet** e **TreeSet**. HashSet armazena seus elementos em uma tabela de hash; e TreeSet armazena seus elementos em uma árvore. As tabelas de hash são apresentadas na Seção 20.11. As árvores são discutidas na Seção 22.7.

A Figura 20.16 utiliza um HashSet para remover strings duplicadas de uma List. Lembre-se de que tanto List como Collection são tipos genéricos, assim, a linha 16 cria uma List que contém objetos String e a linha 20 passa uma Collection de Strings para o método printNonDuplicates.

```
1  // Figura 20.16: SetTest.java
2  // HashSet utilizado para remover valores duplicados do array de strings.
3  import java.util.List;
4  import java.util.Arrays;
5  import java.util.HashSet;
6  import java.util.Set;
7  import java.util.Collection;
8
9  public class SetTest
```

```
10   {
11      public static void main( String[] args )
12      {
13         // cria e exibe uma List< String >
14         String[] colors = { "red", "white", "blue", "green", "gray",
15            "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
16         List< String > list = Arrays.asList( colors );
17         System.out.printf( "List: %s\n", list );
18
19         // elimina duplicatas e imprime valores únicos
20         printNonDuplicates( list );
21      } // fim de main
22
23      // cria um Set a partir de uma Collection para eliminar duplicatas
24      private static void printNonDuplicates( Collection< String > values )
25      {
26         // cria um HashSet
27         Set< String > set = new HashSet< String >( values );
28
29         System.out.print( "\nNonduplicates are: " );
30
31         for ( String value : set )
32            System.out.printf( "%s ", value );
33
34         System.out.println();
35      } // fim do método printNonDuplicates
36   } // fim da classe SetTest
```

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

Nonduplicates are: orange green white peach gray cyan red blue tan
```

Figura 20.16 | `HashSet` utilizado para remover valores duplicados de um array de strings.

O método `printNonDuplicates` (linhas 24–35) aceita um argumento `Collection`. A linha 27 constrói um `HashSet<String>` a partir do argumento `Collection<String>`. Por definição, `Sets` não contêm duplicata, então quando o `HashSet` é construído, ele remove qualquer duplicata da `Collection`. As linhas 31–32 imprimem os elementos em `Set`.

Conjuntos classificados

A estrutura de coleções também inclui a **interface `SortedSet`** (que estende `Set`) para conjuntos que mantêm seus elementos em ordem de classificação — a ordem natural dos elementos (por exemplo, números estão em ordem crescente) ou uma ordem especificada por um `Comparator`. A classe `TreeSet` implementa `SortedSet`. O programa na Figura 20.17 coloca strings em um `TreeSet`. As strings são classificadas à medida que são adicionadas ao `TreeSet`. Esse exemplo também demonstra métodos de visualização de intervalo, que permitem que um programa visualize uma parte de uma coleção.

```
1   // Figura 20.17: SortedSetTest.java
2   // Utilizando SortedSets e TreeSets.
3   import java.util.Arrays;
4   import java.util.SortedSet;
5   import java.util.TreeSet;
6
7   public class SortedSetTest
8   {
9      public static void main( String[] args )
10     {
11        // cria TreeSet a partir do array colors
12        String[] colors = { "yellow", "green", "black", "tan", "grey",
13           "white", "orange", "red", "green" };
14        SortedSet< String > tree =
15           new TreeSet< String >( Arrays.asList( colors ) );
16
17        System.out.print( "sorted set: " );
18        printSet( tree ); // conteúdo de saída de árvore
19
20        // obtém headSet com base em "orange"
```

```
21          System.out.print( "headSet (\"orange\"):  " );
22          printSet(tree.headSet( "orange" ) );
23
24          // obtém tailSet baseado em "orange"
25          System.out.print( "tailSet (\"orange\"):  " );
26          printSet(tree.tailSet( "orange" ) );
27
28          // obtém o primeiro e o último elementos
29          System.out.printf( "first: %s\n", tree.first() );
30          System.out.printf( "last : %s\n", tree.last() );
31       } // fim de main
32
33       // imprime a SortedSet utilizando a instrução for aprimorada
34       private static void printSet( SortedSet< String > set )
35       {
36          for ( String s : set )
37             System.out.printf( "%s ", s );
38
39          System.out.println();
40       } // fim do método printSet
41    } // fim da classe SortedSetTest
```

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"):  black green grey
tailSet ("orange"):  orange red tan white yellow
first: black
last : yellow
```

Figura 20.17 | Utilizando `SortedSets` e `TreeSets`.

As linhas 14–15 criam um `TreeSet<String>` que contém os elementos do array `colors`, então atribuem a nova variável `TreeSet<String>` à variável `SortedSet<String>` `tree`. A linha 18 gera saída do conjunto inicial de strings utilizando o método `printSet` (linhas 34–40), que discutiremos em breve. A linha 22 chama o **método `TreeSet` `headSet`** para obter um subconjunto do `TreeSet` em que cada elemento é menor que `"orange"`. A visualização retornada de `headSet` é então enviada para saída com `printSet`. Se o subconjunto sofrer alguma alteração, o `TreeSet` original também será alterado, pois o subconjunto retornado por `headSet` é uma visualização do `TreeSet`.

A linha 26 chama o **método `TreeSet` `tailSet`** para obter um subconjunto em que cada elemento seja maior ou igual a `"orange"`, e então gera o resultado. Qualquer alteração feita pela visualização `tailSet` ocorre no `TreeSet` original. As linhas 29–30 chamam os **métodos `SortedSet` `first`** e **`last`** para obter os menores e os maiores elementos do conjunto, respectivamente.

O método `printSet` (linhas 34–40) aceita um `SortedSet` como um argumento e o imprime. As linhas 36–37 imprimem cada elemento do `SortedSet` utilizando a instrução `for` aprimorada.

20.11 Mapas

Os **`map`**s associam chaves a valores. As chaves em um `Map` devem ser únicas, mas não os valores associados. Se um `Map` contiver tanto chaves como valores únicos, diz-se que ele implementa um **mapeamento um para um**. Se apenas as chaves forem únicas, diz-se que o `Map` implementa um **mapeamento de "muitos para um"** — muitas chaves podem mapear para um valor.

Os `mapas` diferem de `Sets` no sentido de que `Maps` contêm chaves e valores, enquanto `Sets` contêm apenas valores. Três das várias classes que implementam a interface `Map` são **`Hashtable`**, **`HashMap`** e **`TreeMap`**. `Hashtables` e `HashMaps` armazenam elementos em tabelas de hash e `TreeMaps` armazenam elementos em árvores. Esta seção discute as tabelas de hash e fornece um exemplo que utiliza um `HashMap` para armazenar pares chave/valor. A interface **`SortedMap`** estende `Map` e mantém suas chaves em ordem de classificação — na ordem natural dos elementos ou em uma ordem especificada por uma implementação `Comparator`. A classe `TreeMap` implementa `SortedMap`.

Implementação `Map` com tabelas de hash

As linguagens de programação orientada a objetos facilitam a criação de novos tipos. Quando um programa cria objetos de tipos novos ou existentes, ele pode precisar armazená-los e recuperá-los eficientemente. Armazenar e recuperar informações com arrays é eficiente se algum aspecto de seus dados corresponder diretamente com o valor numérico e se essas chaves forem únicas e fortemente empacotadas. Se você tiver 100 funcionários com CPF de nove dígitos e quiser armazenar e recuperar dados sobre os funcionários usando o CPF como chave, a tarefa exigirá um array com mais de 700 milhões de elementos, porque os CPFs de nove dígitos devem iniciar com 001-733 de acordo com o site Social Security Administration.

```
www.socialsecurity.gov/employer/stateweb.htm
```

Isso é complicado para praticamente todos os aplicativos que utilizam números do CPF como chaves. Um programa que tivesse um array tão grande assim poderia alcançar alto desempenho tanto para armazenar como para recuperar registros de empregados simplesmente utilizando o número do CPF como o índice de array.

Numerosos aplicativos têm esse problema, ou seja, as chaves são tanto do tipo errado (por exemplo, inteiros não positivos que correspondem a subscritos de array) como chaves do tipo certo, mas estão escassamente espalhadas em um enorme intervalo. O que é necessário é um esquema de alta velocidade para converter chaves, como números do CPF, códigos de produtos em estoque e muitos outros em índices de array únicos. Então, quando um aplicativo precisasse armazenar algo, o esquema poderia converter a chave do aplicativo rapidamente em um índice e o registro poderia ser armazenado nessa posição do array. A recuperação é realizada da mesma maneira: Uma vez que o aplicativo tem uma chave pela qual ele quer recuperar um registro de dados, o aplicativo simplesmente aplica a conversão à chave — isso produz o índice de array em que os dados são armazenados e recuperados.

O esquema que descrevemos aqui é a base de uma técnica chamada **hashing**. Por que esse nome? Quando convertemos uma chave em um índice de array, literalmente embaralhamos os bits, formando um tipo de número "confusamente misturado," ou hasheado. O número realmente não tem nenhuma importância real além de sua utilidade em armazenar e recuperar um registro de dados particulares.

Um glitch (falha, em geral de pequena gravidade) no esquema é chamado de **colisão** — esta ocorre quando duas chaves diferentes produzem hash para a mesma célula (ou elemento) no array. Não podemos armazenar dois valores no mesmo espaço, então precisamos localizar uma posição alternativa para todos os valores depois do primeiro que produz hash para um índice de array particular. Há muitos esquemas para fazer isso. Um é um "novo hash" (isto é, aplicar a transformação de hashing à chave para fornecer uma próxima célula candidata no array). O processo de hashing é projetado para distribuir os valores por toda a tabela; desse modo, a suposição é de que uma célula disponível será localizada com apenas alguns hashes.

Outro esquema utiliza um hash para localizar a primeira célula candidata. Se essa célula estiver ocupada, sucessivas células são pesquisadas em ordem até que uma célula disponível seja localizada. A recuperação funciona da mesma maneira: A chave sofre hash uma vez para determinar a localização inicial e verificar se ela contém os dados desejados. Se ela contiver, a pesquisa é concluída. Se não contiver, células sucessivas são pesquisadas linearmente até que os dados desejados sejam localizados.

A solução mais popular para colisões de tabela de hash é fazer cada célula da tabela ser um "bucket" de hash, em geral uma lista vinculada de todos os pares chave/valor que sofrem hash para essa célula. Essa é a solução que as classes `Hashtable` e `HashMap` do Java (do pacote `java.util`) implementam. Tanto `Hashtable` como `HashMap` implementam a interface `Map`. As principais diferenças entre eles são que `HashMap` é não sincronizado (múltiplas threads não devem modificar um `HashMap` concorrentemente) e permite chaves `null` e valores `null`.

O **fator de carga** de uma tabela de hash afeta o desempenho de esquemas de hashing. O fator de carga é a relação do número de células ocupadas na tabela de hash com o número total de células na tabela de hash. Quanto mais a proporção se aproximar de 1,0, maior a chance de colisões.

Dica de desempenho 20.2

O fator de carga em uma tabela de hash é um exemplo clássico de uma troca entre espaço de memória e tempo de execução: aumentando o fator de carga, melhoramos a utilização da memória, mas o programa executa mais lentamente por causa do aumento das colisões de hashing. Diminuindo o fator de carga, melhoramos a velocidade do programa, em virtude da redução das colisões de hashing, mas fazemos uma pobre utilização da memória porque uma parte maior da tabela de hash permanece vazia.

As tabelas de hash são complexas de programar. Os alunos de ciência da computação estudam esquemas de hashing em cursos chamados "Estruturas de dados" e "Algoritmos." As classes `Hashtable` e `HashMap` permitem o uso de hashing sem a necessidade de implementar mecanismos de tabela de hash. Esse conceito é extremamente importante em nosso estudo de programação orientada a objetos. Como discutido nos capítulos anteriores, as classes encapsulam e ocultam a complexidade (isto é, detalhes de implementação) e oferecem interfaces amigáveis ao usuário. Criar adequadamente as classes para exibir esse comportamento é uma das habilidades mais estimadas no campo da programação orientada a objetos. A Figura 20.18 utiliza um `HashMap` para contar o número de ocorrências de cada palavra em uma string.

```
1  // Figura 20.18: WordTypeCount.java
2  // O programa conta o número de ocorrências de cada palavra em uma String.
3  import java.util.Map;
4  import java.util.HashMap;
5  import java.util.Set;
6  import java.util.TreeSet;
7  import java.util.Scanner;
8
9  public class WordTypeCount
10 {
11    public static void main( String[] args )
12    {
13       // cria HashMap para armazenar chaves de String e valores Integer
14       Map< String, Integer > myMap = new HashMap< String, Integer >();
```

```
15
16          createMap( myMap ); // cria mapa com base na entrada do usuário
17          displayMap( myMap ); // exibe o conteúdo do mapa
18       } // fim de main
19
20       // cria mapa de entrada do usuário
21       private static void createMap( Map< String, Integer > map )
22       {
23          Scanner scanner = new Scanner( System.in ); // cria o scanner
24          System.out.println( "Enter a string:" ); // solicita a entrada do usuário
25          String input = scanner.nextLine();
26
27          // tokeniza a entrada
28          String[] tokens = input.split( " " );
29
30          // processamento de texto de entrada
31          for ( String token : tokens )
32          {
33             String word = token.toLowerCase(); // obtém palavra em letras minúsculas
34
35             // se o mapa contiver a palavra
36             if (map.containsKey( word ) ) // is word in map
37             {
38                int count = map.get( word ); // obtém a contagem atual
39                map.put( word, count + 1 ); // incrementa a contagem
40             } // fim do if
41             else
42                map.put( word, 1 ); // adiciona nova palavra com uma contagem de 1 para mapa
43          } // fim do for
44       } // fim do método createMap
45
46       // exibe o conteúdo do mapa
47       private static void displayMap( Map< String, Integer > map )
48       {
49          Set< String > keys = map.keySet(); //obtém chaves
50
51          // classifica as chaves
52          TreeSet< String > sortedKeys = new TreeSet< String >( keys );
53
54          System.out.println( "\nMap contains:\nKey\t\tValue" );
55
56          // gera a saída de cada chave no mapa
57          for ( String key : sortedKeys )
58             System.out.printf( "%-10s%10s\n", key, map.get( key ) );
59
60          System.out.printf(
61             "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
62       } // fim do método displayMap
63    } // fim da classe WordTypeCount
```

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words
```

```
Map contains:
Key             Value
a                  1
another            1
different          1
is                 2
sample             2
sentence           2
several            2
this               2
with               2
words              2

size: 10
isEmpty: false
```

Figura 20.18 | O programa conta o número de ocorrências de cada palavra em uma `String`.

A linha 14 cria um `HashMap` vazio com uma capacidade inicial padrão (16 elementos) e um fator de carga padrão (0,75) — esses padrões são criados na implementação de `HashMap`. Quando o número de posições ocupadas no `HashMap` tornar-se maior que a capacidade vezes o fator de carga, a capacidade é automaticamente dobrada. Note que `HashMap` é uma classe genérica que aceita dois argumentos de tipo — o tipo da chave (isto é, `String`) e o tipo do valor (isto é, `Integer`). Lembre-se de que os argumentos de tipo passados para uma classe genérica devem ser tipos por referência, daí o segundo argumento de tipo ser `Integer`, não `int`.

A linha 16 chama o método `createMap` (linhas 21–44), que utiliza um `map` para armazenar o número de ocorrências de cada palavra na frase. A linha 25 obtém a entrada de usuário, e a linha 28 a tokeniza. O loop nas linhas 31–43 converte o próximo token em letra minúscula (linha 33) e chama o **método `Map` `containsKey`** (linha 36) para determinar se a palavra está no mapa (e assim ocorreu anteriormente na string). Se `Map` não contiver um mapeamento para a palavra, a linha 42 utiliza o **método `Map` `put`** para criar uma nova entrada no mapa, com a palavra como a chave e um objeto `Integer` que contém `1` como o valor. O autoboxing ocorre quando o programa passa o inteiro `1` para o método `put`, porque o mapa armazena o número de ocorrências da palavra como um `Integer`. Se a palavra existir no mapa, a linha 38 utiliza o **método `Map` `get`** para obter o valor associado (a contagem) da chave no mapa. A linha 39 incrementa esse valor e utiliza `put` para substituir o valor associado da chave no mapa. O método `put` retorna o valor anterior associado da chave, ou `null` se a chave não estiver no mapa.

O método `displayMap` (linhas 47–62) exibe todas as entradas no mapa. Ele utiliza o **método `HashMap` `keySet`** (linha 49) para obter um conjunto das chaves. As chaves têm o tipo `String` no `map`, então o método `keySet` retorna um tipo genérico `Set` com o parâmetro de tipo especificado para ser `String`. A linha 52 cria um `TreeSet` das chaves, em que as chaves são classificadas. O loop nas linhas 57–58 acessa cada chave e seu valor no mapa. A linha 58 exibe cada chave e seu valor utilizando o especificador de formato `%-10s` para alinhar à esquerda cada chave e formatar o especificador `%10s` à direita de cada valor. Observe que as chaves são exibidas em ordem crescente. A linha 61 chama o **método `Map` `size`** para obter o número de pares chave/valor no `Map`. A linha 61 também chama o **método `Map` `isEmpty`**, que retorna um `boolean` que indica se o `Map` está vazio.

20.12 Classe Properties

Um objeto **`Properties`** é uma `Hashtable` persistente que normalmente armazena pares chave/valor de strings — assumindo que você utiliza os métodos **`setProperty`** e **`getProperty`** para manipular a tabela em vez dos métodos `Hashtable` `put` e `get` herdados. Por "persistente", queremos dizer que o objeto `Properties` pode ser gravado em um fluxo de saída (possivelmente um arquivo) e lido de volta por um fluxo de entrada (input stream). Um uso comum de objetos `Properties` em versões anteriores do Java era manter os dados de configuração de aplicativo ou preferências de usuário de aplicativos. [*Nota*: A **Preferences API** (pacote **`java.util.prefs`**) foi projetada para substituir esse uso particular da classe `Properties`, mas está além do escopo deste livro. Para aprender mais, visite `java.sun.com/javase/6/docs/technotes/guides/preferences/index.html`.]

A classe `Properties` estende a classe `Hashtable`. A Figura 20.19 demonstra vários métodos da classe `Properties`.

```
1  // Figura 20.19: PropertiesTest.java
2  // Demonstra classe Properties do pacote java.util.
3  import java.io.FileOutputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.util.Properties;
7  import java.util.Set;
8
9  public class PropertiesTest
```

```
{
   public static void main( String[] args )
   {
      Properties table = new Properties(); // cria tabela Properties

      // configura propriedades
      table.setProperty( “color”, “blue” );
      table.setProperty( “width”, “200” );

      System.out.println( "After setting properties" );
      listProperties( table ); // exibe os valores da propriedade

      // substitui o valor de propriedade
      table.setProperty( “color”, “red” );

      System.out.println( "After replacing properties" );
      listProperties( table ); // exibe os valores da propriedade

      saveProperties( table ); // salva as propriedades

      table.clear(); // tabela vazia

      System.out.println( "After clearing properties" );
      listProperties( table ); // exibe os valores da propriedade

      loadProperties( table ); // carrega propriedades

      // obtém valor de cor da propriedade
      Object value = table.getProperty( “color” );

      // verifica se o valor está na tabela
      if ( value != null )
         System.out.printf( "Property color's value is %s\n", value );
      else
         System.out.println( "Property color is not in table" );
   } // fim de main

   // salva as propriedades para um arquivo
   private static void saveProperties( Properties props )
   {
      // salva o conteúdo da tabela
      try
      {
         FileOutputStream output = new FileOutputStream( "props.dat" );
         props.store( output, “Sample Properties” ); // salva as propriedades
         output.close();
         System.out.println( "After saving properties" );
         listProperties( props ); // exibe os valores da propriedade
      } // fim do try
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // fim do catch
   } // fim do método saveProperties

   // carrega as propriedades de um arquivo
   private static void loadProperties( Properties props )
   {
      // carrega o conteúdo da tabela
      try
      {
         FileInputStream input = new FileInputStream( "props.dat" );
         props.load( input ); // carrega propriedades
         input.close();
         System.out.println( "After loading properties" );
         listProperties( props ); // exibe os valores da propriedade
      } // fim do try
      catch ( IOException ioException )
      {
         ioException.printStackTrace();
      } // fim do catch
```

```
81       } // fim do método loadProperties
82
83       // gera saída de valores de propriedade
84       private static void listProperties( Properties props )
85       {
86          Set< Object > keys = props.keySet(); // obtém nomes de propriedade
87
88          // gera saída de pares nome/valor
89          for ( Object key : keys )
90             System.out.printf(
91                "%s\t%s\n", key, props.getProperty( ( String ) key ) );
92
93          System.out.println();
94       } // fim do método listProperties
95    } // fim da classe PropertiesTest
```

```
After setting properties
color   blue
width   200

After replacing properties
color   red
width   200

After saving properties
color   red
width   200

After clearing properties

After loading properties
color   red
width   200

Property color's value is red
```

Figura 20.19 | Classe `Properties` do pacote `java.util`.

A linha 13 utiliza o construtor sem argumento para criar uma `Properties` `table` vazia sem propriedades padrão. A classe `Properties` também fornece um construtor sobrecarregado que recebe uma referência a um objeto `Properties` que contém os valores de propriedade padrão. As linhas 16 e 17 chamam o método `Properties` `setProperty` para armazenar um valor para a chave especificada. Se a chave não existir em `table`, `setProperty` retorna `null`; caso contrário, ela retorna o valor anterior dessa chave.

A linha 38 chama o método `Properties` `getProperty` para localizar o valor associado com a chave especificada. Se a chave não for localizada nesse objeto `Properties`, `getProperty` retorna `null`. Uma versão sobrecarregada desse método recebe um segundo argumento que especifica o valor padrão a retornar se `getProperty` não puder localizar a chave.

A linha 54 chama o **método `Properties` `store`** para salvar o conteúdo do objeto `Properties` no `OutputStream` especificado como o primeiro argumento (nesse caso, um `FileOutputStream`). O segundo argumento, uma `String`, é uma descrição no arquivo. O **método `Properties` `list`**, que aceita um argumento `PrintStream`, é útil para exibir a lista de propriedades.

A linha 72 chama o **método `Properties` `load`** para restaurar o conteúdo do objeto `Properties` do `InputStream` especificado como o primeiro argumento (nesse caso, um `FileInputStream`). A linha 86 chama o método `Properties` `keySet` para obter um `Set` dos nomes da propriedade. A linha 91 obtém o valor de uma propriedade passando uma chave para o método `getProperty`.

20.13 Coleções sincronizadas

No Capítulo 26, discutimos multithreading. Exceto para `Vector` e `Hashtable`, as coleções na estrutura de coleções são não sincronizadas por padrão, portanto, podem operar eficientemente quando o multithreading não é necessário. Mas como essas coleções são não sincronizadas, o acesso concorrente por múltiplas threads a uma `Collection` poderia causar resultados indeterminados ou erros fatais. Para evitar potenciais problemas de encadeamento, os **empacotadores de sincronização** são utilizados para coleções que poderiam ser acessadas por múltiplas threads. Um objeto **empacotador** (wrapper) recebe chamadas de método, adiciona sincronização de thread (para evitar acesso simultâneo à coleção) e delega as chamadas para o objeto de coleção empacotado. A API `Collections` fornece um conjunto de métodos `static` para empacotar coleções como versões sincronizadas. Cabeçalhos de método para os empacotadores de sincronização são listados na Figura 20.20. Os detalhes sobre esses métodos estão disponíveis em `java.sun.com/javase/6/docs/api/java/util/Collections.html`. Todos esses métodos aceitam um tipo genérico e retornam uma visualização sincronizada do tipo genérico. Por exemplo, o seguinte código cria uma `List` sincronizada (`list2`) que armazena objetos `String`:

```
List< String > list1 = new ArrayList< String >();
List< String > list2 = Collections.synchronizedList( list1 );
```

Cabeçalhos de método `public static`
`< T > Collection< T > synchronizedCollection( Collection< T > c )`
`< T > List< T > synchronizedList( List< T > aList )`
`< T > Set< T > synchronizedSet( Set< T > s )`
`< T > SortedSet< T > synchronizedSortedSet( SortedSet< T > s )`
`< K, V > Map< K, V > synchronizedMap( Map< K, V > m )`
`< K, V > SortedMap< K, V > synchronizedSortedMap( SortedMap< K, V > m )`

Figura 20.20 | Métodos empacotadores de sincronização.

20.14 Coleções não modificáveis

A classe `Collections` fornece um conjunto de métodos `static` que criam **empacotadores não modificáveis** para coleções. Empacotadores não modificáveis lançam `UnsupportedOperationExceptions` se forem feitas tentativas de modificar a coleção. Os cabeçalhos desses métodos são listados na Figura 20.21. Os detalhes sobre esses eles estão disponíveis em `java.sun.com/javase/6/docs/api/java/util/Collections.html`. Todos esses métodos aceitam um tipo genérico e retornam uma visualização não modificável do tipo genérico. Por exemplo, o seguinte código cria uma `List` (`list2`) não modificável que armazena objetos `String`:

```
List< String > list1 = new ArrayList< String >();
List< String > list2 = Collections.unmodifiableList( list1 );
```

Cabeçalhos de método `public static`
`< T > Collection< T > unmodifiableCollection( Collection< T > c )`
`< T > List< T > unmodifiableList( List< T > aList )`
`< T > Set< T > unmodifiableSet( Set< T > s )`
`< T > SortedSet< T > unmodifiableSortedSet( SortedSet< T > s )`
`< K, V > Map< K, V > unmodifiableMap( Map< K, V > m )`
`< K, V > SortedMap< K, V > unmodifiableSortedMap( SortedMap< K, V > m )`

Figura 20.21 | Métodos empacotadores não modificáveis.

Observação de engenharia de software 20.5

Você pode utilizar um empacotador não modificável para criar uma coleção que oferece acesso de leitura às outras pessoas enquanto permite o acesso de leitura e gravação para você. Você faz isso simplesmente dando às outras pessoas uma referência ao empacotador não modificável ao mesmo tempo em que retém para você uma referência à coleção original.

20.15 Implementações abstratas

A estrutura de coleções fornece várias implementações abstratas de interfaces `Collection` a partir das quais você pode implementar rapidamente aplicativos personalizados completos. Essas implementações abstratas incluem uma implementação magra de `Collection` chamada **`AbstractCollection`**, uma implementação de `List` que permite acesso aleatório a seus elementos chamada **`AbstractList`**, uma implementação de `Map` chamada **`AbstractMap`**, uma implementação de `List` que permite acesso sequencial a seus elementos chamada **`AbstractSequentialList`**, uma implementação de `Set` chamada **`AbstractSet`** e uma implementação de `Queue` chamada **`AbstractQueue`**. Você pode aprender mais sobre essas classes em `java.sun.com/javase/6/docs/api/java/util/package-summary.html`.

Para escrever uma implementação personalizada, você pode estender a implementação abstrata que melhor atender às suas necessidades e implementar cada um dos métodos `abstract` da classe. Então, se sua coleção precisa ser modificável, anule qualquer método concreto que impeça a modificação.

20.16 Conclusão

Este capítulo introduziu a estrutura de coleções do Java. Você aprendeu a hierarquia de coleção e a maneira de utilizar as interfaces de estrutura de coleções para programar com coleções polimorficamente. Você utilizou as classes `ArrayList` e `LinkedList`, que implementam a interface `List`. Apresentamos as interfaces e classes predefinidas do Java para manipular pilhas e filas. Você utilizou vários métodos predefinidos para manipular coleções. Em seguida, você aprendeu a utilizar a interface `Set` e a classe `HashSet` para manipular uma coleção não ordenada de valores únicos. Continuamos nossa apresentação de conjuntos com a interface `SortedSet` e a classe `TreeSet` para manipular uma coleção classificada de valores únicos. Em seguida, você aprendeu sobre as interfaces e classes do Java para manipular pares de chave/valor — `Map`, `SortedMap`, `Hashtable`, `HashMap` e `TreeMap`. Então discutimos a classe `Properties` especializada para manipular pares de chave/valor `String`s que podem ser armazenadas em um arquivo e recuperadas de um arquivo. Por fim, discutimos os métodos `static` da classe `Collections` para obter visualizações não modificáveis e sincronizadas de coleções. O Capítulo 21 demonstra como utilizar as capacidades dos genéricos do Java para que você implemente seus próprios métodos e classes genéricos.

Resumo

Seção 20.1 Introdução

- A estrutura de coleções Java fornece acesso a estruturas de dados predefinidas bem como a métodos para manipulá-las.

Seção 20.2 Visão geral das coleções

- Uma coleção é um objeto que pode armazenar referências a outros objetos.
- As classes e interfaces da estrutura de coleções estão no pacote `java-.util`.

Seção 20.3 Classes empacotadoras de tipo para tipos primitivos

- As classes empacotadoras de tipo (por exemplo, `Integer`, `Double`, `Boolean`) permitem aos programadores manipular valores de tipo primitivo como objetos. Os objetos dessas classes podem ser utilizados em coleções.

Seção 20.4 Autoboxing e auto-unboxing

- O boxing converte um valor primitivo em um objeto da classe empacotadora de tipo correspondente. O unboxing converte um objeto empacotador de tipo no valor primitivo correspondente.
- O Java realiza conversões boxing e unboxing automaticamente.

Seção 20.5 Interface `Collection` e classe `Collections`

- A interface `Collection` é a interface-raiz na hierarquia de coleções. As interfaces `Set` e `List` estendem `Collection`. A interface `Collection` contém operações de adição, limpeza, comparação e retenção de objetos em uma coleção, e o método `iterator` para obter uma coleção `Iterator`.
- A classe `Collections` fornece os métodos `static` para manipular coleções.

Seção 20.6 Listas

- Uma `List` é uma `Collection` ordenada que pode conter elementos duplicados.
- A interface `List` é implementada pelas classes `ArrayList`, `LinkedList` e `Vector`. `ArrayList` é uma implementação de array redimensionável de uma `List`. `LinkedList` é uma implementação de lista encadeada de uma `List`.
- O método `iterador hasNext` determina se uma `Collection` contém outro elemento. O método `next` retorna uma referência ao próximo objeto na `Collection` e avança o `Iterator`.
- O método `subList` retorna uma visualização em uma `List`. Qualquer alteração feita nessa visualização também será feita na `List`.
- O método `clear` remove elementos de um `List`.
- O método `toArray` retorna o conteúdo de uma coleção como um array.

Seção 20.7 Métodos de coleções

- Os algoritmos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill`, `copy`, `addAll`, `frequency` e `disjoint` operam em `List`s. Os algoritmos `min` e `max` operam em `Collections`.
- O algoritmo `addAll` acrescenta todos os elementos de um array a uma coleção, `frequency` calcula quantos elementos da coleção são iguais ao elemento especificado e `disjoint` determina se as duas coleções têm elementos em comum.
- Os algoritmos `min` e `max` localizam os menores e maiores itens em uma coleção.
- A interface `Comparator` fornece um meio de classificar elementos de uma `Collection` em uma ordem diferente de sua ordem natural.

- O método `Collections` `reverseOrder` retorna um objeto `Comparator` que pode ser utilizado com `sort` para classificar elementos de uma coleção na ordem inversa.
- O algoritmo `shuffle` ordena aleatoriamente os elementos de uma `List`.
- O algoritmo `binarySearch` localiza um `Object` em uma `List` classificada.

Seção 20.8 Classe `Stack` do pacote `java.util`

- A classe `Stack` estende `Vector`. O método `Stack` `push` adiciona seu argumento na parte superior da pilha. O método `pop` remove o elemento superior da pilha. O método `peek` retorna uma referência ao elemento na parte superior da pilha sem removê-lo. O método `Stack` `isEmpty` determina se a pilha está vazia.

Seção 20.9 Classe `PriorityQueue` e interface `Queue`

- A interface `Queue` estende a interface `Collection` e fornece operações adicionais para inserir, remover e inspecionar elementos em uma fila.
- `PriorityQueue` implementa a interface `Queue` e ordena elementos pela sua ordem natural ou por um objeto `Comparator` que é fornecido para o construtor.
- O método `PriorityQueue` `offer` insere um elemento na localização adequada com base na ordem de prioridade. O método `poll` remove o elemento da mais alta prioridade da fila de prioridade. O método `peek` obtém uma referência ao elemento de prioridade mais alta da fila de prioridades. O método `clear` remove todos os elementos da fila de prioridades. O método `size` obtém o número de elementos na fila de prioridades.

Seção 20.10 Conjuntos

- Um `Set` é uma `Collection` não ordenada que não contém nenhum elemento duplicado. `HashSet` armazena seus elementos em uma tabela de hash. `TreeSet` armazena seus elementos em uma árvore.
- A interface `SortedSet` estende `Set` e representa um conjunto que mantém seus elementos na ordem de classificação. A classe `TreeSet` implementa `SortedSet`.
- O método `TreeSet` `headSet` obtém uma visualização `TreeSet` que contém elementos que são menores do que um elemento especificado. O método `tailSet` obtém uma visualização `TreeSet` que contém elementos que são maiores que ou iguais a um elemento especificado. Quaisquer alterações feitas nessas visualizações ocorrerão no `TreeSet` original.

Seção 20.11 Mapas

- `Maps` armazenam pares de chave/valor e não podem conter chaves duplicadas. `HashMaps` e `Hashtables` armazenam elementos em uma tabela de hash, e `TreeMaps` armazenam elementos em uma árvore.
- `HashMap` aceita dois argumentos de tipo — o tipo de chave e o tipo do valor.
- O método `HashMap` `put` adiciona uma chave e um valor em um `HashMap`. O método `get` localiza o valor associado com a chave especificada. O método `isEmpty` determina se o mapa está vazio.
- O método `HashMap` `keySet` retorna o conjunto de chaves. O método `Map` `size` retorna o número de pares chave/valor no `Map`.
- A interface `SortedMap` estende `Map` e representa um mapa que mantém suas chaves em ordem de classificação. A classe `TreeMap` implementa `SortedMap`.

Seção 20.12 Classe `Properties`

- Um objeto `Properties` é um objeto `Hashtable` persistente. A classe `Properties` estende `Hashtable`.
- O construtor sem argumentos `Properties` cria uma tabela `Properties` vazia. Um construtor sobrecarregado recebe um objeto `Properties` contendo valores de propriedade padrão.
- O método `Properties` `setProperty` especifica o valor associado com o argumento de chave. O método `getProperty` localiza o valor da chave especificada como um argumento. O método `store` salva o conteúdo de um objeto `Properties` no `OutputStream` especificado. O método `load` restaura o conteúdo de um objeto `Properties` a partir do `InputStream` especificado.

Seção 20.13 Coleções sincronizadas

- As coleções da estrutura de coleções são não sincronizadas. Os empacotadores de sincronização são oferecidos para coleções que podem ser acessadas por múltiplas threads simultaneamente.

Seção 20.14 Coleções não modificáveis

- A classe `Collections` fornece um conjunto de métodos `public` `static` para converter coleções em versões não modificáveis. Empacotadores não modificáveis lançam `UnsupportedOperation-Exceptions` se forem feitas tentativas de modificar a coleção.

Seção 20.15 Implementações abstratas

- A estrutura de coleções fornece várias implementações abstratas de interfaces de coleção a partir das quais você pode concretizar rapidamente implementações personalizadas completas.

Terminologia

AbstractCollection, classe, 664
AbstractList, classe, 664
AbstractMap, classe, 664
AbstractQueue, classe, 664
AbstractSequentialList, classe, 664
AbstractSet, classe, 664
addAll, método da classe Collections, 652
addAll, método de interfaceList, 643
add, método da classe LinkedList, 644
add, método de interface List, 641
addFirst, método da classe LinkedList, 644
addLast, método da classe LinkedList, 644
API de preferências, 661
array de apoio, 644
ArrayList, classe, 639
asList, método da classe Arrays, 644
autoboxing, 638
auto-unboxing, 638
binarySearch, método da classe Collections, 651
Boolean, classe, 638
boxing, conversão, 638
Byte, classe, 638
Character, classe, 638
classes empacotadoras de tipo, 638
clear, método da classe List, 643
clear, método da classe PriorityQueue, 656
coleções, 637
colisão em uma tabela de hash, 659
Collection, interface, 639
Collections, classe, 639
Comparable, interface, 645
Comparator, interface, 645
compare, método de interface Comparator, 647
contains, método de interface Collection, 641
containsKey, método de interface Map, 661
conversão unboxing, 638
copy, método da classe Collections, 650
disjoint, método da classe Collections, 653
Double, classe, 638
empacotador de sincronização, 663
empacotador não modificável, 664
EmptyStackException, classe, 655
estrutura de coleções, 637
F, sufixo para literais float, 655
fator de carga, 659
fill, método da classe Collections, 650
first, método de interface SortedSet, 658
Float, classe, 638
frequency, método da classe Collections, 653
get, método de interface List, 641
get, método de interface Map, 661
getProperty, método da classe Properties, 661
hashing, 659
HashMap, classe, 658
HashSet, classe, 656
Hashtable, classe, 658
hasNext, método de interface Iterator, 641
hasPrevious, método de interface ListIterator, 643
headSet, método da classe TreeSet, 658
Integer, classe, 638
isEmpty, método da classe Stack, 655
isEmpty, método de interface Map, 661
iterador, 637
iterador bidirecional, 643
Iterator, interface, 639
iterator, método de interface Collection, 641
java.util.prefs, pacote, 661
keySet, método da classe HashMap, 661
last, método da classe SortedSet, 658
L, sufixo para long literais, 655
LinkedList, classe, 639
List, interface, 643
list, método da classe Properties, 663
ListIterator, interface, 639
listIterator, método de interface List, 643
load, método da classe Properties, 663
Long, classe, 638
Map, interface, 658
max, método da classe Collections, 650
método de comparação natural, 645
métodos de visualização de intervalo, 643
métodos empacotadores da classe Collections, 639
min, método da classe Collections, 650
muitos para um, mapeamento, 658
next, método de interface Iterator, 641
objeto empacotador (coleções), 663
offer, método da classe PriorityQueue, 655
peek, método da classe PriorityQueue, 655
peek, método da classe Stack, 655
poll, método da classe PriorityQueue, 655
pop, método da classe Stack, 655
previous, método de interface ListIterator, 643
PriorityQueue, classe, 655
Properties, classe, 661
push, método da classe Stack, 655
put, método de interface Map, 661
Queue, interface, 639
remove, método de interface Iterator, 641
reverse, método da classe Collections, 650
reverseOrder, método da classe Collections, 646
sequência, 639
Set, interface, 639
set, método de interface ListIterator, 643
setProperty, método da classe Properties, 661
Short, classe, 638
shuffle, método da classe Collections, 648
size, método da classe PriorityQueue, 656
size, método de interface List, 641
size, método de interface Map, 661
sort, método da classe Collections, 645
SortedMap, interface, 658
SortedSet, interface, 657
Stack, classe, 654
store, método da classe Properties, 663
sublista, 643
subList, método de interface List, 643
tailSet, método da classe TreeSet, 658
toArray, método de interface List, 644
toLowerCase, método da classe String, 643
toUpperCase, método da classe String, 643
TreeMap, classe, 658
TreeSet, classe, 656
um para um, mapeamento, 658
UnsupportedOperationException, classe, 644
Vector, classe, 639
visualização em uma coleção, 643
volume, operação de, 639

Exercícios de autorrevisão

20.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um(a) _____ é utilizado(a) para iterar por uma coleção e pode remover elementos da coleção durante a iteração.

b) Um elemento em uma List pode ser acessado utilizando o(a) _____ do elemento.

c) Assumindo que myArray contém referências aos objetos Double, _____ ocorre quando a instrução "myArray[0] = 1.25;" executa.

d) As classes Java _____ e _____ fornecem as capacidades de estruturas de dados no estilo array que podem redimensionar a si mesmas dinamicamente.

e) Se você não especificar um incremento de capacidade, o sistema irá _____ o tamanho do Vector toda vez que capacidade adicional for necessária.

f) Você pode utilizar um(a) _____ para criar uma coleção que oferece acesso de leitura para outras pessoas ao mesmo tempo em que permite que você tenha acesso de gravação e leitura.

g) Assumindo que `myArray` contém referências aos objetos `Double`, _____ ocorre quando a instrução "`double number = myArray[ 0 ];`" executa.

h) O algoritmo _____ de `Collections` determina se duas coleções têm elementos em comum.

20.2 Determine se cada sentença é *verdadeira* ou *falsa*. Se *falso*, explique por quê.

a) Os valores de tipos primitivos podem ser armazenados diretamente em uma coleção.

b) Uma `Set` pode conter valores duplicados.

c) Um `Map` pode conter chaves duplicatas.

d) Uma `LinkedList` pode conter valores duplicados.

e) `Collections` é uma `interface`.

f) `Iterator`s podem remover elementos.

g) Com hashing, enquanto o fator de carga aumenta, a chance de colisões diminui.

h) Uma `PriorityQueue` permite elementos `null`.

Respostas dos exercícios de autorrevisão

20.1 a) `Iterator`. b) índice. c) autoboxing. d) `ArrayList`, `Vector`. e) dobrar. f) empacotador não modificável. g) auto-unboxing. h) `disjoint`.

20.2 a) Falsa. O autoboxing ocorre ao adicionar um tipo primitivo a uma coleção, o que significa que o tipo primitivo é convertido em sua classe empacotadora de tipo correspondente.

b) Falsa. Um `Set` não pode conter valores duplicados.

c) Falsa. Um `Map` não pode conter chaves duplicadas.

d) Verdadeira.

e) Falsa. `Collections` é uma `class`; e `Collection` é uma `interface`.

f) Verdadeira.

g) Falsa. À medida que o fator de carga aumenta, menos posições estão disponíveis em relação ao seu número total, portanto, a possibilidade de colisões aumenta.

h) Falsa. Tentar inserir um elemento `null` causa uma `NullPointerException`.

Exercícios

20.3 Defina cada um dos seguintes termos:

a) `Collection`

b) `Collections`

c) `Comparator`

d) `List`

e) fator de carga

f) colisão

g) relação de troca espaço/tempo em hashing

h) `HashMap`

20.4 Explique brevemente a operação de cada um dos seguintes métodos da classe `Vector`:

a) `add`

b) `set`

c) `remove`

d) `removeAllElements`

e) `removeElementAt`

f) `firstElement`

g) `lastElement`

h) `contains`

i) `indexOf`

j) `size`

k) `capacity`

20.5 Explique por que inserir elementos adicionais em um objeto `Vector` cujo tamanho atual é menor que sua capacidade é uma operação relativamente rápida e por que inserir elementos adicionais em um objeto `Vector` cujo tamanho atual está dentro da capacidade é uma operação relativamente lenta.

20.6 Estendendo a classe `Vector`, os projetistas do Java foram capazes de criar a classe `Stack` rapidamente. Quais são os aspectos negativos dessa utilização de herança, particularmente para a classe `Stack`?

20.7 Responda resumidamente às seguintes perguntas:

a) Qual é a principal diferença entre um `Set` e um `Map`?

b) O que acontece quando você adiciona um valor de tipo primitivo (por exemplo, `double`) a uma coleção?

c) Você pode imprimir todos os elementos em uma coleção sem utilizar um `Iterator`? Se puder, como você os imprime?

20.8 Explique a principal operação de cada um dos seguintes métodos relacionados com `Iterator`:

a) `iterator`

b) `hasNext`

c) `next`

20.9 Explique brevemente a operação de cada um dos seguintes métodos da classe `HashMap`:

a) `put`

b) `get`

c) `isEmpty`

d) `containsKey`

e) `keySet`

20.10 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Os elementos em uma `Collection` devem ser classificados em ordem crescente antes que uma `binarySearch-` possa ser realizada.

b) O método `first` obtém o primeiro elemento em uma `TreeSet`.

c) Uma `List` criada com o método `Arrays asList` é redimensionável.

20.11 Explique a operação de cada um dos seguintes métodos da classe `Properties`:

a) `load`

b) `store`

c) `getProperty`

d) `list`

20.12 Reescreva as linhas 16–25 na Figura 20.3 para que sejam mais concisas utilizando o método `asList` e o construtor `LinkedList` que aceita um argumento `Collection`.

20.13 (***Eliminação de duplicatas***) Escreva um programa que leia em uma série de nomes e elimine os duplicados armazenando-os em um `Set`. Permita que o usuário procure um primeiro nome.

20.14 (***Contando letras***) Modifique o programa da Figura 20.18 para contar o número de ocorrências de cada letra em vez do número de cada palavra. Por exemplo, a string `"HELLO THERE"` contém dois `H`s, três `E`s, dois `L`s, um `O`, um `T` e um `R`. Exiba os resultados.

20.15 (***Seletor de cor***) Utilize uma `HashMap` para criar uma classe reutilizável a fim de escolher uma das 13 cores predefinidas na classe `Color`. Os nomes das cores devem ser utilizados como chaves; e os objetos `Color` predefinidos devem ser utilizados como valores. Coloque essa classe em um pacote que pode ser importado em qualquer programa Java. Utilize sua nova classe em um aplicativo que permita ao usuário selecionar uma cor e desenhar uma forma nessa cor.

20.16 (***Contando palavras duplicadas***) Escreva um programa que determine e imprima o número de palavras duplicadas em uma frase. Trate da mesma maneira letras minúsculas e letras maiúsculas. Ignore a pontuação.

20.17 (***Inserindo elementos em uma `LinkedList` na ordem classificada***) Escreva um programa que insira 25 números inteiros aleatórios de 0 a 100 em um objeto `LinkedList`. O programa deve classificar os elementos e, então, calcular a soma dos elementos e a média de ponto flutuante dos elementos.

20.18 (***Copiando e invertendo `LinkedList`s***) Escreva um programa que cria um objeto `LinkedList` de 10 caracteres e um segundo objeto `LinkedList` que contenha uma cópia da primeira lista, mas na ordem inversa.

20.19 (***Números primos e fatores primos***) Escreva um programa que aceite como entrada um número inteiro e determine se é ou não um número primo. Se o número não for primo, exiba seus fatores primos únicos. Lembre-se de que os fatores de um número primo são somente 1 e o próprio número primo. Cada número que não é primo tem uma fatoração em primos única. Por exemplo, considere o número 54. Os fatores primos de 54 são 2, 3, 3 e 3. Quando os valores são multiplicados, o resultado é 54. Para o número 54, a saída dos fatores primos deve ser 2 e 3. Utilize `Set`s como parte de sua solução.

20.20 (***Classificando palavras com um `TreeSet`***) Escreva um programa que utilize um método `String split` para tokenizar uma linha de texto inserida pelo usuário e coloque cada token em um `TreeSet`. Imprima os elementos do `TreeSet`. [*Nota*: Isso deve fazer com que os elementos sejam impressos na ordem de classificação ascendente.]

20.21 (***Alterando a ordem de classificação de uma `PriorityQueue`***) A saída da Figura 20.15 (`PriorityQueueTest`) mostra que `PriorityQueue` ordena elementos `Double` em ordem crescente. Reescreva a Figura 20.15 de modo que ela ordene os elementos `Double` em ordem decrescente (isto é, `9.8` deve ser o elemento de maior prioridade em vez de `3.2`).

21

Todo gênio vê o mundo de um ângulo diferente de seus pares.
— **Havelock Ellis**

… nossa individualidade especial, enquanto distinta da nossa humanidade genérica.
— **Oliver Wendell Holmes, Sr.**

Born under one law, to another bound. [Nascido sob uma lei, subjugado a outra.]
— **Lorde Brooke**

Você lida com o material bruto da opinião, e, se minhas convicções têm alguma validade, em última instância a opinião governa o mundo.
— **Woodrow Wilson**

Classes e métodos genéricos

Objetivos

Neste capítulo, você aprenderá:

- A criar métodos genéricos que realizam tarefas idênticas em argumentos de diferentes tipos.
- A criar uma classe `Stack` genérica que pode ser utilizada para armazenar objetos de qualquer tipo de classe ou interface.
- A entender como sobrecarregar métodos genéricos com métodos não genéricos ou com outros métodos genéricos.
- A entender tipos brutos e como eles ajudam a alcançar a retrocompatibilidade.
- A utilizar curingas quando informações precisas de tipo sobre um parâmetro não são requeridas no corpo de método.
- A identificar o relacionamento entre herança e genéricos.

Sumário

21.1 Introdução
21.2 Motivação para métodos genéricos
21.3 Métodos genéricos: implementação e tradução em tempo de compilação
21.4 Questões adicionais da tradução em tempo de compilação: métodos que utilizam um parâmetro de tipo como o tipo de retorno
21.5 Sobrecarregando métodos genéricos
21.6 Classes genéricas
21.7 Tipos brutos
21.8 Curingas em métodos que aceitam parâmetros de tipo
21.9 Genéricos e herança: notas
21.10 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

21.1 Introdução

Você já utilizou as classes e os métodos genéricos existentes nos capítulos 7 e 20. Neste capítulo, aprenderá a escrever seus próprios. Também aprenderá os relacionamentos entre os genéricos e outros recursos Java, como sobrecarga e herança.

Seria conveniente se pudéssemos escrever um único método `sort` para classificar os elementos em um array de `Integer`, em um array de `String` ou em um array de qualquer tipo que suporte ordenamento (isto é, seus elementos podem ser comparados). Também seria conveniente se pudéssemos escrever uma única classe `Stack` que seria utilizada como uma `Stack` de inteiros, uma `Stack` de números de ponto flutuante, uma `Stack` de `String`s ou uma `Stack` de qualquer outro tipo. Seria ainda mais conveniente se pudéssemos detectar não correspondências de tipos em tempo de compilação — conhecida como **segurança de tipo em tempo de compilação**. Por exemplo, se uma `Stack` armazenasse somente inteiros, ao tentar inserir uma `String` nessa `Stack` deveria emitir um erro em *tempo de compilação*.

Este capítulo discute **genéricos**, que fornecem significado para criar os modelos gerais já mencionados. **Métodos genéricos** e **classes genéricas** (e interfaces) permitem especificar, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados, respectivamente. Os genéricos também fornecem segurança de tipo em tempo de compilação que permite capturar tipos inválidos em tempo de compilação.

Poderíamos escrever um método genérico para classificar um array de objetos e então invocar o método genérico com arrays de `Integer`s, arrays de `Double`s, arrays de `String`s e assim por diante, para classificar os elementos no array. O compilador realizaria uma verificação de tipo para assegurar que o array passado para o método de classificação contenha elementos do mesmo tipo. Poderíamos escrever uma única classe `Stack` genérica que manipulasse uma pilha de objetos e então instanciasse objetos `Stack` em uma pilha de `Integer`s, uma pilha de `Double`s, uma pilha de `String`s e assim por diante. O compilador realizaria a verificação de tipo para assegurar que a `Stack` armazena elementos do mesmo tipo.

Observação de engenharia de software 21.1

Métodos e classes genéricas estão entre as capacidades mais poderosas do Java para reutilização de software com segurança de tipo em tempo de compilação.

21.2 Motivação para métodos genéricos

Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados. Para encorajar o uso dos métodos genéricos, vamos começar com um exemplo (Figura 21.1) contendo os métodos `printArray` sobrecarregados (linhas 21–28, linhas 31–38 e linhas 41–48) que imprimem representações `String` dos elementos de um array `Integer`, um array `Double` e um array `Character`, respectivamente. Poderíamos ter utilizado array de tipos primitivos `int`, `double` e `char`. Utilizamos arrays das classes empacotadoras de tipo para configurar nosso exemplo de método genérico, porque apenas tipos por referência podem ser utilizados com classes e métodos genéricos.

```
1  // Figura 21.1: OverloadedMethods.java
2  // Imprimindo elementos do array com métodos sobrecarregados.
3  public class OverloadedMethods
4  {
5     public static void main( String[] args )
6     {
7        // cria arrays de Integer, Double e Character
8        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
9        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
```

```
10          Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
11
12          System.out.println( "Array integerArray contains:" );
13          printArray( integerArray ); // passa um array de Integer
14          System.out.println( "\nArray doubleArray contains:" );
15          printArray( doubleArray ); // passa um array Double
16          System.out.println( "\nArray characterArray contains:" );
17          printArray( characterArray ); // passa um array de Character
18       } // fim de main
19
20       // método printArray para imprimir um array de Integer
21       public static void printArray( Integer[] inputArray )
22       {
23          // exibe elementos do array
24          for ( Integer element : inputArray )
25             System.out.printf( "%s ", element );
26
27          System.out.println();
28       } // fim do método printArray
29
30       // método printArray para imprimir um array de Double
31       public static void printArray( Double[] inputArray )
32       {
33          // exibe elementos do array
34          for ( Double element : inputArray )
35             System.out.printf( "%s ", element );
36
37          System.out.println();
38       } // fim do método printArray
39
40       // método printArray para imprimir um array de Character
41       public static void printArray( Character[] inputArray )
42       {
43          // exibe elementos do array
44          for ( Character element : inputArray )
45             System.out.printf( "%s ", element );
46
47          System.out.println();
48       } // fim do método printArray
49    } // fim da classe OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 21.1 | Imprimindo elementos do array com métodos sobrecarregados.

O programa começa declarando e inicializando três arrays — array `Integer integerArray` de seis elementos (linha 8), array `Array doubleArray` de sete elementos (linha 9) e array `Character characterArray` de cinco elementos (linha 10). Em seguida, as linhas 12–17 exibem o conteúdo de cada array.

Quando o compilador encontra uma chamada de método, ele tenta localizar uma declaração de método com o mesmo nome e parâmetros que correspondam aos tipos de argumentos da chamada. Nesse exemplo, cada chamada a `printArray` corresponde a uma das declarações do método `printArray`. Por exemplo, a linha 13 chama `printArray` com `integerArray` como seu argumento. O compilador determina o tipo do argumento (isto é, `Integer[]`) e tenta localizar um método `printArray` que especifica um parâmetro `Integer[]` (linhas 21–28), e então configura uma chamada a esse método. De maneira semelhante, quando o compilador encontra a chamada na linha 15, ele determina o tipo do argumento (isto é, `Double[]`), tenta localizar um método `printArray` que especifica um parâmetro `Double[]` (linhas 31–38) e, então, configura uma chamada a esse método. Por fim, quando o compilador encontra a chamada a `printArray` na linha 17, ele determina o tipo do argumento (isto é, `Character[]`), tenta localizar um método `printArray` que especifica um parâmetro `Character[]` (linhas 41–48) e, então, configura uma chamada a esse método.

Estude cada método `printArray`. Observe que o tipo de elemento do array de tipos aparece no cabeçalho de cada método (linhas 21, 31 e 41) e no cabeçalho da instrução `for` (linhas 24, 34 e 44). Se fôssemos substituir os tipos de elemento em cada método por um nome genérico — `T` por convenção — os três métodos se pareceriam com aquele na Figura 21.2. Parece que se fosse possível substituir o tipo de elemento do array em cada um dos três métodos por um tipo genérico único, seríamos então capazes de declarar um método `printArray`

que pode exibir as representações `String` dos elementos de qualquer array que contém objetos. O método na Figura 21.2 é semelhante à declaração do método `printArray` genérico que discutimos na Seção 21.3.

```
1  public static void printArray( T[] inputArray )
2  {
3     // exibe elementos do array
4     for ( T element : inputArray)
5        System.out.printf( "%s ", element );
6
7     System.out.println();
8  } // fim do método printArray
```

Figura 21.2 | Método `printArray` em que nomes de tipos reais são substituídos pelo nome genérico `T` por convenção.

21.3 Métodos genéricos: implementação e tradução em tempo de compilação

Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos sobrecarregados podem ser codificados mais compacta e convenientemente com um método genérico. Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes. Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método apropriadamente.

A Figura 21.3 reimplementa o aplicativo da Figura 21.1 utilizando um método `printArray` genérico (linhas 22–29). Observe que as chamadas do método `printArray` nas linhas 14, 16 e 18 são idênticas àquelas da Figura 21.1 (linhas 14, 16 e 18) e que as saídas dos dois aplicativos são idênticas. Isso demonstra radicalmente o poder expressivo dos genéricos.

```
1  // Figura 21.3: GenericMethodTest.java
2  // Imprimindo elementos do array com o método genérico printArray.
3
4  public class GenericMethodTest
5  {
6     public static void main( String[] args )
7     {
8        // cria arrays de Integer, Double e Character
9        Integer[] intArray = { 1, 2, 3, 4, 5, 6 };
10       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
11       Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
12
13       System.out.println( "Array integerArray contains:" );
14       printArray( integerArray ); // passa um array de Integer
15       System.out.println( "\nArray doubleArray contains:" );
16       printArray( doubleArray ); // passa um array Double
17       System.out.println( "\nArray characterArray contains:" );
18       printArray( characterArray ); // passa um array de Character
19    } // fim de main
20
21    // método genérico printArray
22    public static < T > void printArray( T[] inputArray )
23    {
24       // exibe elementos do array
25       for ( T element : inputArray )
26          System.out.printf( "%s ", element );
27
28       System.out.println();
29    } // fim do método printArray
30 } // fim da classe GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 21.3 | Imprimindo elementos do array com o método genérico `printArray`.

A linha 22 inicia a declaração do método `printArray`. Todas as declarações de métodos genéricos contêm uma **seção de parâmetro de tipo** delimitado por **colchetes angulares** (< e >) que precede o tipo de retorno do método (< `T` > nesse exemplo). Cada seção de parâmetro de tipo contém um ou mais **parâmetros de tipos** (também chamados **parâmetros de tipo formais**), separados por vírgulas. Um parâmetro de tipo, também conhecido como uma **variável de tipo**, é um identificador que especifica um nome genérico do tipo. Os parâmetros de tipo podem ser utilizados para declarar o tipo de retorno, tipos de parâmetros e tipos de variáveis locais em uma declaração de método genérico e atuam como marcadores de lugar para os tipos dos argumentos passados ao método genérico, conhecidos como **argumentos de tipos reais**. O corpo de um método genérico é declarado como o de qualquer outro método. Observe que parâmetros de tipos podem representar somente tipos por referência — não tipos primitivos (como `int`, `double` e `char`). Também observe que os nomes dos parâmetros de tipos por toda a declaração de método devem corresponder aqueles declarados na seção de parâmetro de tipo. Por exemplo, a linha 25 declara `element` como tipo `T`, que corresponde ao parâmetro de tipo (`T`) declarado na linha 22. Além disso, um parâmetro de tipo pode ser declarado somente uma vez na seção de parâmetro de tipo, mas pode aparecer mais de uma vez na lista de parâmetros do método. Por exemplo, o nome do parâmetro de tipo `T` aparece duas vezes na seguinte lista de parâmetros do método:

```
public static < T > void printTwoArrays( T[] array1, T[] array2 )
```

Os nomes de parâmetro de tipo não precisam ser únicos entre diferentes métodos genéricos.

Erro comum de programação 21.1

Ao declarar um método genérico, não conseguir colocar uma seção de parâmetro de tipo antes do tipo de retorno de um método é um erro de sintaxe — o compilador não entenderá os nomes do parâmetro de tipo quando eles forem encontrados no método.

A seção de parâmetro de tipo do método `printArray` declara o parâmetro do tipo `T` como o marcador de lugar para o tipo de elemento do array que `printArray` enviará para a saída. Observe que `T` aparece na lista de parâmetros como o tipo de elemento do array (linha 22). O cabeçalho da instrução `for` (linha 25) também utiliza `T` como o tipo de elemento. Estas são as duas localizações exatas em que os métodos `printArray` sobrecarregados da Figura 21.1 especificaram `Integer`, `Double` ou `Character` como o tipo de elemento do array. O restante do `printArray` é idêntico às versões apresentadas na Figura 21.1.

Boa prática de programação 21.1

É recomendável que parâmetros de tipos sejam especificados como letras maiúsculas individuais. Tipicamente, um parâmetro de tipo que representa o tipo de um elemento do array (ou outra coleção) é nomeado `T`.

Como na Figura 21.1, o programa começa declarando e inicializando o array `Integer integerArray` de seis elementos (linha 9), o array `Double doubleArray` de sete elementos (linha 10) e o array `Character characterArray` de cinco elementos (linha 11). O programa então gera uma saída para cada array chamando `printArray` (linhas 14, 16 e 18) — uma vez com o argumento `integerArray`, uma vez com o argumento `doubleArray` e uma vez com o argumento `characterArray`.

Quando o compilador encontra a linha 14, ele primeiro determina o tipo do argumento `integerArray` (isto é, `Integer[]`) e tenta localizar um método chamado `printArray` que especifica um único parâmetro `Integer[]`. Não há tal método nesse exemplo. Em seguida, o compilador determina se há um método genérico chamado `printArray` que especifica um parâmetro de array individual e utiliza um parâmetro de tipo para representar o tipo de elemento do array. O compilador determina que o `printArray` (linhas 22–29) é uma correspondência e configura uma chamada ao método. O mesmo processo é repetido para chamadas ao método `printArray` nas linhas 16 e 18.

Erro comum de programação 21.2

Se o compilador não puder encontrar uma correspondência entre uma chamada de método e uma declaração de método genérico ou não genérico, ocorrerá um erro de compilação.

Erro comum de programação 21.3

Se o compilador não encontrar uma declaração de método que corresponda exatamente a uma chamada de método, mas encontrar dois ou mais métodos genéricos que podem satisfazer a chamada de método, ocorrerá um erro de compilação.

Além de configurar as chamadas de método, o compilador também determina se as operações no corpo do método podem ser aplicadas a elementos do tipo armazenado no argumento do array. A única operação executada nos elementos do array nesse exemplo é gerar saída para sua representação `String`. A linha 26 realiza uma chamada a `toString` implícita em cada `element`. Para trabalhar com genéricos, todo elemento do array deve ser um objeto de um tipo de classe ou interface. Como todos os objetos têm um método `toString`, o compilador fica satisfeito com fato de que a linha 26 realiza uma operação válida para qualquer objeto no argumento do array do `printArray`. Os métodos `toString` das classes `Integer`, `Double` e `Character` retornam a representação de `String` dos valores subjacentes do valor `int`, `double` ou `char`, respectivamente.

Erasure em tempo de compilação

Quando o compilador traduz o método genérico `printArray` em bytecodes Java, ele remove a seção de parâmetro de tipo e substitui os parâmetros de tipo por tipos reais. Esse processo é conhecido como **erasure**. Por padrão, todos os tipos genéricos são substituídos pelo tipo `Object`. Assim, a versão compilada do método `printArray` aparece como mostrada na Figura 21.4 — há somente uma cópia desse código utilizada para todas as chamadas a `printArray` no exemplo. Isso é bem diferente de outros mecanismos semelhantes, como templates do C++ em que uma cópia separada do código-fonte é gerada e compilada para cada tipo passado como um argumento para o método. Como veremos na Seção 21.4, a tradução e compilação dos genéricos é um pouco mais complicada do que aquilo que foi discutido nesta seção.

Declarando `printArray` como um método genérico na Figura 21.3, eliminamos a necessidade dos métodos sobrecarregados da Figura 21.1, poupando 19 linhas de código e criando um método reutilizável que pode gerar saída das representações de `String` dos elementos em qualquer array que contém objetos. Entretanto, esse exemplo em particular poderia simplesmente ter declarado o método `printArray` como mostrado na Figura 21.4, utilizando um array `Object` como o parâmetro. Isso produziria os mesmos resultados, pois qualquer `Object` pode ser enviado para a saída como uma `String`. Em um método genérico, os benefícios tornam-se aparentes quando o método também utiliza um parâmetro de tipo, o tipo de retorno do método, como demonstraremos na próxima seção.

```
1  public static void printArray(Object [] inputArray )
2  {
3     // exibe elementos do array
4     for (Object element : inputArray )
5        System.out.printf( "%s ", element );
6
7     System.out.println();
8  } // fim do método printArray
```

Figura 21.4 | O método genérico `printArray` depois de a erasure ser realizada pelo compilador.

21.4 Questões adicionais da tradução em tempo de compilação: métodos que utilizam um parâmetro de tipo como tipo de retorno

Vamos considerar o exemplo de um método genérico em que parâmetros de tipo são utilizados no tipo de retorno e na lista de parâmetros (Figura 21.5). O aplicativo utiliza um método genérico `maximum` para determinar e retornar o maior dos seus três argumentos do mesmo tipo. Infelizmente, o operador relacional > não pode ser utilizado com tipos por referência. Entretanto, é possível comparar dois objetos da mesma classe se essa classe implementar a **interface** genérica **`Comparable<T>`** (pacote `java.lang`). Todas as classes empacotadoras de tipo para tipos primitivos implementam essa interface. Como ocorre com as classes genéricas, as **interfaces genéricas** permitem especificar, com uma única declaração de interface, um conjunto de tipos relacionados. Objetos `Comparable<T>` têm um **método `compareTo`**. Por exemplo, se houver dois objetos `Integer`, `integer1` e `integer2`, eles poderão ser comparados com a expressão:

```
integer1.compareTo( integer2 )
```

É sua responsabilidade ao declarar uma classe que implementa `Comparable<T>` para declarar o método `compareTo` a fim de ele comparar o conteúdo dos dois objetos dessa classe e retornar os resultados da comparação. O método deve voltar 0 se os objetos forem iguais, um número inteiro negativo se `object1` for menor que `object2` ou um número inteiro positivo se `object1` for maior que `object2`. Por exemplo, o método `compareTo` da classe `Integer` compara os valores de `int` armazenados em dois objetos `Integer`. Um benefício da implementação da interface `Comparable<T>` é que objetos `Comparable<T>` podem ser utilizados com os métodos de classificação e pesquisa da classe `Collections` (pacote `java.util`). Discutimos esses métodos no Capítulo 20, "Coleções genéricas". Neste exemplo, utilizaremos o método `compareTo` no método `maximum` para ajudar a determinar o maior valor.

```
1  // Figura 21.5: MaximumTest.java
2  // O método genérico maximum retorna o maior dos três objetos.
3
4  public class MaximumTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
9           maximum( 3, 4, 5 ) );
10       System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
11          6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
12       System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
```

```
13            "apple", "orange", maximum( "pear", "apple", "orange" ) );
14      } // fim de main
15
16      // determina o maior dos três objetos Comparable
17      public static < T extends Comparable< T > > T maximum( T x, T y, T z )
18      {
19         T max = x; // supõe que x é inicialmente o maior
20
21         if ( y.compareTo( max ) > 0 )
22            max = y; // y é o maior até agora
23
24         if ( z.compareTo( max ) > 0 )
25            max = z; // z é o maior
26
27         return max; // retorna o maior objeto
28      } // fim do método Maximum
29   } // fim da classe MaximumTest
```

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Figura 21.5 | Método genérico `maximum` com um limite superior no seu parâmetro de tipo.

Método genérico `maximum`

O método genérico `maximum` (linhas 17–28) utiliza o parâmetro de tipo `T` como o tipo de retorno do método (linha 17), como o tipo dos parâmetros do método de `x`, `y` e `z` (linha 17) e como o tipo da variável local `max` (linha 19). A seção de parâmetro de tipo especifica que `T` estende `Comparable<T>` — somente objetos das classes que implementam a interface `Comparable<T>` podem ser utilizados com esse método. Nesse caso, `Comparable` é conhecido como o **limite superior** do parâmetro de tipo. Por padrão, `Object` é o limite superior. Observe que declarações do parâmetro de tipo que limitam o parâmetro sempre utilizam a palavra-chave `extends` independentemente de o parâmetro de tipo estender uma classe ou implementar uma interface. Esse parâmetro de tipo é mais restritivo do que aquele especificado para `printArray` na Figura 21.3, o qual foi capaz de gerar a saída de arrays contendo qualquer tipo de objeto. A restrição à utilização de objetos `Comparable<T>` é importante, pois nem todos os objetos podem ser comparados. Entretanto, é garantido que objetos `Comparable<T>` terão um método `compareTo`.

O método `maximum` utiliza o mesmo algoritmo utilizado na Seção 6.4 para determinar o maior dos seus três argumentos. O método supõe que seu primeiro argumento (`x`) é o maior e o atribui à variável local `max` (linha 19). Em seguida, a instrução `if` nas linhas 21–22 determina se `y` é maior que `max`. A condição invoca o método `compareTo` de `y` com a expressão `y.compareTo(max)`, que retorna um número inteiro negativo, `0` ou número inteiro positivo, para determinar o relacionamento com `max` de `y`. Se o valor de retorno de `compareTo` for maior que `0`, então `y` é maior e será atribuído à variável `max`. De maneira semelhante, a instrução `if` nas linhas 24–25 determina se `z` é maior que `max`. Se for, a linha 25 atribui `z` a `max`. Então, a linha 27 retorna `max` ao chamador.

Chamando o método `maximum`

Em `main` (linhas 6–14), a linha 9 chama `maximum` com os inteiros `3`, `4` e `5`. Quando o compilador encontra essa chamada, ele primeiro procura um método `maximum` que recebe três argumentos do tipo `int`. Não há tal método, assim o compilador procura um método genérico que possa ser utilizado e encontra o método genérico `maximum`. Entretanto, lembre-se de que os argumentos para um método genérico devem ser de um tipo por referência. O compilador assim converte por autoboxing os três valores de `int` em objetos `Integer` e especifica que os três objetos `Integer` serão passados para `maximum`. Observe que a classe `Integer` (pacote `java.lang`) implementa a interface `Comparable<Integer>` de tal maneira que o método `compareTo` compara os valores `int` em dois objetos `Integer`. Portanto, `Integers` são argumentos válidos para o método `maximum`. Quando o `Integer` representando o valor máximo é retornado, tentamos enviá-lo para a saída com o especificador de formato `%d`, que gera a saída de um valor de tipo primitivo `int`. Dessa forma, o valor de retorno de `maximum` é impresso como um valor `int`.

Um processo semelhante ocorre para os três argumentos `double` passados para `maximum` na linha 11. Cada `double` é convertido por autoboxing em um objeto `Double` e passado para `maximum`. Novamente, isso é permitido porque a classe `Double` (pacote `java.lang`) implementa a interface `Comparable<Double>`. O `Double` retornado por `maximum` é gerado com o especificador de formato `%.1f`, que envia para a saída um valor de tipo primitivo `double`. Portanto, o valor de retorno de `maximum` é convertido por auto-unboxing e enviado para a saída como um `double`. A chamada a `maximum` na linha 13 recebe três `Strings`, que também são objetos `Comparable<String>`. Observe que colocamos intencionalmente o maior valor em uma posição diferente em cada chamada de método (linhas 9, 11 e 13) para mostrar que o método genérico sempre encontra o valor máximo, independentemente da sua posição na lista de argumentos.

Limite superior de um parâmetro de tipo

Quando o compilador traduz o método genérico `maximum` em bytecodes Java, ele utiliza a técnica de erasure (apresentada na Seção 21.3) para substituir os parâmetros de tipo por tipos reais. Na Figura 21.3, todos os tipos genéricos foram substituídos pelo tipo `Object`. Na verdade, todos os parâmetros de tipo são substituídos por aquilo que se denomina limite superior do parâmetro de tipo, que é especificado na seção do parâmetro de tipo. Para indicar o limite superior, depois do nome do parâmetro de tipo posicione a palavra-chave `extends` e o nome da classe ou interface que representa o limite superior, ou uma lista separada por vírgula dos tipos que representam o limite superior. A lista pode conter zero ou uma classe e zero ou mais interfaces. Por exemplo, a seção do parâmetro de tipo do método `maximum` (Figura 21.5), especificamos o limite superior do parâmetro de tipo `T` como o tipo `Comparable<T>` desta maneira:

```
T extends Comparable< T >
```

Assim, somente objetos `Comparable<T>` podem ser passados como argumentos para `maximum` — qualquer coisa além de um `Comparable<T>` resultará em erros de compilação. A menos que especificado de outro modo, `Object` é o limite superior padrão. A Figura 21.6 simula a erasure de tipos do método maximum, mostrando o código-fonte do método depois de a seção de parâmetro de tipo ter sido removida e o parâmetro de tipo `T` ter sido substituído pelo limite superior, `Comparable`, por toda a declaração de método. Observe que a erasure de `Comparable<T>` é simplesmente `Comparable`.

```
1  public static Comparable maximum(Comparable x, Comparable y, Comparable z)
2  {
3     Comparable max = x; // supõe que x é inicialmente o maior
4
5     if ( y.compareTo( max ) > 0 )
6        max = y; // y é o maior até agora
7
8     if ( z.compareTo( max ) > 0 )
9        max = z; // z é o maior
10
11    return max; // retorna o maior objeto
12 } // fim do método Maximum
```

Figura 21.6 | O método genérico `maximum` depois de a erasure ser realizada pelo compilador.

Depois da erasure, a versão compilada do método `maximum` especifica que ele retorna o tipo `Comparable`. Entretanto, o método chamador não espera receber um `Comparable`. Ele espera receber um objeto do mesmo tipo que foi passado para `maximum` como um argumento — `Integer`, `Double` ou `String` nesse exemplo. Quando o compilador substitui as informações do parâmetro de tipo pelo tipo do limite superior na declaração do método, ele também insere operações explícitas de coerção na frente de cada chamada de método para assegurar que o valor retornado é do tipo esperado pelo chamador. Portanto, a chamada a `maximum` na linha 9 (Figura 21.5) é precedida por uma coerção para `Integer`, como em

```
(Integer) maximum( 3, 4, 5 )
```

a chamada a `maximum` na linha 11 é precedida por uma coerção em `Double`, como em

```
(Double) maximum( 6.6, 8.8, 7.7 )
```

e a chamada a `maximum` na linha 13 é precedida por uma coerção em `String`, como em

```
(String) maximum( "pear", "apple", "orange" )
```

Em cada caso, o tipo da coerção para o valor de retorno é inferido a partir dos tipos dos argumentos de método na chamada de método particular porque, de acordo com a declaração do método, o tipo de retorno e os tipos de argumento correspondem.

Possíveis `ClassCastException`s

Nesse exemplo, você não pode utilizar um método que aceita `Objetos`, porque a classe `Object` só fornece uma comparação de igualdade. Além disso, sem genéricos, você seria responsável por implementar a operação de coerção. O uso de genéricos assegura que a coerção inserida nunca lançará uma `ClassCastException`, supondo que genéricos sejam utilizados por todo seu código (isto é, você não misturou código antigo com código novo de genéricos).

21.5 Sobrecarregando métodos genéricos

Um método genérico pode ser sobrecarregado. Uma classe pode fornecer dois ou mais métodos genéricos que especificam o mesmo nome de método, mas diferentes parâmetros de método. Por exemplo, o método genérico `printArray` da Figura 21.3 poderia ser sobrecarregado por outro método genérico `printArray` com os parâmetros adicionais `lowSubscript` e `highSubscript` para especificar a parte do array que será enviada para a saída (ver Exercício 21.5).

Um método genérico também pode ser sobrecarregado por métodos não genéricos. Quando o compilador encontra uma chamada de método, ele procura a declaração de método que corresponde mais precisamente ao nome de método e aos tipos de argumentos especificados na chamada. Por exemplo, o método genérico `printArray` da Figura 21.3 poderia ser sobrecarregado por uma versão específica para `Strings`, que gera a saída de `Strings` em um formato tabular elegante (ver Exercício 21.6).

Quando o compilador encontra uma chamada de método, ele realiza um processo de correspondência para determinar qual método invocar. O compilador tenta encontrar e utilizar uma correspondência precisa em que o nome do método e os tipos de argumento da chamada de método correspondam àqueles da declaração de um método específico. Se não houver um método desse, o compilador tenta localizar um método com tipos compatíveis ou um método genérico correspondente.

21.6 Classes genéricas

O conceito de uma estrutura de dados, como uma pilha, pode ser entendido independentemente do tipo de elemento que ela manipula. Classes genéricas fornecem um meio de descrever o conceito de uma pilha (ou de qualquer outra classe) de uma maneira independente do tipo. Podemos então instanciar objetos específicos de tipo da classe genérica. Essa capacidade fornece uma excelente oportunidade de reutilização de software.

Como visto no Capítulo 20, uma vez que tenha uma classe genérica, você pode utilizar uma notação concisa e simples para indicar o(s) tipo(s) real(is) que deve(m) ser utilizado(s) no lugar do(s) parâmetro(s) de tipo da classe. Em tempo de compilação, o compilador Java garante a segurança de tipo do seu código e utiliza as técnicas de erasure descritas nas seções 21.3–21.4 para permitir que o código do seu cliente interaja com a classe genérica.

Uma classe `Stack` genérica, por exemplo, poderia ser a base para criar muitas classes `Stack` lógicas (por exemplo, "`Stack` de `Double`", "`Stack` de `Integer`", "`Stack` de `Character`", "`Stack` de `Employee`"). Essas classes são conhecidas como **classes parametrizadas** ou **tipos parametrizados** porque aceitam um ou mais parâmetros. Lembre-se de que parâmetros de tipo só representam tipos por referência, o que significa que a classe `Stack` genérica não pode ser instanciada com tipos primitivos. Entretanto, é possível instanciar uma `Stack` que armazena objetos das classes empacotadoras de tipo do Java e permitir que o Java utilize o autoboxing para converter os valores primitivos em objetos. Lembre-se de que o autoboxing ocorre quando o valor de um tipo primitivo (por exemplo, `int`) é inserido em uma `Stack` que contém objetos da classe empacotadora (por exemplo, `Integer`). O auto-unboxing ocorre quando um objeto da classe empacotadora é removido da `Stack` e atribuído a uma variável de tipo primitivo.

Implementando uma classe `Stack` genérica

A Figura 21.7 apresenta uma declaração da classe `Stack` genérica. Uma declaração de classe genérica se parece com uma não genérica, mas o nome da classe é seguido por uma seção do parâmetro de tipo (linha 5). Nesse caso, o parâmetro de tipo `T` representa o tipo de elemento que a `Stack` manipulará. Como ocorre com métodos genéricos, a seção de parâmetro de tipo de uma classe genérica pode ter um ou mais parâmetros de tipo separado por vírgulas. (Você criará uma classe genérica com dois parâmetros de tipo no Exercício 21.8.) O parâmetro de tipo `T` é utilizado por toda a declaração de classe `Stack` para representar o tipo de elemento. [*Nota*: Esse exemplo implementa uma `Stack` como uma `ArrayList`.]

```
1  // Figura 21.7: Stack.java
2  // Empilha a declaração de classe genérica.
3  import java.util.ArrayList;
4
5  public class Stack< T >
6  {
7     private ArrayList< T > elements; // ArrayList armazena elementos de pilha
8
9     // construtor sem argumento cria uma pilha do tamanho padrão
10    public Stack()
11    {
12       this( 10 ); // tamanho padrão da pilha
13    } // fim do construtor sem argumentos da classe Stack
14
15    // construtor cria uma pilha com o número especificado de elementos
16    public Stack( int capacity )
17    {
18       int initCapacity = capacity > 0 ? capacity : 10; // valida
19       elements = new ArrayList< T >( initCapacity ); // cria a ArrayList
20    } // fim do construtor Stack de um argumento
21
22    // insere o elemento na pilha
23    public void push( T pushValue )
24    {
25       elements.add( pushValue ); // insere pushValue na Stack
```

```
26      } // fim do método push
27
28      // retorna o elemento superior se não estiver vazia; do contrário lança uma EmptyStackException
29      public T pop()
30      {
31         if ( elements.isEmpty() ) // se a pilha estiver vazia
32            throw new EmptyStackException( "Stack is empty, cannot pop" );
33
34         // remove e retorna o elemento superior da Stack
35         return elements.remove( elements.size() - 1 );
36      } // fim do método pop
37   } // fim da classe Stack < T >
```

Figura 21.7 | Declaração da classe genérica `Stack`.

A classe `Stack` declara a variável `elements` como um `ArrayList<T>` (linha 7). Essa `ArrayList` armazenará os elementos da `stack`. Como você sabe, uma `ArrayList` pode crescer dinamicamente, assim objetos da nossa classe `Stack` também podem crescer dinamicamente. O construtor sem argumento da classe `Stack` (linhas 10–13) invoca o construtor de um argumento (linhas 16–20) para criar uma `Stack` no qual a `ArrayList` subjacente tem uma capacidade de `10` elementos. O construtor de um argumento também pode ser chamado diretamente para criar uma `Stack` com uma capacidade inicial especificada. A linha 18 valida o argumento do construtor. A linha 19 cria a `ArrayList` com a capacidade especificada (ou 10 se a capacidade for inválida).

O método `push` (linhas 23–26) utiliza o método `add` `ArrayList` para acrescentar o item inserido ao final de `ArrayList` `elements`. O último elemento em `ArrayList` representa a parte superior da pilha.

O método `pop` (linhas 29–36) primeiro determina se está sendo feita uma tentativa de remover um elemento de uma `Stack` vazia. Nesse caso, a linha 32 lança uma `EmptyStackException` (declarada na Figura 21.8). Do contrário, a linha 35 retorna o elemento na parte superior da `Stack` removendo o último elemento na `ArrayList` subjacente.

A classe `EmptyStackException` (Figura 21.8) fornece um construtor sem argumento e um construtor de um argumento. O construtor sem argumentos configura a mensagem de erro padrão e o construtor de um argumento configura uma mensagem de erro personalizada.

```
1   // Figura 21.8: EmptyStackException.java
2   // Declaração da classe EmptyStackException.
3   public class EmptyStackException extends RuntimeException
4   {
5      // construtor sem argumento
6      public EmptyStackException()
7      {
8         this( "Stack is empty" );
9      } // fim do construtor sem argumentos de EmptyStackException
10
11     // construtor de um argumento
12     public EmptyStackException( String message )
13     {
14        super( message );
15     } // fim do construtor de EmptyStackException de um argumento
16  } // fim da classe EmptyStackException
```

Figura 21.8 | Declaração de classe `EmptyStackException`.

Como ocorre com métodos genéricos, quando uma classe genérica é compilada, o compilador executa a técnica de erasure nos parâmetros de tipo da classe e os substitui pelos seus limites superiores. Para a classe `Stack` (Figura 21.7), nenhum limite superior é especificado, portanto o limite superior padrão, `Object`, é utilizado. O escopo do parâmetro de tipo de uma classe genérica é a classe inteira. Entretanto, parâmetros de tipo não podem ser utilizados nas declarações de variáveis `static` de uma classe.

Testando a classe `Stack` genérica da Figura 21.7

Agora, vamos considerar o aplicativo (Figura 21.9) que utiliza a classe genérica `Stack` (Figura 21.7). As linhas 12–13 criam e inicializam variáveis do tipo `Stack<Double>` (pronuncia-se "`Stack` de `Double`") e `Stack<Integer>` (pronuncia-se "`Stack` de `Integer`"). Os tipos `Double` e `Integer` são conhecidos como **argumentos de tipo** `Stack`. Eles são utilizados pelo compilador para substituir os parâmetros de tipo de modo que o compilador possa executar a verificação de tipo e inserir operações de coerção conforme necessário. Discutiremos as operações de coerção em mais detalhes a seguir. As linhas 12–13 instanciam `doubleStack` com uma capacidade de `5` e `integerStack` com uma capacidade de 10 (o padrão). As linhas 16–17 e 20–21 chamam os métodos `testPushDouble` (linhas 25–36), `testPopDouble` (linhas 39–59), `testPushInteger` (linhas 62–73) e `testPopInteger` (linhas 76–96), respectivamente, para demonstrar as duas `Stacks` nesse exemplo.

```
1  // Figura 21.9: Stacktest.java
2  // Programa de teste da classe genérica Stack.
3
4  public class StackTest
5  {
6     public static void main( String[] args )
7     {
8        double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
9        int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11       // Cria uma Stack< Double > e uma Stack< Integer
12       Stack< Double > doubleStack = new Stack< Double >( 5 );
13       Stack< Integer > integerStack = new Stack< Integer >();
14
15       // insere os elementos de doubleElements em doubleStack
16       testPushDouble( doubleStack, doubleElements );
17       testPopDouble( doubleStack ); // remove de doubleStack
18
19       // insere os elementos de integerElements em integerStack
20       testPushInteger( integerStack, integerElements );
21       testPopInteger( integerStack ); // remove de integerStack
22    } // fim de main
23
24    // testa o método push com a pilha de doubles
25    private static void testPushDouble(
26       Stack< Double > stack, double[] values )
27    {
28       System.out.println( "\nPushing elements onto doubleStack" );
29
30       // insere elementos na Stack
31       for ( double value : values )
32       {
33          System.out.printf( "%.1f ", value );
34          stack.push( value ); // insere em doubleStack
35       } // for final
36    } // fim do método testPushDouble
37
38    // testa o método pop com a pilha de doubles
39    private static void testPopDouble( Stack< Double > stack )
40    {
41       // remove elementos da pilha
42       try
43       {
44          System.out.println( "\nPopping elements from doubleStack" );
45          double popValue; // armazena o elemento removido da pilha
46
47          // remove todos os elementos da Stack
48          while ( true )
49          {
50             popValue = stack.pop(); // remove de doubleStack
51             System.out.printf( "%.1f ", popValue );
52          } // fim do while
53       } // fim do try
54       catch( EmptyStackException emptyStackException )
55       {
56          System.err.println();
57          emptyStackException.printStackTrace();
58       } // fim da captura de EmptyStackException
59    } // fim do método testPopDouble
60
61    // testa o método push com a pilha de integers
62    private static void testPushInteger(
63       Stack< Integer > stack, int[] values )
64    {
65       System.out.println( "\nPushing elements onto integerStack" );
66
67       // insere elementos na Stack
68       for ( int value : values )
```

```
69          {
70             System.out.printf( "%d ", value );
71             stack.push( value ); // insere em integerStack
72          } // for final
73       } // fim do método testPushInteger
74
75       // testa o método pop com a pilha de integers
76       private static void testPopInteger( Stack< Integer > stack )
77       {
78          // remove os elementos da pilha
79          try
80          {
81             System.out.println( "\nPopping elements from integerStack" );
82             int popValue; // armazena o elemento removido da pilha
83
84             // remove todos os elementos da Stack
85             while ( true )
86             {
87                popValue = stack.pop(); // remove de intStack
88                System.out.printf( "%d ", popValue );
89             } // fim do while
90          } // fim do try
91          catch( EmptyStackException emptyStackException )
92          {
93             System.err.println();
94             emptyStackException.printStackTrace();
95          } // fim da captura de EmptyStackException
96       } // fim do método testPopInteger
97    } // fim da classe StackTest
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopDouble(StackTest.java:50)
        at StackTest.main(StackTest.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest.testPopInteger(StackTest.java:87)
        at StackTest.main(StackTest.java:21)
```

Figura 21.9 | Programa de teste da classe genérica `Stack`.

Métodos `testPushDouble` e `testPopDouble`

O método `testPushDouble` (linhas 25–36) invoca o método `push` (linha 34) para inserir os valores `double` 1.1, 2.2, 3.3, 4.4 e 5.5 do array `doubleElements` em `doubleStack`. Observe que o autoboxing ocorre na linha 34 quando o programa tenta inserir um valor de tipo primitivo `double` em `doubleStack`, que armazena somente referências a objetos `Double`.

O método `testPopDouble` (linhas 39–59) invoca o método `Stack` `pop` (linha 50) em um loop `while` infinito (linhas 48–52) para remover todos os valores da pilha. Observe na saída que os valores são de fato `removidos` na ordem "último a entrar, primeiro a sair" (a característica definidora das pilhas). Quando o loop tenta utilizar `pop` para remover um sexto valor, a `doubleStack` estará vazia, portanto `pop` lança uma `EmptyStackException`, o que faz o programa prosseguir para o bloco `catch` (linhas 54–58) a fim de tratar a exceção. O rastreamento da pilha indica a exceção que ocorreu e mostra que o método `Stack` `pop` gerou a exceção na linha 32 do arquivo `Stack.java` (Figura 21.7). O rastreamento também mostra que o método `pop` foi chamado pelo método `StackTest` `testPopDouble` na linha 50 de `StackTest.java` e que o método `testPopDouble` foi chamado a partir do método `main` na linha 17 de `StackTest.java`. Essas informações permitem determinar os métodos que estavam na pilha de chamadas de métodos no momento em que a exceção ocorreu. Como o programa captura a exceção, ela é considerada como tendo sido tratada e o programa pode continuar a executar.

O auto-unboxing ocorre na linha 50 quando o programa atribui o objeto `Double` removido da pilha a uma variável primitiva `double`. A partir da Seção 21.4, lembre-se de que o compilador insere coerções para assegurar que os tipos adequados sejam retornados a partir de métodos genéricos. Depois da erasure, o método `pop` `Stack` retorna o tipo `Object`, mas o código cliente em `testPopDouble` espera receber um `double` quando o método `pop` retorna. Assim, o compilador insere uma coerção `Double`, como em

```
popValue = ( Double ) stack.pop();
```

O valor atribuído a `popValue` passará pelo unboxing do objeto `Double` retornado por `pop`.

Métodos `testPushInteger` e `testPopInteger`

O método `testPushInteger` (linhas 62–73) invoca o método `push` `Stack` para inserir valores em `integerStack` até que ela esteja cheia. O método `testPopInteger` (linhas 76–96) invoca o método `pop` `Stack` para remover valores de `integerStack`. Mais uma vez, observe que os valores são removidos na ordem "último a entrar, primeiro a sair". Durante o processo de erasure, o compilador reconhece que o código de cliente no método `testPopInteger` espera receber um `int` quando o método `pop` retorna. Dessa forma, o compilador insere uma coerção em `Integer`, como em

```
popValue = ( Integer ) stack.pop();
```

O valor atribuído a `popValue` passará pelo unboxing do objeto `Integer` retornado por `pop`.

Criando métodos genéricos para testar a classe `Stack<T>`

Observe que o código nos métodos `testPushDouble` e `testPushInteger` é quase idêntico a inserir valores em uma `Stack<Double>` ou em uma `Stack<Integer>`, respectivamente, e o código nos métodos `testPopDouble` e `testPopInteger` é quase idêntico a remover valores de uma `Stack<Double>` ou de uma `Stack<Integer>`, respectivamente. Isso apresenta outra oportunidade de utilizar métodos genéricos. A Figura 21.10 declara o método genérico `testPush` (linhas 24–35) para realizar as mesmas tarefas, como `testPushDouble` e `testPushInteger` na Figura 21.9 — isto é, inserir (`push`) valores em uma `Stack<T>`. De maneira semelhante, o método genérico `testPop` (linhas 38–58) executa as mesmas tarefas que `testPopDouble` e `testPopInteger` na Figura 21.9 — isto é, utiliza `pop` para remover valores de uma `Stack<T>`. Observe que a saída da Figura 21.10 corresponde precisamente àquela da Figura 21.9.

```
1  // Figura 21.10: StackTest2.java
2  // Passando objetos Stack genéricos para métodos genéricos.
3  public class StackTest2
4  {
5     public static void main( String[] args )
6     {
7        Double [] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
8        Integer [] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9
10       // Cria uma Stack< Double > e uma Stack< Integer
11       Stack< Double > doubleStack = new Stack< Double >( 5 );
12       Stack< Integer > integerStack = new Stack< Integer >();
13
14       // insere os elementos de doubleElements em doubleStack
15       testPush( "doubleStack", doubleStack, doubleElements );
16       testPop( "doubleStack", doubleStack ); // remove de doubleStack
17
18       // insere os elementos de integerElements em integerStack
19       testPush( "integerStack", integerStack, integerElements );
20       testPop( "integerStack", integerStack ); // remove de integerStack
21    } // fim de main
22
23    // método genérico testPush insere elementos em uma Stack
24    public static < T > void testPush( String name , Stack< T > stack,
25       T[] elements )
26    {
27       System.out.printf( "\nPushing elements onto %s\n", name );
28
29       // insere elementos na Stack
30       for ( T element : elements )
31       {
32          System.out.printf( "%s ", element );
33          stack.push( element ); // insere o elemento na pilha
34       } // for final
35    } // fim do método testPush
```

```
36
37      // método genérico testPop remove elementos de uma Stack
38      public static < T > void testPop( String name, Stack< T > stack )
39      {
40         // remove elementos da pilha
41         try
42         {
43            System.out.printf( "\nPopping elements from %s\n", name );
44            T popValue; // armazena o elemento removido da pilha
45
46            // remove todos os elementos da Stack
47            while ( true )
48            {
49               popValue = stack.pop();
50               System.out.printf( "%s ", popValue );
51            } // fim do while
52         } // fim do try
53         catch( EmptyStackException emptyStackException )
54         {
55            System.out.println();
56            emptyStackException.printStackTrace();
57         } // fim da captura de EmptyStackException
58      } // fim do método testPop
59   } // fim da classe StackTest2
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:17)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at StackTest2.testPop(StackTest2.java:50)
        at StackTest2.main(StackTest2.java:21
```

Figura 21.10 | Passando objetos `Stack` genéricos para métodos genéricos.

As linhas 11–12 criam os objetos `Stack<Double>` e `Stack<Integer>`, respectivamente. As linhas 15–16 e 19–20 invocam os métodos genéricos `testPush` e `testPop` para testar os objetos `Stack`. Lembre-se de que parâmetros de tipo só podem representar tipos por referência. Portanto, para ser capaz de passar arrays `doubleElements` e `integerElements` para o método genérico `testPush`, os arrays declarados nas linhas 7–8 devem ser declarados com os tipos empacotadores `Double` e `Integer`. Quando esses arrays são inicializados com valores de tipos primitivos, o compilador autoempacota cada valor de tipo primitivo.

O método genérico `testPush` (linhas 24–35) utiliza o parâmetro de tipo `T` (especificado na linha 24) para representar o tipo de dados armazenado na `Stack<T>`. O método genérico recebe três argumentos — uma `String` que representa o nome do objeto `< Stack<T>` para propósitos de saída, uma referência a um objeto de tipo `Stack<T>` e um array de tipo `T` — o tipo de elementos que será colocado em `Stack<T>`. Observe que o compilador impõe uma consistência entre o tipo da `Stack` e os elementos que serão colocados na `Stack` quando o método `push` é invocado, que é o valor real da chamada do método genérico. O método genérico `testPop` (linhas 38–58) recebe dois argumentos — uma `String` que representa o nome do objeto `Stack<T>` para propósitos de saída e uma referência a um objeto do tipo `Stack<T>`.

21.7 Tipos brutos

Os programas de teste para a classe genérica `Stack` na Seção 21.6 instanciam `Stacks` com argumentos do tipo `Double` e `Integer`. Também há a possibilidade de instanciar uma classe genérica `Stack` sem especificar um argumento de tipo, como a seguir:

```
Stack objectStack = new Stack( 5 ); // nenhum argumento de tipo especificado
```

Nesse caso, diz-se que o `objectStack` tem um **tipo bruto**, o que significa que o compilador utiliza implicitamente o tipo `Object` por toda a classe genérica para cada argumento de tipo. Assim, a instrução precedente cria uma `Stack` que pode armazenar objetos de qualquer tipo. Isso é importante para retrocompatibilidade com versões anteriores do Java. Por exemplo, todas as estruturas de dados do Java Collections Framework (ver Capítulo 20, Coleções genéricas) armazenavam referências a `Objects`, mas agora são implementadas como tipos genéricos.

Uma variável `Stack` de tipo bruto pode ser atribuída a uma `Stack` que especifica um argumento de tipo, como um objeto `Stack <Double>`, como a seguir:

```
Stack rawTypeStack2 = new Stack< Double >( 5 );
```

porque o tipo `Double` é uma subclasse de `Object`. Essa atribuição é permitida porque os elementos em uma `Stack<Double>` (isto é, objetos `Double`) são certamente objetos — a classe `Double` é uma subclasse indireta de `Object`.

De maneira semelhante, uma variável `Stack` que especifica um argumento de tipo na sua declaração pode ser atribuída a um tipo bruto do objeto `Stack`, como em:

```
Stack< Integer > integerStack = new Stack( 10 );
```

Embora essa atribuição seja permitida, é perigosa porque uma `Stack` do tipo bruto armazenaria outros tipos além de `Integer`. Nesse caso, o compilador emite uma mensagem de alerta que indica a atribuição insegura.

Utilizando tipos brutos com a classe genérica Stack

O programa de teste da Figura 21.11 utiliza a noção de tipos brutos. A linha 11 instancia a classe genérica `Stack` com o tipo bruto, o que indica que `rawTypeStack1` pode conter objetos de qualquer tipo. A linha 14 atribui uma `Stack<Double>` à variável `rawTypeStack2`, que é declarada como uma `Stack` do tipo bruto. A linha 17 atribui uma `Stack` do tipo bruto à variável `Stack <Integer>` , o que é válido, mas faz com que o compilador emita uma mensagem de alerta (Figura 21.12) indicando uma atribuição potencialmente insegura — mais uma vez, isso ocorre porque uma `Stack` do tipo bruto armazenaria outros tipos além de `Integer`. Além disso, as chamadas aos métodos genéricos `testPush` e `testPop` nas linhas 19–22 resulta em mensagens de alerta do compilador (Figura 21.12). Isso ocorre porque `rawTypeStack1` e `rawTypeStack2` são declaradas como `Stacks` do tipo bruto, mas cada um dos métodos `testPush` e `testPop` espera um segundo argumento que é uma `Stack` com um argumento de tipo específico. Esses alertas indicam que o compilador não pode garantir o fato de que os tipos manipulados pelas pilhas são os tipos corretos, na medida em que não fornecemos uma variável declarada com um argumento de tipo. Os métodos `testPush` (linhas 28–39) e `testPop` (linhas 42–62) são o mesmo como na Figura 21.10.

```
1  // Figura 21.11: RawTypeTest.java
2  // Programa de teste de tipos brutos.
3  public class RawTypeTest
4  {
5     public static void main( String[] args )
6     {
7        Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
8        Integer[] integerElements = {  1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9
10       // Pilha de tipos brutos atribuídos à classe Stack da variável de tipos brutos
11       Stack rawTypeStack1 = new Stack( 5 );
12
13       // Stack< Double > atribuído à Stack da variável de tipos brutos
14       Stack rawTypeStack2 = new Stack< Double >( 5 );
15
16       // Pilha de tipos crus atribuídos à variável Stack< Integer >
17       Stack< Integer > integerStack = new Stack( 10 );
18
19       testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
20       testPop( "rawTypeStack1", rawTypeStack1 );
21       testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
22       testPop( "rawTypeStack2", rawTypeStack2 );
23       testPush( "integerStack", integerStack, integerElements );
24       testPop( "integerStack", integerStack );
25    } // fim de main
26
27    // método genérico insere elementos na pilha
28    public static < T > void testPush( String name, Stack< T > stack,
29       T[] elements )
```

```
30      {
31         System.out.printf( "\nPushing elements onto %s\n", name );
32
33         // insere elementos na Stack
34         for ( T element : elements )
35         {
36            System.out.printf( "%s ", element );
37            stack.push( element ); // insere o elemento na pilha
38         } // for final
39      } // fim do método testPush
40
41      // método genérico testPop remove elementos da pilha
42      public static < T > void testPop( String name, Stack< T > stack )
43      {
44         // remove elementos da pilha
45         try
46         {
47            System.out.printf( "\nPopping elements from %s\n", name );
48            T popValue; // armazena o elemento removido da pilha
49
50            // remove elementos da Stack
51            while ( true )
52            {
53               popValue = stack.pop(); // remove da pilha
54               System.out.printf( "%s ", popValue );
55            } // fim do while
56         } // fim do try
57         catch( EmptyStackException emptyStackException )
58         {
59            System.out.println();
60            emptyStackException.printStackTrace();
61         } // fim da captura de EmptyStackException
62      } // fim do método testPop
63   } // fim da classe RawTypeTest
```

```
Pushing elements onto rawTypeStack1
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack1
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:20)

Pushing elements onto rawTypeStack2
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack2
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:22)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:24)
```

Figura 21.11 | Programa de teste de tipos brutos.

A Figura 21.12 mostra as mensagens de alerta geradas pelo compilador quando o arquivo `RawTypeTest.java` (Figura 21.11) é compilado com a opção `-Xlint:unchecked`, que fornece informações adicionais sobre operações potencialmente perigosas no código que utiliza genéricos. O primeiro alerta é gerado para a linha 17, que atribuiu um tipo bruto de `Stack` a uma variável `Stack<Integer>` — o compilador não pode assegurar que todos os objetos na `Stack` serão objetos `Integer`. O próximo alerta ocorre na linha 19. O compilador determina o argumento de tipo do método `testPush` a partir do array `Double` passado como o terceiro argumento, porque o segundo argumento do método é uma variável `Stack` do tipo bruto. Nesse caso, `Double` é o argumento de tipo, portanto o compilador espera uma `Stack<Double>` como o segundo argumento. O alerta ocorre porque o compilador não pode assegurar que o tipo bruto `Stack` só contém `Double`s. O alerta na linha 21 ocorre pela mesma razão, embora a `Stack` real que `rawTypeStack2` referencia seja uma `Stack<Double>`. O compilador não pode garantir que a variável sempre irá se referir ao mesmo objeto `Stack`, assim ele deve utilizar o tipo declarado da variável para realizar todas as verificações de tipos. As linhas 20 e 22 geram alertas porque o método `testPop` espera como um argumento uma `Stack` à qual foi especificado um argumento de tipo. Entretanto, em cada chamada a `testPop`, passamos uma variável `Stack` de tipo bruto. Portanto, o compilador indica um alerta uma vez que não pode verificar os tipos utilizados no corpo do método.

```
RawTypeTest.java:17: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Integer>
      Stack< Integer > integerStack = new Stack( 10 );
                                      ^
RawTypeTest.java:19: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
                                 ^
RawTypeTest.java:19: warning: [unchecked] unchecked method invocation: <T>testPush(java.lang.
String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
               ^
RawTypeTest.java:20: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop( "rawTypeStack1", rawTypeStack1 );
                                ^
RawTypeTest.java:20: warning: [unchecked] unchecked method invocation: <T>testPop(java.lang.
String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop( "rawTypeStack1", rawTypeStack1 );
              ^
RawTypeTest.java:21: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<java.lang.Double>
      testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
                                 ^
RawTypeTest.java:21: warning: [unchecked] unchecked method invocation: <T>testPush(java.lang.
String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
      testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
               ^
RawTypeTest.java:22: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<T>
      testPop( "rawTypeStack2", rawTypeStack2 );
                                ^
RawTypeTest.java:22: warning: [unchecked] unchecked method invocation: <T>testPop(java.lang.
String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
      testPop( "rawTypeStack2", rawTypeStack2 );
              ^
9 warnings
```

Figura 21.12 | Mensagens de alerta do compilador.

21.8 Curingas em métodos que aceitam parâmetros de tipo

Nesta seção, apresentamos um conceito poderoso sobre genéricos conhecido como **curingas**. Para esse propósito, também apresentaremos uma nova estrutura de dados do pacote `java.util`. No Capítulo 20, discutimos o Java Collections Framework, que fornece várias estruturas e algoritmos de dados genéricos que manipulam os elementos dessas estruturas de dados. Talvez a mais simples dessas estruturas de dados seja a classe `ArrayList` — uma estrutura de dados dinamicamente redimensionável como de um array. Como parte dessa discussão, você aprenderá a criar uma `ArrayList`, adicionar elementos a ela e percorrer esses elementos utilizando uma instrução `for` aprimorada.

Antes de apresentarmos os curingas, vamos considerar um exemplo que nos ajuda a motivar a utilização deles. Suponha que você quer implementar um método genérico `sum` que soma os números em uma coleção, como uma `ArrayList`. Começaria inserindo os números na coleção. Como se sabe, classes genéricas só podem ser utilizadas com tipos de classe ou interface, portanto os números passariam por um autoboxing como objetos das classes empacotadoras de tipo. Por exemplo, qualquer valor `int` seria autoempacotado como um objeto `Integer` e qualquer valor `double` seria convertido por autoboxing em um objeto `Double`. Queremos ser capazes de somar todos os números na `ArrayList` independentemente dos seus tipos. Por essa razão, declararemos a `ArrayList` com o argumento de tipo `Number`, que é a superclasse de `Integer` e `Double`. Além disso, o método `sum` receberá um parâmetro do tipo `ArrayList<Number>` e somará seus elementos. A Figura 21.13 demonstra a somatória dos elementos de uma `ArrayList` de `Numbers`.

```
1  // Figura 21.13: TotalNumbers.java
2  // Somando os números em uma ArrayList<Number>.
3  import java.util.ArrayList;
4
5  public class TotalNumbers
6  {
7     public static void main( String[] args )
8     {
9        // cria, inicializa e gera saída de ArrayList de números contendo
10       // tanto Integers como Doubles e então exibe o total dos elementos
11       Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
12       ArrayList< Number > numberList = new ArrayList< Number >();
13
14       for ( Number element : numbers )
15          numberList.add( element ); // insere cada número na numberList
16
17       System.out.printf( "numberList contains: %s\n", numberList );
18       System.out.printf( "Total of the elements in numberList: %.1f\n",
19          sum( numberList ) );
20    } // fim de main
21
22    // calcula o total de elementos em ArrayList
23    public static double sum( ArrayList< Number > list )
24    {
25       double total = 0; // inicializa o total
26
27       // calcula a soma
28       for ( Number element : list )
29          total += element.doubleValue();
30
31       return total;
32    } // fim do método sum
33 } // fim da classe TotalNumbers
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Figura 21.13 | Somando os números em uma `ArrayList<Number>`.

A linha 11 declara e inicializa um array de `Numbers`. Como os inicializadores são valores primitivos, o Java autoempacota cada valor de tipo primitivo como um objeto do seu tipo empacotador correspondente. Os valores `int` `1` e `3` passam por um autoboxing como objetos `Integer`, e os valores `double` `2.4` e `4.1` passam por um autoboxing como objetos `Double`. A linha 12 declara e cria um objeto `ArrayList` que armazena `Numbers` e lhe atribui à variável `numberList`. Observe que não temos de especificar o tamanho do `ArrayList` porque ele aumentará automaticamente à medida que inserimos objetos.

As linhas 14–15 percorrem o array `numbers` e colocam cada elemento em `numberList`. A linha 17 gera saída do conteúdo do `ArrayList` como uma `String`. Essa instrução invoca implicitamente o método `toString` da `ArrayList` , que retorna uma `String` na forma `"[elements]"` , em que elementos é uma lista separada por vírgulas das representações da `String` dos elementos. As linhas 18–19 exibem a soma dos elementos que é retornada pela chamada ao método `sum`.

O método `sum` (linhas 23–32) recebe um `ArrayList` de `Numbers` e calcula o total do `Numbers` na coleção. O método utiliza valores `double` para executar os cálculos e retorna o resultado como um `double`. As linhas 28–29 utilizam a instrução `for` aprimorada, projetada para funcionar tanto em coleções como arrays do Collections Framework, para somar os elementos da `ArrayList`. A instrução `for` atribui cada `Number` do `ArrayList` à variável `element` e, então, utiliza o **método `Number doubleValue`** para obter o valor do tipo primitivo subjacente do `Number` como um valor `double`. O resultado é adicionado a `total`. Quando o loop termina, o método retorna o `total`.

Implementando o método `sum` com um argumento de tipo de curinga no seu parâmetro

Lembre-se de que o propósito do método `sum` na Figura 21.13 era somar qualquer tipo de `Numbers` armazenado em uma `ArrayList`. Criamos uma `ArrayList` de `Numbers` que continham tanto objetos `Integer` como `Double`. A saída da Figura 21.13 demonstra que o método `sum` funcionou adequadamente. Dado o fato de que esse método `sum` pode somar os elementos de uma `ArrayList` de `Numbers`, talvez você esperasse que o método também funcionasse para `ArrayLists` que contêm elementos de somente um tipo numérico, como `ArrayList<Integer>`. Assim, modificamos a classe `TotalNumbers` para criar uma `ArrayList` de `Integers` e passá-la para o método `sum`. Ao compilar o programa, o compilador emite a mensagem de erro a seguir:

```
sum(java.util.ArrayList<java.lang.Number>) in TotalNumbersErrors
cannot be applied to (java.util.ArrayList<java.lang.Integer>)
```

Embora `Number` seja a superclasse de `Integer`, o compilador não considera o tipo parametrizado `ArrayList<Number>` uma superclasse de `ArrayList<Integer>`. Se fosse, cada operação que podemos realizar em `ArrayList<Number>` também funcionaria em uma `ArrayList<Integer>`. Considere o fato de que você pode adicionar um objeto `Double` a uma `ArrayList<Number>` porque um `Double` *é um* `Number`, mas você não pode adicionar um objeto `Double` a um `ArrayList<Integer>` porque um `Double` *não é um* `Integer`. Portanto, o relacionamento de subtipos não se aplica.

Como é possível criar uma versão mais flexível do método `sum` que possa somar os elementos de qualquer `ArrayList` que contém elementos de qualquer subclasse de `Number`? É aí que os **argumentos de tipo curingas** são importantes. Os curingas permitem especificar parâmetros de método, valores de retorno, variáveis ou campos e assim por diante, que atuam como supertipos ou subtipos de tipos parametrizados. Na Figura 21.14, o parâmetro do método `sum` é declarado na linha 50 com o tipo:

```
ArrayList< ? extends Number >
```

Um argumento do tipo curinga é denotado pelo ponto de interrogação (**?**), que representa por si mesmo um "tipo desconhecido". Nesse caso, o curinga estende a classe `Number`, o que significa que o curinga tem um limite superior de `Number`. Portanto, o argumento de tipo desconhecido deve ser `Number` ou uma subclasse de `Number`. Com o tipo de parâmetro mostrado aqui, o método `sum` pode receber um argumento `ArrayList` que contém qualquer tipo de `Number`, como `ArrayList<Integer>` (linha 20), `ArrayList<Double>` (linha 33) ou `ArrayList<Number>` (linha 46).

```
1  // Figura 21.14: WildcardTest.java
2  // Programa de teste de curinga.
3  import java.util.ArrayList;
4
5  public class WildcardTest
6  {
7     public static void main( String[] args )
8     {
9        // cria, inicializa e gera saída de ArrayList de Integers, então
10       // exibe o total dos elementos
11       Integer[] integers = { 1, 2, 3, 4, 5 };
12       ArrayList< Integer > integerList = new ArrayList< Integer >();
13
14       // insere elementos na integerList
15       for ( Integer element : integers )
16          integerList.add( element );
17
18       System.out.printf( "integerList contains: %s\n", integerList );
19       System.out.printf( "Total of the elements in integerList: %.0f\n\n",
20          sum( integerList ) );
21
22       // cria, inicializa e gera saída do ArrayList de Doubles, então
23       // exibe o total dos elementos
24       Double[] doubles = { 1.1, 3.3, 5.5 };
```

```
25          ArrayList< Double > doubleList = new ArrayList< Double >();
26
27          // insere elementos na doubleList
28          for ( Double element : doubles )
29             doubleList.add( element );
30
31          System.out.printf( "doubleList contains: %s\n", doubleList );
32          System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
33             sum( doubleList ) );
34
35          // cria, inicializa e gera saída de ArrayList de números contendo
36          // tanto Integers como Doubles e então exibe o total dos elementos
37          Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38          ArrayList< Number > numberList = new ArrayList< Number >();
39
40          // insere elementos na numberList
41          for ( Number element : numbers )
42             numberList.add( element );
43
44          System.out.printf( "numberList contains: %s\n", numberList );
45          System.out.printf( "Total of the elements in numberList: %.1f\n",
46             sum( numberList ) );
47       } // fim de main
48
49       // soma os elementos; utilizando um curinga no parâmetro ArrayList
50       public static double sum( ArrayList< ? extends Number > list )
51       {
52          double total = 0; // inicializa o total
53
54          // calcula a soma
55          for ( Number element : list )
56             total += element.doubleValue();
57
58          return total;
59       } // fim do método sum
60    } // fim da classe WildcardTest
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Figura 21.14 | Programa de teste de curinga genérico.

As linhas 11–20 criam e inicializam um `ArrayList<Integer>`, geram a saída dos seus elementos e somam seus elementos chamando o método `sum` (linha 20). As linhas 24–33 realizam as mesmas operações para uma `ArrayList<Double>`. As linhas 37–46 realizam as mesmas operações para uma `ArrayList<Number>` que contém `Integers` e `Doubles`.

No método `sum` (linhas 50–59), embora os tipos de elemento do argumento de `ArrayList` não sejam conhecidos diretamente pelo método, eles são conhecidos como pelo menos do tipo `Number`, porque o curinga foi especificado com o limite superior `Number`. Por essa razão, a linha 56 é permitida, porque todos os objetos `Number` têm um método `doubleValue`.

Embora curingas forneçam flexibilidade ao passar tipos parametrizados para um método, eles também têm algumas desvantagens. Como o curinga (`?`) no cabeçalho do método (linha 50) não especifica um nome de parâmetro de tipo, você não pode utilizá-lo como um nome de tipo por todo o corpo do método (isto é, não pode substituir `Number` por `?` na linha 55). Você pode, porém, declarar o método `sum` desta maneira:

```
public static <T extends Number> double sum( ArrayList< T > list )
```

o que permite ao método receber uma `ArrayList` que contém elementos de qualquer subclasse `Number`. Você pode então utilizar o parâmetro de tipo `T` por todo o corpo do método.

Se o curinga for especificado sem um limite superior, somente os métodos do tipo `Object` podem ser invocados nos valores do tipo curinga. Além disso, métodos que utilizam curingas nos seus argumentos de tipo do parâmetro não podem ser utilizados para adicionar elementos a uma coleção referenciada pelo parâmetro.

Erro comum de programação 21.4

Utilizar um curinga na seção de parâmetro de tipo de um método ou utilizar um curinga como um tipo explícito de uma variável no corpo do método é um erro de sintaxe.

21.9 Genéricos e herança: notas

Os genéricos podem ser utilizados com a herança de várias maneiras:

- Uma classe genérica pode ser derivada de uma classe não genérica. Por exemplo, a classe `Object` é uma superclasse direta ou indireta de cada classe genérica.
- Uma classe genérica pode ser derivada de outra classe genérica. Por exemplo, a classe genérica `Stack` (no pacote `java.util`) é uma subclasse da classe genérica `Vector` (no pacote `java.util`). Discutimos essas classes no Capítulo 20.
- Uma classe não genérica pode ser derivada de uma classe genérica. Por exemplo, a classe não genérica `Properties` (no pacote `java.util`) é uma subclasse da classe genérica `Hashtable` (no pacote `java.util`). Também discutimos essas classes no Capítulo 20.
- Por fim, um método genérico em uma subclasse pode sobrescrever um método genérico em uma superclasse se os dois métodos tiverem a mesma assinatura.

21.10 Conclusão

Este capítulo introduziu genéricos. Você aprendeu a declarar métodos e classes genéricas. Discutimos como a compatibilidade com versões anteriores é alcançada via tipos brutos. Também aprendeu a utilizar curingas em um método genérico ou classe genérica. No Capítulo 22, você aprenderá a implementar suas próprias estruturas de dados dinâmicas personalizadas que podem aumentar ou reduzir de tamanho em tempo de execução. Em particular, implementará essas estruturas de dados utilizando as capacidades dos genéricos que aprendeu neste capítulo. Para obter informações adicionais sobre genéricos, visite nosso Java Resource Center em `www.deitel.com/Java/` e clique no tópico Java Generics sob o título Resource Center Contents.

Resumo

Seção 21.1 Introdução

- Os métodos genéricos permitem especificar, com uma única declaração de método, um conjunto de métodos relacionados.
- Classes e interfaces genéricas permitem especificar conjuntos de tipos relacionados.

Seção 21.2 Motivação para métodos genéricos

- Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados.
- Quando o compilador encontra uma chamada de método, ele tenta localizar uma declaração de método com o mesmo nome de método e parâmetros que sejam compatíveis com os tipos de argumentos na chamada de método.

Seção 21.3 Métodos genéricos: implementação e tradução em tempo de compilação

- Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, elas podem ser codificadas mais compacta e convenientemente com um método genérico. Uma única declaração de método genérico pode ser chamada com argumentos de diferentes tipos de dados. Com base nos tipos dos argumentos passados para um método genérico, o compilador trata cada chamada de método apropriadamente.
- Todas as declarações de métodos genéricos contêm uma seção de parâmetro de tipo delimitado por colchetes angulares (< e >) que precede o tipo de retorno do método.
- Uma seção do parâmetro de tipo contém um ou mais parâmetros de tipo separados por vírgulas.
- Um parâmetro de tipo é um identificador que especifica um nome de tipo genérico. Os parâmetros de tipo podem ser utilizados como o tipo de retorno, tipos de parâmetro e tipos de variáveis locais em uma declaração de método genérico e atuam como marcadores de lugar para os tipos dos argumentos passados para o método genérico, conhecidos como argumentos de tipo reais. Os parâmetros de tipo podem representar somente tipos por referência.
- Nomes de parâmetro de tipo utilizados por toda uma declaração de método devem corresponder àqueles declarados na seção do parâmetro de tipo. O nome de um parâmetro de tipo pode ser declarado somente uma vez na seção de parâmetro de tipo, mas pode aparecer mais de uma vez na lista de parâmetros do método.

- Quando o compilador encontra uma chamada de método, ele determina os tipos de argumento e tenta localizar um método com o mesmo nome e parâmetros que correspondem aos tipos de argumento. Se não houver um desses métodos, o compilador procura métodos com o mesmo nome, parâmetros compatíveis e métodos genéricos correspondentes.
- Objetos de uma classe que implementa a interface genérica `Comparable` podem ser comparados com o método `compareTo`, que retorna `0` se os objetos forem iguais, um número inteiro negativo se o primeiro objeto for menor que o segundo ou um número inteiro positivo se o primeiro objeto for maior do que o segundo.
- Todas as classes empacotadoras de tipos para tipos primitivos implementam a interface `Comparable`.
- Objetos `Comparable` podem ser utilizados com os métodos de pesquisa e classificação da classe `Collections`.
- Quando um método genérico é compilado, o compilador remove a seção do parâmetro de tipo e substitui os parâmetros de tipo por tipos reais. Esse processo é conhecido como erasure. Por padrão cada parâmetro de tipo é substituído pelo seu limite superior, que é `Object` a menos que especificado de outro modo.

Seção 21.4 Questões adicionais da tradução em tempo de compilação: métodos que utilizam um parâmetro de tipo como o tipo de retorno

- Quando a erasure é executada em um método que retorna uma variável de tipo, coerções explícitas são inseridas na frente de cada chamada de método para assegurar que o valor retornado tem o tipo esperado pelo chamador.

Seção 21.5 Sobrecarregando métodos genéricos

- Um método genérico pode ser sobrecarregado com outros métodos genéricos ou com métodos não genéricos.

Seção 21.6 Classes genéricas

- As classes genéricas fornecem um meio de descrever uma classe de uma maneira independente de tipo. Podemos então instanciar objetos específicos de tipo da classe genérica.
- Uma declaração de classe genérica se parece com a declaração de uma classe não genérica, exceto que o nome de classe é seguido por uma seção de parâmetro de tipo. A seção do parâmetro de tipo de uma classe genérica pode ter um ou mais parâmetros de tipo separados por vírgulas.
- Quando uma classe genérica é compilada, o compilador realiza a erasure nos parâmetros de tipo da classe e os substitui pelos seus limites superiores.
- Os parâmetros de tipo não podem ser utilizados nas declarações de uma classe `static`.
- Ao instanciar um objeto de uma classe genérica, os tipos especificados dentro de colchetes angulares depois do nome de classe são conhecidos como argumentos de tipo. O compilador utiliza-os para substituir os parâmetros de tipo a fim de que ele possa executar a verificação dos tipos e inserir operações de coerção conforme necessário.

Seção 21.7 Tipos brutos

- É possível instanciar uma classe genérica sem especificar um argumento de tipo. Nesse caso, diz-se que o novo objeto da classe tem um tipo bruto — o compilador utiliza implicitamente o tipo `Object` (ou o limite superior do parâmetro de tipo) por toda a classe genérica para cada argumento de tipo.

Seção 21.8 Curingas em métodos que aceitam parâmetros de tipo

- A classe `Number` é a superclasse tanto de `Integer` como `Double`.
- O método `Number` `doubleValue` obtém o valor do tipo primitivo subjacente de `Number` como um valor de `double`.
- Os argumentos do tipo curinga permitem especificar parâmetros de método, valores de retorno, variáveis e assim por diante, que atuam como supertipos dos tipos parametrizados. Um argumento do tipo curinga é denotado pelo ponto de interrogação (`?`), que representa um "tipo desconhecido". Um curinga também pode ter um limite superior.
- Como um curinga (`?`) não é um nome do parâmetro de tipo, você não pode utilizá-lo como um nome de tipo por todo o corpo de um método.
- Se um curinga for especificado sem um limite superior, somente os métodos do tipo `Object` podem ser invocados nos valores do tipo de curinga.
- Métodos que utilizam curingas como argumentos de tipo não podem ser utilizados para adicionar elementos a uma coleção referenciada pelo parâmetro.

Seção 21.9 Genéricos e herança: Notas

- Uma classe genérica pode ser derivada de uma classe não genérica. Por exemplo, `Object`- é uma superclasse direta ou indireta de cada classe genérica.
- Uma classe genérica pode ser derivada de outra classe genérica.
- Uma classe não genérica pode ser derivada de uma classe genérica.
- Um método genérico em uma subclasse pode sobrescrever um método genérico em uma superclasse se os dois métodos tiverem as mesmas assinaturas.

Terminologia

? (argumento de tipo curinga), 688
argumento de tipo, 679
argumento de tipo curinga, 688
argumentos reais de tipo, 674
classe parametrizada, 678
classes genéricas, 671
colchetes angulares (< e >), 674
`Comparable<T>`, interface, 675
`compareTo`, método de `Comparable<T>`, 675
curinga em um parâmetro de tipo genérico, 687
`doubleValue`, método de `Number`, 688
erasure, 675
genéricos, 671
interface genérica, 675
limite superior de um parâmetro de tipo, 676
método genérico, 671
parâmetro formal de tipo, 674
seção de parâmetro de tipo, 674
segurança de tipo em tempo de compilação, 671
sinais de maior e menor (< e >), 674
tipo, parâmetro, 674
tipo, variável, 674
tipo bruto, 684
tipo parametrizado, 678

Exercícios de autorrevisão

21.1 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Um método genérico não pode ter o mesmo nome de método de um método não genérico.

b) Todas as declarações de métodos genéricos têm uma seção de parâmetro de tipo que precede imediatamente o nome de método.

c) Um método genérico pode ser sobrecarregado por outro método genérico com o mesmo nome do método, mas diferentes parâmetros de método.

d) Um parâmetro de tipo pode ser declarado somente uma vez na seção de parâmetro de tipo, mas pode aparecer mais de uma vez na lista de parâmetros do método.

e) Os nomes dos parâmetros de tipo entre diferentes métodos genéricos devem ser únicos.

f) O escopo de um parâmetro de tipo da classe genérico é a classe inteira, exceto seus membros `static`.

21.2 Preencha as lacunas em cada uma das sentenças:

a) _____ e _____ permitem especificar, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados, respectivamente.

b) Uma seção de parâmetro de tipo é delimitada por _____.

c) O _____ de um método genérico pode ser utilizado para especificar os tipos de argumento do método, especificar o tipo de retorno do método e declarar variáveis dentro do método.

d) A instrução "`Stack objectStack = new Stack();`" indica que `objectStack` armazena _____.

e) Na declaração de uma classe genérica, o nome da classe é seguido por um(a) _____.

f) A sintaxe _____ especifica que o limite superior de um curinga é o tipo `T`.

Respostas dos exercícios de autorrevisão

21.1 a) Falsa. Métodos genéricos e não genéricos podem ter o mesmo nome de método. Um método genérico pode sobrecarregar outro método genérico com o mesmo nome do método, mas diferentes parâmetros de método. Um método genérico também pode ser sobrecarregado fornecendo métodos não genéricos com o mesmo nome de método e número de argumentos. b) Falsa. Todas as declarações de métodos genéricos têm uma seção de parâmetro de tipo que precede imediatamente o tipo de retorno do método. c) Verdadeira. d) Verdadeira. e) Falsa. Os nomes de parâmetro de tipo entre diferentes métodos genéricos não precisam ser únicos. f) Verdadeira.

21.2 a) Métodos genéricos, classes genéricas. b) colchetes angulares (< e >). c) parâmetros de tipo. d) um tipo bruto. e) seção de parâmetro de tipo. f) `? extends T`.

Exercícios

21.3 Explique o uso da seguinte notação em um programa Java:

```
public class Array< T > { }
```

21.4 Escreva um método genérico `selectionSort` com base no programa de classificação das Figuras 19.6–19.7. Escreva um programa de teste que insere, classifica e gera saída de um array `Integer` e de um array `Float`. [*Dica*: Utilize `<T extends Comparable<T>>` na seção do parâmetro de tipo do método `selectionSort`, de modo que possa utilizar o método `compareTo` para comparar os objetos do tipo que `T` representa.]

21.5 Sobrecarregue o método genérico `printArray` da Figura 21.3 de modo que ele aceite dois argumentos adicionais de inteiros, `lowSubscript` e `highSubscript`. Uma chamada a esse método imprime somente a parte especificada do array. Valide `lowSubscript` e `highSubscript`. Se um estiver fora do intervalo, o método `printArray` sobrecarregado deverá lançar uma `InvalidSubscriptException`; caso contrário, `printArray` deve retornar o número de elementos impresso. Modifique então `main` para praticar as duas versões de `printArray` nos arrays `integerArray`, `doubleArray` e `characterArray`. Teste todas as capacidades das duas versões de `printArray`.

21.6 Sobrecarregue o método genérico `printArray` da Figura 21.3 com uma versão não genérica que imprime especificamente um array de `Strings` em formato tabular elegante, como mostrado na saída do exemplo a seguir:

```
Array stringArray contains:
one     two       three     four
five    six       seven     eight
```

21.7 Escreva uma versão genérica simples do método `isEqualTo` que compara seus dois argumentos com o método `equals` e retorna `true` se forem iguais e `false` caso contrário. Utilize esse método genérico em um programa que chama `isEqualTo` com uma variedade de tipos predefinidos, como `Object` ou `Integer`. Qual resultado você obtém ao tentar executar esse programa?

21.8 Escreva uma classe genérica `Pair` que tem dois parâmetros de tipo — `F` e `S` — cada um representando o tipo do primeiro e segundo elemento do par, respectivamente. Adicione os métodos *get* e *set* ao primeiro e ao segundo elemento do par. [*Dica*: O cabeçalho de classe deve ser `public class Pair<F, S>`.]

21.9 Como métodos genéricos podem ser sobrecarregados?

21.10 O compilador realiza um processo de correspondência para determinar qual método chamar quando um método é invocado. Sob quais circunstâncias uma tentativa de fazer uma correspondência resulta em um erro em tempo de compilação?

21.11 Explique por que um programa Java utilizaria a instrução

```
ArrayList< Employee > workerList = new ArrayList< Employee >();
```

Much that I bound, I could not free;
Much that I freed returned to me.
[Muito do que eu prendi, não pude libertar;
Muito que eu libertei voltou para mim.]
— **Lee Wilson Dodd**

'Will you walk a little faster?' said a whiting to a snail,
'There's a porpoise close behind us, and he's treading on my tail.'
['Não dá pra ir mais rápido?' disse a enchova para o caracol
'Tem um delfim atrás de mim, e ele está me empurrando.']
— **Lewis Carroll**

Há sempre espaço no ponto mais alto.
— **Daniel Webster**

Prossiga — continue andando.
— **Thomas Morton**

Virarei uma nova folha.
— **Miguel de Cervantes**

22

Estruturas de dados genéricas personalizadas

Objetivos

Neste capítulo, você aprenderá:

- A formar estruturas de dados encadeadas utilizando referências, classes autorreferenciais, recursão e genéricos.
- A criar e manipular estruturas de dados dinâmicas, como listas encadeadas, filas, pilhas e árvores binárias.
- Vários aplicativos importantes de estruturas de dados encadeadas.
- Como criar estruturas de dados reutilizáveis com classes, herança e composição.

Sumário

22.1 Introdução
22.2 Classes autorreferenciais
22.3 Alocação dinâmica de memória
22.4 Listas vinculadas
22.5 Pilhas
22.6 Filas
22.7 Árvores
22.8 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Seção especial: construindo seu próprio compilador

22.1 Introdução

Este capítulo demonstra como construir **estruturas de dados dinâmicas** que crescem e encolhem em tempo de execução. As **listas encadeadas** são coleções de itens de dados "vinculados em uma cadeia"; as inserções e exclusões podem ser feitas em qualquer lugar de uma lista encadeada. As **pilhas** são importantes em compiladores e sistemas operacionais; as inserções e as exclusões são feitas somente no fim de uma pilha — sua **parte superior**. As **filas** representam filas de espera; as inserções são feitas na parte de atrás (também referida como **cauda**) de uma fila e as exclusões são feitas na parte da frente da fila (também referida como **cabeça**). As **árvores binárias** facilitam a pesquisa e classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos, a compilação de expressões em linguagem de máquina e muitas outras aplicações interessantes.

Discutimos cada um desses principais tipos de estrutura de dados e implementamos programas que os criam e manipulam. Utilizamos classes, herança e composição para criá-las e empacotá-las a fim de implementar as capacidades de reutilização e manutenção. Em geral, você utilizaria uma das classes de coleção predefinidas que discutimos no Capítulo 20. Contudo, as técnicas que apresentamos aqui podem ser utilizadas sempre que você precisar construir suas próprias coleções personalizadas.

Por uma questão de simplicidade, os exemplos deste capítulo manipulam valores primitivos. Contudo, as implementações de estrutura de dados neste capítulo podem armazenar objetos da maioria dos tipos. (O exemplo de árvore binária requer objetos que implementem a interface `Comparable<T>`.)

Se sentir-se confiante, pode tentar o projeto principal descrito na seção especial intitulada Building Your Own Compiler (Construindo seu próprio compilador), que postamos on-line em `www.deitel.com/books/jhtp8/`. Você vem utilizando um compilador Java para traduzir seus programas Java em bytecodes a fim de poder executar esses programas em seu computador. Nesse projeto, realmente construirá seu próprio compilador. Essa seção apresentará instruções escritas em uma linguagem de alto nível simples, mas poderosa e semelhante às primeiras versões da popular linguagem Basic e as converterá em instruções SML (Simpletron Machine Language) — SML é a linguagem que você aprendeu na seção especial do Capítulo 7, Construindo seu próprio computador. Seu programa Simpletron Simulator então executará o programa SML produzido por seu compilador! A implementação desse projeto utilizando uma abordagem orientada a objetos lhe fornecerá uma excelente oportunidade para praticar grande parte do que aprendeu neste livro. A seção especial o orienta cuidadosamente ao longo das especificações da linguagem de alto nível e descreve os algoritmos que você precisará para converter cada instrução de linguagem de alto nível em instruções de linguagem de máquina. Se gosta de desafios, pode tentar os muitos aprimoramentos no compilador e no Simpletron Simulator sugeridos nos exercícios.

22.2 Classes autorreferenciais

Uma **classe autorreferencial** contém uma variável de instância que referencia outro objeto do mesmo tipo de classe. Por exemplo, a declaração de classe genérica

```
class Node< T >
{
   private T data;
   private Node< T > nextNode; // referência ao próximo nó vinculado
   public Node( T data ) { /* corpo do construtor */ }
   public void setData( T data ) { /* corpo do método */ }
   public T getData() { /* corpo do método */ }
   public void setNext( Node< T > next ) { /* corpo do método */ }
   public Node< T > getNext() { /* corpo do método */ }
} // fim da classe Node< T >
```

declara a classe `Node`, que tem duas variáveis de instância `private` — a variável `data` (do tipo genérico `T`) e `Node<T> nextNode`. A variável `nextNode` referencia um objeto `Node<T>`, um objeto da mesma classe que está sendo declarada aqui, daí o termo "classe autorreferencial". O campo `nextNode` é um **link** — ele "vincula" um objeto do tipo `Node<T>` a outro objeto do mesmo tipo. O tipo `Node<T>` também tem cinco métodos: um construtor que recebe um valor para inicializar `data`, um método `setData` para configurar o valor de `data`, um método `getData` para retornar o valor de `data`, um método `setNext` para configurar o valor de `nextNode` e um método `getNext` para retornar uma referência ao próximo nó.

Os programas podem vincular objetos autorreferenciais para formar estruturas de dados tão úteis quanto listas, filas, pilhas e árvores. A Figura 22.1 ilustra dois objetos autorreferenciais vinculados entre si para formar uma lista. Uma barra invertida — representando uma referência `null` — é colocada no membro de link do segundo objeto autorreferencial para indicar que o link não referencia outro objeto. Note que a barra invertida é ilustrativa; ela não corresponde ao caractere de barra invertida no Java. Utilizamos a referência `null` para indicar o fim de uma estrutura de dados.

Figura 22.1 | Objetos de classe autorreferencial vinculados entre si.

22.3 Alocação dinâmica de memória

Criar e manter estruturas de dados dinâmicas requer **alocação dinâmica de memória** — permissão para que um programa obtenha mais espaço de memória em tempo de execução para armazenar novos nós e liberar espaço não mais necessário. Lembre-se de que os programas Java não explicitamente liberam memória alocada dinamicamente. Em vez disso, o Java realiza coleta de lixo automática de objetos que não são mais referenciados em um programa.

O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de espaço em disco disponível em um sistema de memória virtual. Frequentemente, os limites são muito menores, porque a memória disponível do computador deve ser compartilhada entre muitos aplicativos.

A criação da declaração e expressão de instância de classe

```
// 10 são os dados de nodeToAdd
Node< Integer > nodeToAdd = new Node< Integer >( 10 );
```

aloca a memória para armazenar um objeto `Node<Integer>` e retornar uma referência ao objeto, que é atribuído a `nodeToAdd`. Se a memória insuficiente estiver disponível, a expressão lança um `OutOfMemoryError`.

As seções a seguir discutem que listas, pilhas, filas e árvores utilizam alocação dinâmica de memória e classes autorreferenciais para criar estruturas de dados dinâmicas.

22.4 Listas vinculadas

Uma lista vinculada é uma coleção linear (isto é, uma sequência) de objetos autorreferenciais de classe, chamados **nós**, conectados por links de referência — daí o termo lista "vinculada". Em geral, um programa acessa uma lista encadeada via uma referência ao primeiro nó. O programa acessa cada nó subsequente via a referência de link armazenado no nó anterior. Por convenção, a referência de vínculo no último nó da lista é configurada como `null`. Os dados são armazenados em uma lista encadeada dinamicamente — o programa cria cada nó conforme necessário. As pilhas e filas são estruturas de dados também lineares e, como veremos, são versões limitadas de listas vinculadas. As árvores são estruturas de dados não lineares.

As listas de dados podem ser armazenadas em arrays, mas as listas vinculadas fornecem várias vantagens. Uma lista vinculada é apropriada quando o número de elementos de dados a ser representado na estrutura de dados é imprevisível. As listas encadeadas são dinâmicas, portanto, o comprimento de uma lista pode aumentar ou diminuir de acordo com a necessidade, enquanto o tamanho de um array Java "convencional" não pode ser alterado — ele é fixado quando o programa cria o array. Os arrays "convencionais" podem tornar-se cheios. As listas vinculadas tornam-se cheias apenas quando o sistema tem memória insuficiente para satisfazer solicitações de alocação de armazenamento dinâmico. O pacote `java.util` contém a classe `LinkedList` (discutida no Capítulo 20) para implementar e manipular listas encadeadas que crescem e encolhem durante a execução de programa.

Dica de desempenho 22.1

Um array pode ser declarado para conter mais elementos do que o número de itens esperado, mas isso desperdiça memória. Nessas situações, as listas encadeadas fornecem melhor uso de memória permitindo ao programa adaptar-se às necessidades de armazenamento em tempo de execução.

Dica de desempenho 22.2

A inserção em uma lista encadeada é rápida — somente duas referências precisam ser modificadas (depois de localizar ponto de inserção). Todos os objetos existentes de nó permanecem em suas localizações atuais na memória.

As listas vinculadas podem ser mantidas em ordem de classificação simplesmente inserindo cada novo elemento no ponto adequado na lista. (Sem dúvida, realmente, leva tempo para localizar o ponto de inserção adequado.) Os elementos existentes da lista não precisam ser movidos.

Dica de desempenho 22.3

A inserção e a exclusão em um array classificado podem consumir muito tempo — todos os elementos que se seguem ao elemento inserido ou excluído devem ser deslocados apropriadamente.

Listas encadeadas individualmente

Nós de lista vinculada normalmente não são armazenados contiguamente na memória. Em vez disso, são logicamente contíguos. A Figura 22.2 ilustra uma lista vinculada com vários nós. Esse diagrama apresenta uma **lista encadeada individualmente** — cada nó contém uma referência ao próximo nó da lista. Em geral, as listas encadeadas são implementadas como listas duplamente encadeadas — cada nó contém uma referência ao próximo nó na lista e uma referência ao anterior. A classe `LinkedList` do Java é a implementação de uma lista duplamente encadeada.

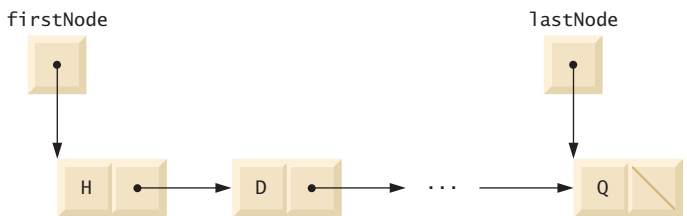


Figura 22.2 | Representação gráfica de lista encadeada.

Dica de desempenho 22.4

Normalmente, os elementos de um array são contíguos na memória. Isso permite acesso imediato a qualquer elemento do array, porque seu endereço pode ser calculado diretamente como seu deslocamento a partir do início do array. As listas encadeadas não têm recursos para tal acesso imediato — um elemento só pode ser acessado percorrendo a lista da parte da frente (ou da parte da trás em uma lista duplamente encadeada).

Implementando uma classe `List` genérica

O programa das figuras 22.3–22.5 utiliza um objeto de nossa classe genérica `List` para manipular uma lista de objetos variados. O programa consiste em quatro classes — `ListNode` (Figura 22.3, linhas 6–37), `List` (Figura 22.3, linhas 40–147), `EmptyListException` (Figura 22.4) e `ListTest` (Figura 22.5). As classes `List`, `ListNode` e `EmptyListException` encontram-se no pacote `com.deitel.ch22`, portanto, podem ser reutilizadas por todo este capítulo. Encapsulada em cada objeto `List` está uma lista vinculada de objetos `ListNode`. [*Nota*: Muitas das classes neste capítulo são declaradas no pacote `com.deitel.ch22`. Toda classe assim deve ser compilada com a opção de linha de comando `-d` para `javac`. Ao compilar as classes que não estão nesse pacote e ao executar os programas, não deixe de utilizar a opção `-classpath` com `javac` e `java`, respectivamente.]

```
1  // Figura 22.3: List.java
2  // Declarações da classe ListNode e List.
3  package com.deitel.ch22;
4
5  // classe para representar um nó em uma lista
6  class ListNode< T >
7  {
8     // membros de acesso de pacote; List pode acessar esses diretamente
9     T data; // dados para esse nó
10    ListNode< T > nextNode; // referência para o próximo nó na lista
11
```

```
12       // construtor cria uma ListNode que referencia o objeto
13       ListNode( T object )
14       {
15          this( object, null );
16       } // fim do construtor de um argumento ListNode
17
18       // construtor cria ListNode que referencia o objeto
19       // especificado e a próxima ListNode
20       ListNode( T object, ListNode< T > node )
21       {
22          data = object;
23          nextNode = node;
24       } // fim do construtor de dois argumentos ListNode
25
26       // retorna referência aos dados no nó
27       T getData()
28       {
29          return data; // retorna o item nesse nó
30       } // fim do método getData
31
32       // retorna referência ao próximo nó na lista
33       ListNode< T > getNext()
34       {
35          return nextNode; // obtém próximo nó
36       } // fim do método getNext
37    } // fim da classe ListNode< T >
38
39    // definição da classe List
40    public class List< T >
41    {
42       private ListNode< T > firstNode;
43       private ListNode< T > lastNode;
44       private String name; // string como "lista" usada na impressão
45
46       // construtor cria List vazia com "list" como o nome
47       public List()
48       {
49          this( "list" );
50       } // fim do construtor sem argumentos List
51
52       // construtor cria uma List vazia com um nome
53       public List( String listName )
54       {
55          name = listName;
56          firstNode = lastNode = null;
57       } // fim do construtor de um argumento List
58
59       // insere o item na frente de List
60       public void insertAtFront( T insertItem )
61       {
62          if ( isEmpty() ) // firstNode e lastNode referenciam o mesmo objeto
63             firstNode = lastNode = new ListNode< T >( insertItem );
64          else // firstNode referenciam o novo nó
65             firstNode = new ListNode< T >( insertItem, firstNode );
66       } // fim do método insertAtFront
67
68       // insere o item no fim de List
69       public void insertAtBack( T insertItem )
70       {
71          if ( isEmpty() ) // firstNode e lastNode referenciam o mesmo objeto
72             firstNode = lastNode = new ListNode< T >( insertItem );
73          else // nextNode do lastNode referencia o novo nó
74             lastNode = lastNode.nextNode = new ListNode< T >( insertItem );
75       } // fim do método insertAtBack
76
77       // remove o primeiro nó de List
78       public T removeFromFront() throws EmptyListException
79       {
80          if ( isEmpty() ) // lança exceção se List estiver vazia
```

```
81             throw new EmptyListException( name );
82
83          T removedItem = firstNode.data; // recupera dados sendo removidos
84
85          // atualiza referências firstNode e lastNode
86          if ( firstNode == lastNode )
87             firstNode = lastNode = null;
88          else
89             firstNode = firstNode.nextNode;
90
91          return removedItem; // retorna dados de nó removidos
92       } // fim do método removeFromFront
93
94       // remove o último nó de List
95       public T removeFromBack() throws EmptyListException
96       {
97          if ( isEmpty() ) // lança exceção se List estiver vazia
98             throw new EmptyListException( name );
99
100         T removedItem = lastNode.data; // recupera dados sendo removidos
101
102         // atualiza referências firstNode e lastNode
103         if ( firstNode == lastNode )
104            firstNode = lastNode = null;
105         else // localiza o novo último nó
106         {
107            ListNode< T > current = firstNode;
108
109            // faz loop enquanto nó atual não referencia lastNode
110            while ( current.nextNode != lastNode )
111               current = current.nextNode;
112
113            lastNode = current; // atual é novo lastNode
114            current.nextNode = null;
115         } // fim de else
116
117         return removedItem; // retorna dados de nó removidos
118      } // fim do método removeFromBack
119
120      // determina se a lista estiver vazia
121      public boolean isEmpty()
122      {
123         return firstNode == null; // retorna true se a lista estiver vazia
124      } // fim do método isEmpty
125
126      // gera saída do conteúdo da lista
127      public void print()
128      {
129         if ( isEmpty() )
130         {
131            System.out.printf( "Empty %s\n", name );
132            return;
133         } // fim do if
134
135         System.out.printf( "The %s is: ", name );
136         ListNode< T > current = firstNode;
137
138         // enquanto não estiver no fim de lista, gera saída dos dados do nó atual
139         while ( current != null )
140         {
141            System.out.printf( "%s ", current.data );
142            current = current.nextNode;
143         } // fim do while
144
145         System.out.println( "\n" );
146      } // fim do método print
147   } // fim da classe List< T >
```

Figura 22.3 | Declarações de classe `ListNode` e `List`.

```
1  // Figura 22.4: EmptyListException.java
2  // declaração da classe EmptyListException.
3  package com.deitel.ch22;
4
5  public class EmptyListException extends RuntimeException
6  {
7     // construtor sem argumento
8     public EmptyListException()
9     {
10       this( "List" ); // chama outro construtor de EmptyListException
11    } // fim do construtor sem argumento EmptyListException
12
13    // construtor de um argumento
14    public EmptyListException( String name )
15    {
16       super( name + " is empty" ); // chama construtor de superclasse
17    } // fim do construtor de um argumento EmptyListException
18 } // fim da classe EmptyListException
```

Figura 22.4 | Declaração da classe `EmptyListException`.

```
1  // Figura 22.5: ListTest.java
2  // Classe ListTest para demonstrar capacidades de List.
3  import com.deitel.ch22.List;
4  import com.deitel.ch22.EmptyListException;
5
6  public class ListTest
7  {
8     public static void main( String[] args )
9     {
10       List< Integer > list = new List< Integer >(); // cria uma List
11
12       // insere inteiros na lista
13       list.insertAtFront( -1 );
14       list.print();
15       list.insertAtFront( 0 );
16       list.print();
17       list.insertAtBack( 1 );
18       list.print();
19       list.insertAtBack( 5 );
20       list.print();
21
22       // remove objetos da lista; imprime depois de cada remoção
23       try
24       {
25          int removedItem = list.removeFromFront();
26          System.out.printf( "\n%d removed\n", removedItem );
27          list.print();
28
29          removedItem = list.removeFromFront();
30          System.out.printf( "\n%d removed\n", removedItem );
31          list.print();
32
33          removedItem = list.removeFromBack();
34          System.out.printf( "\n%d removed\n", removedItem );
35          list.print();
36
37          removedItem = list.removeFromBack();
38          System.out.printf( "\n%d removed\n", removedItem );
39          list.print();
40       } // fim do try
41       catch ( EmptyListException emptyListException )
42       {
43          emptyListException.printStackTrace();
44       } // fim do catch
45    } // fim de main
46 } // fim da classe ListTest
```

```
The list is: -1
The list is: 0 -1
The list is: 0 -1 1
The list is: 0 -1 1 5

0 removed
The list is: -1 1 5

-1 removed
The list is: 1 5

5 removed
The list is: 1

1 removed
Empty list
```

Figura 22.5 | Classe `ListTest` para demonstrar capacidades `List`.

Classes genéricas `ListNode` e `List`

A classe genérica `ListNode` (Figura 22.3, linhas 6–37) declara campos de acesso de pacote `data` e `nextNode`. O campo `data` é uma referência do tipo `T`, portanto, seu tipo será determinado quando o código de cliente criar o objeto `List` correspondente. A variável `nextNode` armazena uma referência ao próximo objeto `ListNode` na lista encadeada (ou `null` se o nó for o último na lista).

As linhas 42–43 da classe `List` (Figura 22.3, linhas 40–47) declaram referências ao primeiro e último `ListNodes` em uma `List` (`firstNode` e `lastNode`, respectivamente). Os construtores (linhas 47–50 e 53–57) inicializam ambas as referências como `null`. Os métodos mais importantes da classe `List` são `insertAtFront` (linhas 60–66), `insertAtBack` (linhas 69–75), `removeFromFront` (linhas 78–92) e `removeFromBack` (linhas 95–118). O método `isEmpty` (linhas 121–124) é um *método predicado* que determina se a lista está vazia (isto é, a referência ao primeiro nó da lista é `null`). Os métodos predicados em geral testam uma condição e não modificam o objeto em que eles são chamados. Se a lista estiver vazia, o método `isEmpty` retorna `true`; caso contrário, retorna `false`. O método `print` (linhas 127–146) exibe o conteúdo da lista. Discutimos os métodos da classe `List` mais detalhadamente depois de discutirmos a classe `ListTest`.

Classe `ListTest`

O método `main` da classe `ListTest` (Figura 22.5) cria um objeto `List<Integer>`, (linha 10), insere objetos no início da lista utilizando o método `insertAtFront`, insere objetos no fim da lista utilizando o método `insertAtBack` , exclui objetos na frente da lista utilizando o método `removeFromFront` e exclui objetos do fim da lista utilizando o método `removeFromBack`. Depois que cada operação de inserção e remoção, `ListTest` chama o método `List` `print` para exibir o conteúdo da lista atual. Se uma tentativa de remover um item de uma lista vazia for feita, uma `EmptyListException` (Figura 22.4) é lançada, então as chamadas de método para `removeFromFront` e `removeFromBack` são colocadas em um bloco `try` que é seguido por um handler de exceção apropriado. Observe nas linhas 13, 15, 17 e 19 que o aplicativo passa os valores primitivos literais `int` para os métodos `insertAtFront` e `insertAtBack`. Cada um desses métodos foi declarado com um parâmetro do tipo genérico `T` (Figura 22.3, linhas 60 e 69). Como esse exemplo manipula um `List<Integer>`, o tipo `T` representa a classe de empacotador do tipo `Integer`. Nesse caso, a JVM converte (autoboxing) cada valor literal em um objeto `Integer` e esse objeto é realmente inserido na lista.

Método `List` `insertAtFront`

Agora discutimos cada método de classe `List` (Figura 22.3) em detalhes e fornecemos diagramas que mostram as manipulações de referência realizadas pelos métodos `insertAtFront`, `insertAtBack`, `removeFromFront` e `removeFromBack`. O método `insertAtFront` (linhas 60–66 da Figura 22.3) coloca um novo nó na frente da lista. Os passos são:

1. Chamar `isEmpty` para determinar se a lista está vazia (linha 62).
2. Se a lista for vazia, atribua a `firstNode` e `lastNode` o novo `ListNode` inicializado com `insertItem` (linha 63). (Lembre-se de que os operadores de atribuição avaliam da direita para a esquerda.) O construtor `ListNode` nas linhas 13–16 chama o construtor `ListNode` nas linhas 20–24 para configurar a variável de instância `data` para referenciar o `insertItem` passado como um argumento e para configurar a referência `nextNode` como `null`, porque esse é o primeiro e último nó na lista.
3. Se a lista não estiver vazia, o novo nó é "vinculado" na lista configurando `firstNode` como um novo objeto `ListNode` e inicializando esse objeto com `insertItem` e `firstNode` (linha 65). Quando o construtor `ListNode` (linhas 20–24) executar, ele configura a variável de instância `data` para referenciar o `insertItem` passado como um argumento e realiza a inserção configurando a referência `nextNode` do novo nó como o `ListNode` passado como um argumento, que era anteriormente o primeiro nó.

Na Figura 22.6, a parte (a) mostra uma lista e um novo nó durante a operação `insertAtFront` e antes de o programa vincular o novo nó na lista. As setas pontilhadas na parte (b) ilustram o *Passo 3* da operação `insertAtFront` que permite que o nó que contém `12` torne-se o novo primeiro nó na lista.

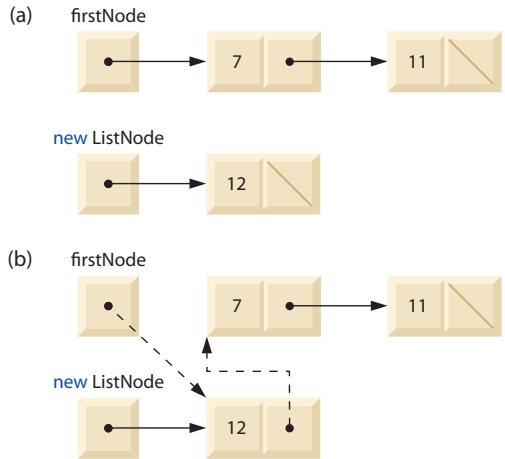


Figura 22.6 | Representação gráfica da operação `insertAtFront`.

Método `List` `insertAtBack`

O método `insertAtBack` (linhas 69–75 da Figura 22.3) coloca um novo nó na parte de trás da lista. Os passos são:

1. Chamar `isEmpty` para determinar se a lista está vazia (linha 71).
2. Se a lista for vazia, atribua a `firstNode` e `lastNode` o novo `ListNode` inicializado com `insertItem` (linha 72). O construtor `ListNode` nas linhas 13–16 chama o construtor nas linhas 20–24 para configurar a variável de instância `data` a fim de referenciar o `insertItem` passado como um argumento e configurar a referência `nextNode` como `null`.
3. Se a lista não estiver vazia, a linha 74 vincula o novo nó na lista atribuindo a `lastNode` e `lastNode.nextNode` a referência ao novo `ListNode` que foi inicializado com `insertItem`. O construtor do `ListNode` (linhas 13–16), configura a variável de instância `data` para referenciar o `insertItem` passado como um argumento e configura a referência `nextNode` como `null`, pois esse é o último nó na lista.

Na Figura 22.7, a parte (a) mostra uma lista e um novo nó durante a operação `insertAtBack` e antes de vincular o novo nó à lista. As setas pontilhadas na parte (b) ilustram o *Passo 3* do método `insertAtBack`, que adiciona o novo nó ao fim de uma lista que não estiver vazia.

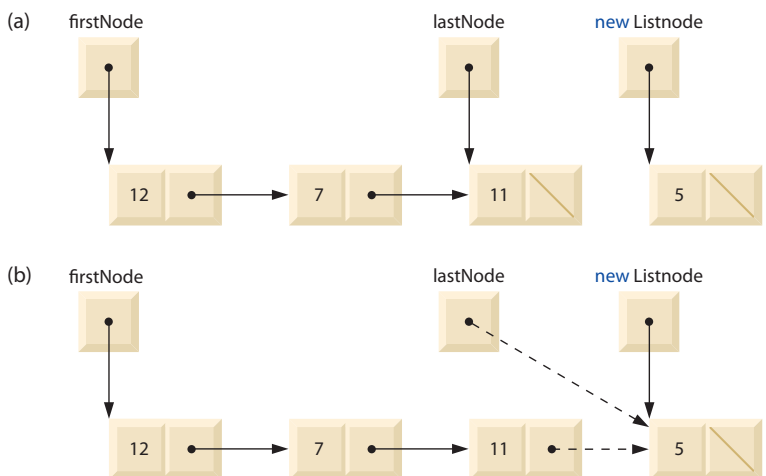


Figura 22.7 | Representação gráfica da operação `insertAtBack`.

Método `List` `removeFromFront`

O método `removeFromFront` (linhas 78–92 da Figura 22.3) remove o primeiro nó da lista e retorna uma referência aos dados removidos. Se a lista estiver vazia quando o programa chamar esse método, o método lançará uma `EmptyListException` (linhas 80–81). Caso contrário, o método retorna uma referência aos dados removidos. Os passos são:

1. Atribua `firstNode.data` (os dados sendo removidos) a `removedItem` (linha 83).
2. Se `firstNode` e `lastNode` referenciarem o mesmo objeto (linha 86), a lista terá apenas um elemento nessa hora. Então, o método configura `firstNode` e `lastNode` como `null` (linha 87) para remover o nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó, então o método deixa a referência `lastNode` como está e atribui o valor de `firstNode.nextNode` ao `firstNode` (linha 89). Portanto, `firstNode` referencia o nó que, anteriormente, era o segundo nó na lista.
4. Retornar a referência `removedItem` (linha 91).

Na Figura 22.8, parte (a) ilustra a lista antes da operação de remoção. As linhas tracejadas e setas na parte (b) mostram as manipulações de referência.

Método `List` `removeFromBack`

O método `removeFromBack` (linhas 95–118 da Figura 22.3) remove o último nó de uma lista e retorna uma referência aos dados removidos. O método lança uma `EmptyListException` (linhas 97–98) se a lista estiver vazia quando o programa chamar esse método. Os passos são:

1. Atribuir `lastNode.data` (os dados sendo removidos da lista) a `removedItem` (linha 100).

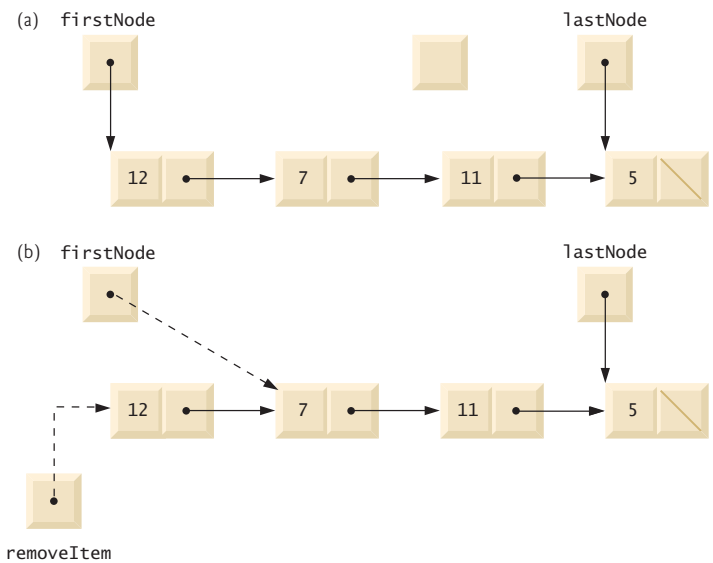


Figura 22.8 | Representação gráfica da operação `removeFromFront`.

2. Se `firstNode` e `lastNode` referenciarem o mesmo objeto (linha 103), a lista terá apenas um elemento nessa hora. Então, a linha 104 configura `firstNode` e `lastNode` como `null` para remover esse nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó, crie a referência `ListNode current` e a atribua `firstNode` (linha 107).
4. Agora "percorrer a lista" com `current` até ela referenciar o nó antes do último nó. O loop `while` (linhas 110–111) atribui `current.nextNode` a `current` contanto que `current.nextNode` (o próximo nó na lista) não seja `lastNode`.
5. Depois de localizar o penúltimo nó, atribuir `current` a `lastNode` (linha 113) para atualizar qual o nó é o último na lista.
6. Configurar o `current.nextNode` como `null` (linha 114) para remover o último nó da lista e termina a lista no nó atual.
7. Retornar a referência `removedItem` (linha 117).

Na Figura 22.9, parte (a) ilustra a lista antes da operação de remoção. As linhas tracejadas e setas na parte (b) mostram as manipulações de referência.

Método `List` `print`

O método `print` (linhas 127–146) primeiro determina se a lista está vazia (linhas 129–133). Se estiver, `print` exibe uma mensagem que indica que a lista está vazia e o controle retorna o método chamador. Caso contrário, `print` gera saída dos dados da lista. A linha 136 cria `ListNode current` e a inicializa com `firstNode`. Enquanto `current` não estiver `null`, há mais itens na lista. Portanto, a linha 141 gera saída de uma representação de string de `current.data`. A linha 142 muda para o próximo nó na lista atribuindo o valor de referência `current.nextNode` a `current`. Esse algoritmo de impressão é idêntico para listas vinculadas, pilhas e filas.

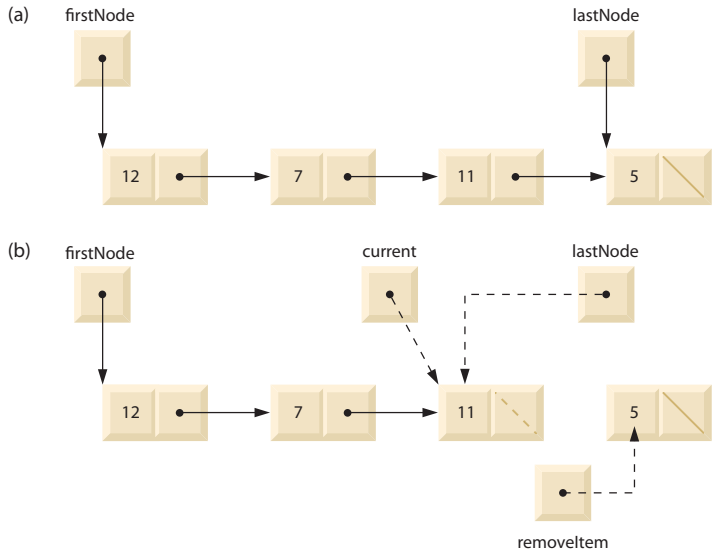


Figura 22.9 | Representação gráfica da operação `removeFromBack`.

22.5 Pilhas

Uma pilha é uma versão limitada de uma lista — novos nós podem ser inseridos e removidos de uma pilha apenas na parte superior. Por essa razão, uma pilha é referida como uma estrutura de dados **primeiro a entrar, primeiro a sair (Last-In, First-Out — LIFO)**. O membro link no último nó é configurado como `null` para indicar a parte inferior da pilha. Observe que não é necessário que uma pilha seja implementada como uma lista encadeada — ela também pode ser implementada utilizando um array.

Os principais métodos para manipular uma pilha são **`push`** e **`pop`**, que adicionam um novo nó à superior da pilha e removem um nó da superior da pilha, respectivamente. O método `pop` também retorna os dados do nó removido.

As pilhas têm muitas aplicações interessantes. Por exemplo, quando um programa chama um método, o método chamado deve saber retornar ao seu chamador, assim o endereço de retorno do método chamador é inserido na pilha de execução do programa (discutido na Seção 6.6). Se uma série de chamadas de método ocorre, os sucessivos endereços de retorno são empilhados na ordem primeiro a entrar, primeiro a sair de modo que cada método possa retornar para seu chamador. As pilhas suportam as chamadas de método recursivo da mesma maneira que as chamadas de método não recursivo convencionais.

A pilha de execução de programa também contém a memória criada para variáveis locais a cada invocação de um método durante a execução de um programa. Quando o método retorna para seu chamador, a memória para variáveis locais desse método é removida da pilha e essas variáveis não são mais conhecidas para o programa. Se a variável local for uma referência e o objeto referenciado não tiver nenhuma outra variável referenciando-o, o objeto pode sofrer coleta de lixo.

Os compiladores utilizam pilhas para avaliar expressões aritméticas e gerar o código de linguagem de máquina para processá-las. Os exercícios neste capítulo exploram várias aplicações de pilhas, incluindo seu uso para desenvolver um compilador completo. Além disso, o pacote `java.util` contém a classe `Stack` (ver Capítulo 20) para implementar e manipular pilhas que podem crescer e encolher durante a execução do programa.

Nesta seção, tiramos proveito do íntimo relacionamento entre lista e pilha para implementar uma classe de pilha reutilizando a classe `List<T>` da Figura 22.3. Demonstramos duas formas diferentes da capacidade de reutilização. Em primeiro lugar, implementamos a classe de pilha estendendo a classe `List`. Em seguida, implementamos uma classe de pilha de execução idêntica por meio da composição incluindo uma referência a um objeto `List` como um variável de instância `private`. As estruturas de dados de lista, pilha e fila neste capítulo são implementadas para armazenar referências a objetos de qualquer tipo a fim de encorajar mais a capacidade de reutilização.

Classe de pilha que herda de `List<T>`

As figuras 22.10–22.11 criam e manipulam uma classe de pilha que estende a classe `List<T>` da Figura 22.3. Queremos que a pilha tenha os métodos `push`, `pop`, `isEmpty` e `print`. Essencialmente, esses são os métodos `List<T>` `insertAtFront`, `removeFromFront`, `isEmpty` e `print`. Naturalmente, a classe `List<T>` contém outros métodos (como `insertAtBack` e `removeFromBack`) que não tornaríamos acessível pela interface `public` à classe de pilha. É importante lembrar que todos os métodos na interface de classe `public` da `List<T>` também são métodos `public` da subclasse `StackInheritance<T>` (Figura 22.10). Cada método de `StackInheritance<T>` chama o método `List<T>` apropriado — por exemplo, método `push` chama `insertAtFront` e o método `pop` chama `removeFromFront`. Clientes `StackInheritance<T>` podem chamar os métodos `isEmpty` e `print` porque eles são herdados de `List<T>`. A classe `StackInheritance<T>` é declarada no pacote `com.deitel.ch22` (linha 3) para a reutilização. Observe que `StackInheritance<T>` não importa `List<T>` — as classes estão no mesmo pacote.

```
1  // Figura 22.10: StackInheritance.java
2  // StackInheritance estende a classe List.
3  package com.deitel.ch22;
4
5  public class StackInheritance< T > extends List< T >
6  {
7     // construtor sem argumento
8     public StackInheritance()
9     {
10       super( "stack" );
11    } // fim do construtor sem argumento StackInheritance
12
13    // adiciona objeto à pilha
14    public void push( T object )
15    {
16       insertAtFront( object );
17    } // fim do método push
18
19    // remove objeto da pilha
20    public T pop() throws EmptyListException
21    {
22       return removeFromFront();
23    } // fim do método pop
24 } // fim da classe StackInheritance
```

Figura 22.10 | `StackInheritance` estende a classe `List`.

O método `main` da classe `StackInheritanceTest` (Figura 22.11) cria um objeto da classe `StackInheritance<T>` chamado `stack` (linhas 10–11). O programa adiciona inteiros na pilha (linhas 14, 16, 18 e 20). O autoboxing é utilizado aqui para inserir objetos `Integer` na estrutura de dados. As linhas 28–33 removem os objetos da pilha em um loop `while` infinito. Se o método `pop` for invocado em uma pilha vazia, o método lança uma `EmptyListException`. Nesse caso, o programa exibe o rastreamento de pilha da exceção, que mostra os métodos na pilha de execução do programa no momento em que a exceção ocorreu. Observe que o programa utiliza o método `print` (herdado de `List`) para gerar a saída do conteúdo da pilha.

```
1  // Figura 22.11: StackInheritanceTest.java
2  // Programa de manipulação de pilha.
3  import com.deitel.ch22.StackInheritance;
4  import com.deitel.ch22.EmptyListException;
5
6  public class StackInheritanceTest
7  {
8     public static void main( String[] args )
9     {
10       StackInheritance< Integer > stack =
11          new StackInheritance< Integer >();
12
13       // utiliza método push
14       stack.push( -1 );
15       stack.print();
16       stack.push( 0 );
17       stack.print();
18       stack.push( 1 );
19       stack.print();
20       stack.push( 5 );
21       stack.print();
22
23       // remove itens de pilha
24       try
25       {
26          int removedItem;
27
28          while ( true )
29          {
30             removedItem = stack.pop(); // utiliza método pop
31             System.out.printf( "\n%d popped\n", removedItem );
```

```
32              stack.print();
33           } // fim do while
34        } // fim do try
35        catch ( EmptyListException emptyListException )
36        {
37           emptyListException.printStackTrace();
38        } // fim do catch
39     } // fim de main
40  } // fim da classe StackInheritanceTest
```

```
The stack is: -1
The stack is: 0 -1
The stack is: 1  0 -1
The stack is: 5  1  0 -1

5 popped
The stack is: 1 0 -1

1 popped
The stack is: 0 -1

0 popped
The stack is: -1

-1 popped
Empty stack
com.deitel.ch22.EmptyListException: stack is empty
        at com.deitel.ch22.List.removeFromFront(List.java:81)
        at com.deitel.ch22.StackInheritance.pop(StackInheritance.java:22)
        at StackInheritanceTest.main(StackInheritanceTest.java:30)
```

Figura 22.11 | Programa de manipulação de pilha.

Classe de pilha que contém uma referência a uma `List`

Você também pode implementar uma classe reutilizando uma classe de lista por composição. A Figura 22.12 utiliza uma `private List<T>` (linha 7) na declaração de classe da `StackComposition<T>`. A composição permite que ocultemos os métodos `List<T>` que não devem estar na interface `public` da nossa pilha. Fornecemos os métodos da interface `public` que utilizam somente os métodos `List<T>` necessários. A implementação de cada método de pilha como uma chamada para um método `List<T>` chama-se **delegação** — o método de pilha invocado delega a chamada para o método `List<T>` apropriado. Em particular, `StackComposition<T>` delega chamadas para os métodos `List<T>` `insertAtFront`, `removeFromFront`, `isEmpty` e `print`. Nesse exemplo, não mostramos a classe `StackCompositionTest`, porque a única diferença é que alteramos o tipo da pilha de `StackInheritance` para `StackComposition` (linhas 3 e 10–11 da Figura 22.11).

```
1  // Figura 22.12: StackComposition.java
2  // StackComposition utiliza um objeto List composto.
3  package com.deitel.ch22;
4
5  public class StackComposition< T >
6  {
7     private List< T > stackList;
8
9     // construtor sem argumento
10    public StackComposition()
11    {
12       stackList = new List< T >( "stack" );
13    } // fim do construtor sem argumento StackComposition
14
15    // adiciona objeto à pilha
16    public void push( T object )
17    {
18       stackList.insertAtFront( object );
19    } // fim do método push
20
21    // remove o objeto da pilha
22    public T pop() throws EmptyListException
```

```
23       {
24          return stackList.removeFromFront();
25       } // fim do método pop
26
27       // determina se a pilha está vazia
28       public boolean isEmpty()
29       {
30          return stackList.isEmpty();
31       } // fim do método isEmpty
32
33       // gera saída do conteúdo de pilha
34       public void print()
35       {
36          stackList.print();
37       } // fim do método print
38    } // fim da classe StackComposition
```

Figura 22.12 | `StackComposition` utiliza um objeto `List` composto.

22.6 Filas

Outra estrutura de dados comumente utilizada é a fila. Uma fila é semelhante a uma fila de caixa de um supermercado — o caixa atende a primeira pessoa da fila. Outros clientes entram no fim da fila e esperam ser atendidos. Os nós de fila são removidos apenas a partir da cabeça (ou no início) da fila e são inseridos apenas na cauda (ou no fim). Por essa razão, uma fila é uma estrutura de dados **primeiro a entrar, primeiro a sair (First-In, First-Out — FIFO)**. As operações de inserção e remoção são conhecidas como **enqueue** e **dequeue**.

As filas têm muitas utilizações em sistemas de computador. Cada CPU em um computador pode servir somente um aplicativo por vez. Todo aplicativo que requer tempo de processador é colocado em uma fila. O aplicativo na frente da fila é o próximo a receber o serviço. Cada aplicativo avança gradualmente para frente à medida que os aplicativos antes dele recebem o serviço.

As filas também são utilizadas para suportar **spooling de impressão**. Por exemplo, uma única impressora talvez seja compartilhada por todos os usuários de uma rede. Muitos usuários podem enviar trabalhos de impressão à impressora, mesmo quando a impressora já estiver ocupada. Esses trabalhos de impressão são colocados em uma fila até a impressora ficar disponível. Um programa chamado **spooler** gerencia a fila para assegurar que, à medida que cada trabalho de impressão seja concluído, o próximo trabalho de impressão seja enviado à impressora.

Os pacotes de informações também esperam em filas em redes de computadores. Toda vez que um pacote chegar a um nó de rede, ele deve ser roteado para o próximo nó ao longo do caminho para o destino final do pacote. O nó de roteamento roteia um pacote por vez, então pacotes adicionais são enfileirados até o roteador conseguir roteá-los.

Um servidor de arquivos em uma rede de computadores trata as solicitações de acesso a arquivo de muitos clientes por toda a rede. Os servidores têm uma capacidade limitada para servir solicitações de clientes. Quando essa capacidade é excedida, as solicitações dos clientes esperam em filas.

A Figura 22.13 cria uma classe `Queue<T>` que contém um objeto `List<T>` (Figura 22.3) e fornece os métodos `enqueue`, `dequeue`, `isEmpty` e `print`. A classe `List<T>` contém alguns métodos (por exemplo, `insertAtFront` e `removeFromBack`) que preferimos não tornar acessíveis por meio da interface `public` da classe `Queue<T>` . Utilizar a composição permite que ocultemos outros métodos `public` da classe `List<T>` a partir de clientes da classe `Queue<T>`. Cada método `Queue<T>` chama um método `List<T>` apropriado — o método `enqueue` chama o método `List<T>` `insertAtBack`, o método `dequeue` chama o método `List<T>` `removeFromFront`, o método `isEmpty` chama o método `List<T>` `isEmpty` e o método `print` chama o método `List<T>` `print`. Para a reutilização, a classe `Queue<T>` é declarada no pacote `com.deitel.ch22`.

```
1  // Figura 22.13: Queue.java
2  // Queue utiliza a classe List.
3  package com.deitel.ch22;
4
5  public class Queue
6  {
7     private List< T > queueList;
8
9     // construtor sem argumento
10    public Queue()
11    {
12       queueList = new List< T >( "queue" );
13    } // fim do construtor sem argumento Queue
14
```

```
15       // adiciona o objeto à fila
16       public void enqueue( T object )
17       {
18          queueList.insertAtBack( object );
19       } // fim do método enqueue
20
21       // remove o objeto da fila
22       public T dequeue() throws EmptyListException
23       {
24          return queueList.removeFromFront();
25       } // fim do método dequeue
26
27       // determina se a fila está vazia
28       public boolean isEmpty()
29       {
30          return queueList.isEmpty();
31       } // fim do método isEmpty
32
33       // gera o conteúdo da fila
34       public void print()
35       {
36          queueList.print();
37       } // fim do método print
38    } // fim da classe Queue
```

Figura 22.13 | `Queue` utiliza a classe `List`.

O método `main` da classe `QueueTest` (Figura 22.14) cria e inicializa a variável `Queue<T> queue` (linha 10). As linhas 13, 15, 17 e 19 enfileiram quatro inteiros, tirando proveito de autoboxing para inserir objetos `Integer` na fila. As linhas 27–32 utilizam um loop infinito para desenfileirar os objetos na ordem primeiro a entrar, primeiro a sair. Quando a fila está vazia, o método `dequeue` lança uma `EmptyListException` e o programa exibe o rastreamento de pilha da exceção.

```
1    // Figura 22.14: QueueTest.java
2    // Classe QueueTest.
3    import com.deitel.ch22.Queue;
4    import com.deitel.ch22.EmptyListException;
5
6    public class QueueTest
7    {
8       public static void main( String[] args )
9       {
10         Queue< Integer > queue = new Queue< Integer >();
11
12         // utiliza o método enqueue
13         queue.enqueue( -1 );
14         queue.print();
15         queue.enqueue( 0 );
16         queue.print();
17         queue.enqueue( 1 );
18         queue.print();
19         queue.enqueue( 5 );
20         queue.print();
21
22         // remove os objetos da fila
23         try
24         {
25            int removedItem;
26
27            while ( true )
28            {
29               removedItem = queue.dequeue(); // utiliza método dequeue
30               System.out.printf( "\n%d dequeued\n", removedItem );
31               queue.print();
32            } // fim do while
33         } // fim do try
34         catch ( EmptyListException emptyListException )
35         {
```

```
            emptyListException.printStackTrace();
         } // fim do catch
      } // fim de main
   } // fim da classe QueueTest
```

```
The queue is: -1
The queue is: -1 0
The queue is: -1 0 1
The queue is: -1 0 1 5

-1 dequeued
The queue is: 0 1 5

0 dequeued
The queue is: 1 5

1 dequeued
The queue is: 5

5 dequeued
Empty queue
com.deitel.ch22.EmptyListException: queue is empty
        at com.deitel.ch22.List.removeFromFront(List.java:81)
        at com.deitel.ch22.Queue.dequeue(Queue.java:24)
        at QueueTest.main(QueueTest.java:29)
```

Figura 22.14 | Programa de processamento de fila.

22.7 Árvores

Listas, pilhas e filas são **estruturas de dados lineares** (isto é, **sequências**). Uma árvore é uma estrutura de dados bidimensional, não linear, com propriedades especiais. Os nós da árvore contêm dois ou mais links. Esta seção discute as árvores binárias (Figura 22.15) — árvores cujos nós contêm, cada um, dois links (nenhum, um ou ambos os quais podem ser `null`). O **nó raiz** é o primeiro nó em uma árvore. Cada link no nó raiz referencia um **filho**. O **filho esquerdo** é o primeiro nó na **subárvore esquerda** (também conhecido como o nó raiz da subárvore esquerda) e o **filho direito** é o primeiro nó na **subárvore direita** (também conhecido como o nó raiz da subárvore direita). Os filhos de um nó específico são chamados **irmãos**. Um nó sem filhos é chamado de **nó de folha**. Os cientistas da computação normalmente desenham árvores indo do nó raiz para baixo — o oposto da maneira como a maioria das árvores cresce na natureza.

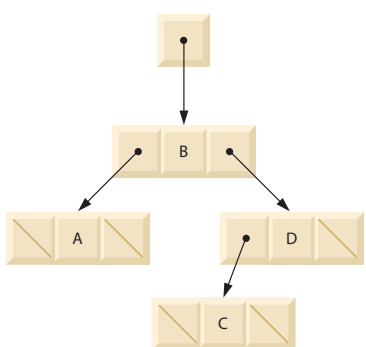


Figura 22.15 | Representação gráfica da árvore binária.

Em nosso exemplo, criamos uma árvore binária especial chamada **árvore de pesquisa binária**. Uma árvore de pesquisa binária (sem valores de nó duplicados) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor no nó pai dessa subárvore e os valores em qualquer subárvore direita são maiores que o valor no nó pai dessa subárvore. A Figura 22.16 ilustra uma árvore de pesquisa binária com 12 valores de inteiro. Observe que a forma da árvore de pesquisa binária que corresponde a um conjunto de dados pode variar, dependendo da ordem em que os valores são inseridos na árvore.

As figuras 22.17–22.18 criam uma classe de árvore de pesquisa binária genérica e a utilizam para manipular uma árvore de números inteiros. O aplicativo na Figura 22.18 percorre a árvore (isto é, passa por todos os seus nós) de três maneiras — utilizando **percursos recur-**

sivos na ordem, na **pré-ordem** e na **pós-ordem**. O programa gera 10 números aleatórios e insere cada um na árvore. A classe `Tree<T>` é declarada no pacote `com.deitel.ch22` para a reutilização.

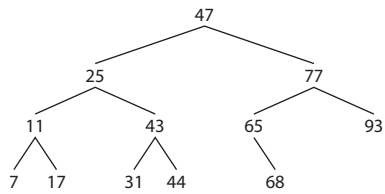


Figura 22.16 | Árvore de pesquisa binária que contém 12 valores.

```
1  // Figura 22.17: Tree.java
2  // Declarações de classe TreeNode e Tree para uma árvore de pesquisa binária.
3  package com.deitel.ch22;
4
5  // definição da classe TreeNode
6  class TreeNode< T extends Comparable< T > >
7  {
8     // membros de acesso de pacote
9     TreeNode< T > leftNode; // nó esquerdo
10    T data; // valor do nó
11    TreeNode< T > rightNode; // nó direito
12
13    // construtor inicializa os dados e os torna um nó de folha
14    public TreeNode( T nodeData )
15    {
16       data = nodeData;
17       leftNode = rightNode = null; // o nó não tem nenhum filho
18    } // fim do construtor TreeNode
19
20    // localiza ponto de inserção e insere novo nó; ignora os valores duplicados
21    public void insert( T insertValue )
22    {
23       // insere na subárvore esquerda
24       if ( insertValue.compareTo( data ) < 0 )
25       {
26          // insere novo TreeNode
27          if ( leftNode == null )
28             leftNode = new TreeNode< T >( insertValue );
29          else // continua percorrendo a subárvore esquerda recursivamente
30             leftNode.insert( insertValue );
31       } // fim do if
32       // insere na subárvore direita
33       else if ( insertValue.compareTo( data ) > 0 )
34       {
35          // insere novo TreeNode
36          if ( rightNode == null )
37             rightNode = new TreeNode< T >( insertValue );
38          else // continua percorrendo a subárvore direita recursivamente
39             rightNode.insert( insertValue );
40       } // fim de else if
41    } // fim do método insert
42 } // fim da classe TreeNode
43
44 // definição da classe Tree
45 public class Tree< T extends Comparable< T > >
46 {
47    private TreeNode< T > root;
48
49    // construtor inicializa uma Tree de inteiros vazia
50    public Tree()
51    {
```

```
52          root = null;
53       } // fim do construtor sem argumento Tree
54
55       // insere um novo nó na árvore de pesquisa binária
56       public void insertNode( T insertValue )
57       {
58          if ( root == null )
59             root = new TreeNode< T >( insertValue ); // cria o nó raiz
60          else
61             root.insert( insertValue ); // chama o método insert
62       } // fim do método insertNode
63
64       // inicia percurso na pré-ordem
65       public void preorderTraversal()
66       {
67          preorderHelper( root );
68       } // fim do método preorderTraversal
69
70       // método recursivo para realizar percurso na pré-ordem
71       private void preorderHelper( TreeNode< T > node )
72       {
73          if ( node == null )
74             return;
75
76          System.out.printf( "%s ", node.data ); // gera saída de dados do nó
77          preorderHelper( node.leftNode );       // percorre subárvore esquerda
78          preorderHelper( node.rightNode );      // percorre subárvore direita
79       } // fim do método preorderHelper
80
81       // inicia percurso na ordem
82       public void inorderTraversal()
83       {
84          inorderHelper( root );
85       } // fim do método inorderTraversal
86
87       // método recursivo para realizar percurso na ordem
88       private void inorderHelper( TreeNode< T > node )
89       {
90          if ( node == null )
91             return;
92
93          inorderHelper( node.leftNode );        // percorre subárvore esquerda
94          System.out.printf( "%s ", node.data ); // gera saída de dados do nó
95          inorderHelper( node.rightNode );       // percorre subárvore direita
96       } // fim do método inorderHelper
97
98       // inicia percurso na pós-ordem
99       public void postorderTraversal()
100      {
101         postorderHelper( root );
102      } // fim do método postorderTraversal
103
104      // método recursivo para realizar percurso na pós-ordem
105      private void postorderHelper( TreeNode< T > node )
106      {
107         if ( node == null )
108            return;
109
110         postorderHelper( node.leftNode );      // percorre subárvore esquerda
111         postorderHelper( node.rightNode );     // percorre subárvore direita
112         System.out.printf( "%s ", node.data ); // gera saída de dados do nó
113      } // fim do método postorderHelper
114   } // fim da classe Tree
```

Figura 22.17 | Declarações da classe `TreeNode` e `Tree` para uma árvore de pesquisa binária.

```
1  // Figura 22.18: TreeTest.java
2  // Programa de teste da árvore binária.
3  import java.util.Random;
4  import com.deitel.ch22.Tree;
5
6  public class TreeTest
7  {
8     public static void main( String[] args )
9     {
10       Tree< Integer > tree = new Tree< Integer >();
11       int value;
12       Random randomNumber = new Random();
13
14       System.out.println( "Inserting the following values: " );
15
16       // insere 10 inteiros aleatórios de 0-99 na árvore
17       for ( int i = 1; i <= 10; i++ )
18       {
19          value = randomNumber.nextInt( 100 );
20          System.out.printf( "%d ", value );
21          tree.insertNode( value );
22       } // for final
23
24       System.out.println ( "\n\nPreorder traversal" );
25       tree.preorderTraversal(); // realiza percurso na pré-ordem da árvore
26
27       System.out.println ( "\n\nInorder traversal" );
28       tree.inorderTraversal(); // realiza percurso na ordem da árvore
29
30       System.out.println ( "\n\nPostorder traversal" );
31       tree.postorderTraversal(); // realiza percurso na pós-ordem da árvore
32       System.out.println();
33    } // fim de main
34 } // fim da classe TreeTest
```

```
Inserting the following values:
49 64 14 34 85 64 46 14 37 55

Preorder traversal
49 14 34 46 37 64 55 85

Inorder traversal
14 34 37 46 49 55 64 85

Postorder traversal
37 46 34 14 55 85 64 49
```

Figura 22.18 | Programa de teste da árvore binária.

Vamos percorrer o programa de árvore binária. O método `main` da classe `TreeTest` (Figura 22.18) começa instanciando um objeto `Tree<T>` vazio e atribuindo sua referência à variável `tree` (linha 10). As linhas 17–22 geram 10 inteiros aleatoriamente, cada um dos quais é inserido na árvore chamando o método `insertNode` (linha 21). O programa então realiza percursos na pré-ordem, na ordem e pós-ordem (esses serão explicados em breve) de `tree` (linhas 25, 28 e 31, respectivamente).

Visão geral da classe `Tree`

A classe `Tree` (Figura 22.17, linhas 45–114) requer que seu argumento de tipo implemente a interface `Comparable`, para que cada valor inserido na árvore possa ser comparado com os valores existentes para determinar o ponto de inserção. A classe tem o campo `private` `root` (linha 47) — uma referência `TreeNode` ao nó raiz da árvore. O construtor de `Tree` (linhas 50–53) inicializa `root` como `null` para indicar que a árvore está vazia. A classe contém o método `insertNode` (linhas 56–62) para inserir um novo nó na árvore e os métodos `preorderTraversal` (linhas 65–68), `inorderTraversal` (linhas 82–85) e `postorderTraversal` (linhas 99–102) para iniciar os percursos da árvore. Cada um desses métodos chama um método utilitário recursivo para realizar as operações de percurso na representação interna da árvore. (Discutimos a recursão no Capítulo 18.)

Método `Tree` `insertNode`

O método `insertNode` da classe `Tree` (linhas 56–62) primeiro determina se a árvore está vazia. Se estiver, a linha 59 aloca um novo `TreeNode`, inicializa o nó com o valor que está sendo inserido na árvore e atribui o novo nó à referência `root`. Se a árvore não estiver vazia, a linha 61 chama o método `TreeNode` `insert` (linhas 21–41). Esse método utiliza a recursão para determinar a localização do novo nó na árvore e insere o nó nessa localização. Um nó pode ser inserido apenas como um nó de folha em uma árvore de pesquisa binária.

Método `TreeNode insert`

O método `TreeNode` `insert` compara o valor a ser inserido com o valor `data` no nó raiz. Se o valor de inserção for menor que os dados do nó raiz (linha 24), o programa determina se a subárvore esquerda está vazia (linha 27). Se estiver, a linha 28 aloca um novo `TreeNode`, inicializa este com o valor que está sendo inserido e atribui o novo nó à referência `leftNode`. Caso contrário, a linha 30 chama recursivamente `insert` para a subárvore esquerda para inserir o valor na subárvore esquerda. Se o valor de inserção for maior que os dados do nó raiz (linha 33), o programa determina se a subárvore direita está vazia (linha 36). Se estiver, a linha 37 aloca um novo `TreeNode`, inicializa este com o valor que está sendo inserido e atribui o novo nó à referência `rightNode`. Caso contrário, a linha 39 chama recursivamente `insert` para a subárvore direita a fim de inserir o valor na subárvore direita. Se o `insertValue` já estiver na árvore, ele é simplesmente ignorado.

Métodos `Tree inorderTraversal`, `preorderTraversal` e `postorderTraversal`

Os métodos `inorderTraversal`, `preorderTraversal` e `postorderTraversal` chamam os métodos auxiliares `Tree` `inorderHelper` (linhas 88–96), `preorderHelper` (linhas 71–79) e `postorderHelper` (linhas 105–113), respectivamente, para percorrer a árvore e imprimir os valores de nó. Os métodos auxiliares da classe `Tree` permitem iniciar um percurso sem passar o nó `root` para o método. A referência `root` é um detalhe de implementação que o programador não deve ser capaz de acessar. Os métodos `inorderTraversal`, `preorderTraversal` e `postorderTraversal` simplesmente aceitam a referência `root` privada e passam-na para o método auxiliar apropriado para iniciar um percurso da árvore. O caso básico para cada método auxiliar determina se a referência que ele recebe é `null` e, se for, retorna imediatamente.

O método `inorderHelper` (linhas 88–96) define os passos para um percurso na ordem:

1. Percorra a subárvore esquerda com uma chamada a `inorderHelper` (linha 93).
2. Processe o valor no nó (linha 94).
3. Percorra a subárvore direita com uma chamada a `inorderHelper` (linha 95).

O percurso na ordem não processa o valor em um nó até que os valores na subárvore esquerda desse nó sejam processados. O percurso na ordem da árvore na Figura 22.19 é

```
6 13 17 27 33 42 48
```

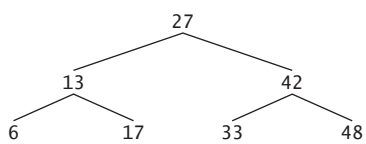


Figura 22.19 | A árvore de pesquisa binária com sete valores.

Observe que o percurso na ordem de uma árvore de pesquisa binária imprime os valores de nó na ordem crescente. O processo de criar uma árvore de pesquisa binária realmente classifica os dados; portanto, ele é chamado de **classificação de árvore binária**.

O método `preorderHelper` (linhas 71–79) define os passos para um percurso na pré-ordem:

1. Processe o valor no nó (linha 76).
2. Percorra a subárvore esquerda com uma chamada a `preorderHelper` (linha 77).
3. Percorra a subárvore direita com uma chamada a `preorderHelper` (linha 78).

O percurso na pré-ordem processa o valor em cada nó quando o nó é visitado. Depois de processar o valor em um nó em particular, o percurso na pré-ordem processa os valores na subárvore esquerda e, então, processa os valores na subárvore direita. O percurso na pré-ordem da árvore na Figura 22.19 é

```
27 13 6 17 42 33 48
```

O método `postorderHelper` (linhas 105–113) define os passos para um percurso na pós-ordem:

1. Percorra a subárvore esquerda com uma chamada a `postorderHelper` (linha 110).
2. Percorra a subárvore direita com uma chamada a `postorderHelper` (linha 111).
3. Processe o valor no nó (linha 112).

O percurso na pós-ordem processa o valor em cada nó depois que todos os filhos desse nó forem processados. A `postorderTraversal` da árvore na Figura 22.19 é

```
6 17 13 33 48 42 27
```

A árvore de pesquisa binária facilita a **eliminação de duplicatas**. Na construção de uma árvore, a operação de inserção reconhece as tentativas de inserir um valor duplicado, porque uma duplicata segue as mesmas decisões "vá para esquerda" ou "vá para direita", em cada

comparação, que o valor original seguiu. Portanto, a operação de inserção por fim compara a duplicata com um nó que contém o mesmo valor. Nesse ponto, a operação de inserção pode decidir descartar o valor duplicado (como fizemos nesse exemplo).

Pesquisar um valor que corresponda a um valor-chave em uma árvore binária é rápido, especialmente em **árvores fortemente empacotadas** (ou **equilibradas**). Em uma árvore compactada, cada nível contém aproximadamente duas vezes o número de elementos que o nível anterior. A Figura 22.19 é uma árvore binária compactada. Uma árvore de pesquisa binária empacotada fortemente com n elementos tem $\log_2 n$ níveis. Portanto, no máximo $\log_2 n$ comparações são necessárias para localizar uma correspondência ou determinar que não há nenhuma correspondência. Pesquisar em uma árvore de pesquisa binária (fortemente empacotada) de 1.000 elementos requer no máximo 10 comparações, pois $2^{10} > 1.000$. Pesquisar em uma árvore de pesquisa binária (fortemente empacotada) de 1.000.000 elementos requer no máximo 20 comparações, pois $2^{20} > 1.000.000$.

Os exercícios do capítulo apresentam algoritmos para várias outras operações de árvore binária, como excluir um item de uma árvore binária, imprimir uma árvore binária em um formato de árvore bidimensional e realizar um **percurso na ordem de nível de uma árvore binária**. O percurso na ordem de nível visita os nós da árvore linha por linha, iniciando no nível do nó raiz. Em cada nível da árvore, um percurso na ordem de nível visita os nós da esquerda para direita. Outros exercícios de árvore binária incluem permitir uma árvore de pesquisa binária conter valores duplicados, inserir valores de string em uma árvore binária e determinar quantos níveis estão contidos em uma árvore binária.

22.8 Conclusão

Este capítulo completa nossa apresentação de estruturas de dados. Nós a iniciamos no Capítulo 20 com uma introdução às coleções predefinidas da estrutura de coleções Java e continuamos no Capítulo 21 mostrando como implementar coleções e métodos genéricos. Neste capítulo, você aprendeu a construir estruturas de dados dinâmicas genéricas que crescem e encolhem em tempo de execução. Aprendeu que as listas encadeadas são coleções de itens de dados que são "vinculados em uma cadeia." Também viu que um aplicativo pode realizar inserções e exclusões no começo e no fim de uma lista encadeada. Você aprendeu que as estruturas de dados pilha e fila são versões limitadas de listas. Quanto às pilhas, vimos que as inserções e exclusões só podem ser feitas na parte superior. Quanto às filas, que representam filas de espera, você viu que as inserções são feitas na cauda (na parte de trás) e as exclusões são feitas na cabeça (na parte da frente). Também aprendeu a estrutura de dados da árvore binária. Você viu uma árvore de pesquisa binária que facilitou a pesquisa e a classificação de dados em alta velocidade e a eliminação eficiente de itens de dados duplicados. Por todo o capítulo, aprendeu a criar e a empacotar essas estruturas de dados para capacidade de reutilização e de manutenção.

Depois, introduzimos os applets Java — programas Java que, em geral, executam em um navegador. Apresentamos uma visão geral dos applets de exemplo do JDK e, em seguida, mostramos como escrever e executar os seus próprios applets. Depois introduzimos as capacidades Java Web Start para carregar um applet e instalar um atalho de área de trabalho a fim de recarregar futuramente o applet sem a necessidade de revisitar o site do applet.

Resumo

Seção 22.1 Introdução

- As estruturas de dados dinâmicas podem crescer e encolher em tempo de execução.
- As listas encadeadas são coleções de itens de dados "vinculados em uma cadeia" — as inserções e exclusões podem ser feitas em qualquer lugar de uma lista encadeada.
- As pilhas são importantes em compiladores e sistemas operacionais — as inserções e as exclusões são feitas somente na parte superior de uma pilha.
- Em uma fila, as inserções são feitas na parte de trás (cauda), e as exclusões, na parte da frente (cabeça).
- As árvores binárias facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos e a compilação de expressões em linguagem de máquina.

Seção 22.2 Classes autorreferenciais

- Uma classe autorreferencial contém uma referência que referencia outro objeto do mesmo tipo de classe. Objetos autorreferenciais podem ser vinculados para formar estruturas de dados dinâmicas.

Seção 22.3 Alocação dinâmica de memória

- O limite para alocação dinâmica de memória pode ser tão grande quanto a memória física disponível no computador ou o espaço em disco disponível em um sistema de memória virtual. Frequentemente, os limites são bem menores, porque a memória disponível do computador deve ser compartilhada entre muitos usuários.
- Se nenhuma memória estiver disponível, um `OutOfMemoryError` é lançado.

Seção 22.4 Listas vinculadas

- Uma lista vinculada é acessada via uma referência ao primeiro nó da lista. Cada nó subsequente é acessado via o membro de referência de link armazenado no nó anterior.
- Por convenção, a referência de link no último nó de uma lista é configurada como `null` para marcar o fim da lista.
- Um nó pode conter dados de qualquer tipo, incluindo objetos de outras classes.
- Uma lista encadeada é apropriada quando o número de elementos de dados a ser armazenado for imprevisível. As listas vinculadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário.
- O tamanho de um array Java "convencional" não pode ser alterado — ele é fixado no momento da criação.
- Os nós de lista normalmente não são armazenados na memória contígua. Em vez disso, são logicamente contíguos.

Seção 22.5 Pilhas

- Uma pilha é uma estrutura de dados do tipo último a entrar, primeiro a sair (Last-In, First-Out — LIFO). Os principais métodos utilizados para manipular uma pilha são `push` e `pop`, que adicionam um novo nó à parte superior da pilha e removem um nó da parte superior da pilha, respectivamente. O método `pop` retorna os dados do nó removido.
- Quando uma chamada de método é feita, o método chamado deve saber retornar para seu chamador, assim o endereço de retorno é inserido na pilha de execução do programa. Se uma série de chamadas de método ocorre, os valores sucessivos de retorno são inseridos na pilha na ordem primeiro a entrar, primeiro a sair de modo que cada método possa retornar para seu chamador.
- A pilha de execução do programa contém o espaço criado para variáveis locais a cada invocação de um método. Quando o método retorna para seu chamador, o espaço para variáveis locais desse método é removido da pilha e essas variáveis não estão mais disponíveis para o programa.
- As pilhas são utilizadas por compiladores para avaliar expressões aritméticas e gerar código de linguagem de máquina para processar as expressões.
- A técnica de implementar cada método de pilha como uma chamada a um método `List` é chamada de delegação — o método de pilha invocado delega a chamada ao método `List` apropriado.

Seção 22.6 Filas

- Uma fila é semelhante a uma fila de caixa em um supermercado — a primeira pessoa na fila é servida primeiro e os outros clientes entram na fila apenas no fim e esperam ser atendidos.
- Os nós da fila só são removidos a partir da cabeça da fila e só são inseridos na cauda. Por essa razão, uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (First-In, First-Out — FIFO).
- As operações de inserção e remoção são conhecidas como `enqueue` e `dequeue`.
- As filas têm muitas utilizações em sistemas de computador. A maioria dos computadores tem apenas um único processador, então somente um usuário por vez pode ser servido. As entradas para os outros aplicativos são colocadas em uma fila. A entrada na frente da fila é a próxima a receber o serviço. Cada entrada avança gradualmente para a frente da fila quando os aplicativos recebem o serviço.

Seção 22.7 Árvores

- Uma árvore é uma estrutura de dados bidimensional não linear. Os nós da árvore contêm dois ou mais links.
- Os nós de árvore contêm dois ou mais links. O nó raiz é o primeiro nó em uma árvore.
- Cada link no nó raiz referencia um filho. O filho esquerdo é o primeiro nó na subárvore esquerda e o filho direito é o primeiro nó na subárvore direita.
- Os filhos de um nó são chamados irmãos. Um nó sem filhos é chamado nó de folha.
- Em uma árvore de pesquisa binária sem valores duplicados, os valores em qualquer subárvore esquerda são menores que o valor no nó pai da subárvore e os valores em qualquer subárvore direita são maiores que o valor no nó pai da subárvore. Um nó pode ser inserido apenas como um nó de folha em uma árvore de pesquisa binária.
- Um percurso na ordem de uma árvore de pesquisa binária processa os valores de nó na ordem crescente.
- Em um percurso na pré-ordem, o valor em cada nó é processado quando o nó é percorrido. Então os valores na subárvore esquerda são processados e, em seguida, os valores na subárvore direita.
- Em um percurso na pós-ordem, o valor em cada nó é processado depois dos valores de seus filhos.
- A árvore de pesquisa binária facilita a eliminação de duplicatas. Quando a árvore é criada, as tentativas de inserir um valor duplicado são reconhecidas porque uma duplicata segue as mesmas decisões "siga para a esquerda" ou "siga para a direita" em cada comparação que o valor original fez. Portanto, a duplicata acaba sendo comparada com um nó contendo o mesmo valor. O valor duplicado pode ser descartado nesse ponto.
- Em uma árvore fortemente empacotada, cada nível contém aproximadamente duas vezes o número de elementos que o anterior. Então uma árvore de pesquisa binária fortemente empacotada com n elementos tem $\log_2 n$ níveis e, portanto, no máximo $\log_2 n$ comparações teriam de ser feitas para localizar uma correspondência ou para determinar que não existe nenhuma correspondência. Pesquisar em uma árvore de pesquisa binária (fortemente empacotada) de 1.000 elementos requer no máximo 10 comparações, pois $2^{10} > 1.000$. Pesquisar em uma árvore de pesquisa binária (fortemente empacotada) de 1.000.000 elementos requer no máximo 20 comparações, pois $2^{20} > 1.000.000$.

Terminologia

adicionar a operação de pilha (*push*), 704
alocação dinâmica de memória, 696
árvore binária, 695
árvore binária, classificação, 713
árvore de pesquisa binária, 709
árvore fortemente empacotada, 714
árvore fortemente equilibrada, 714
cabeça de uma fila, 695
classe autorreferencial, 695
delegar uma chamada de método, 706
eliminação de duplicatas, 713
enqueue, operação de fila, 707
estrutura de dados dinâmica, 695
estrutura de dados linear, 709
estrutura de dados, primeiro a entrar, primeiro a sair (*First-In, First-Out* — FIFO), 707
estrutura de dados último a entrar, primeiro a sair (*Last-In-First-Out* — LIFO), 704
fila, 695
filho direito, 709
filho esquerdo, 709
final de uma fila, 695
lista vinculada, 695
lista vinculada individualmente, 697
nó de folha, 709
nó em uma lista, 696
nó filho, 709
nó raiz, 709
nós irmãos, 709
operação para desenfileiramento de fila, 707
operação para enfileiramento de fila, 707
parte superior de uma pilha, 695
percurso em uma árvore binária em ordem de nível, 714
percurso na ordem, 709
percurso na pós-ordem, 710
percurso na pré-ordem, 710
pilha, 695
remover a operação de pilha (*pop*), 704
sequência, 709
spooler, 707
spooling de impressão, 707
subárvore direita, 709
subárvore esquerda, 709
vincular a outro nó (link), 696

Exercícios de autorrevisão

22.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Uma classe auto_____ é utilizada para formar estruturas de dados dinâmicas que podem crescer e encolher em tempo de execução.

b) Um(a) _____ é uma versão limitada de uma lista encadeada em que nós podem ser inseridos e excluídos somente a partir do início da lista.

c) Um método que não altera uma lista encadeada, mas simplesmente a examina para determinar se ela está vazia, é referido como um método _____.

d) Uma fila é referida como uma estrutura de dados _____ porque os primeiros nós inseridos são os primeiros nós removidos.

e) A referência ao próximo nó em uma lista vinculada é referida como _____.

f) Reivindicar automaticamente memória alocada dinamicamente em Java é chamado de _____.

g) Um(a) _____ é uma versão limitada de uma lista vinculada em que os nós podem ser inseridos apenas no fim da lista e excluídos apenas do início da lista.

h) Um(a) _____ é uma estrutura de dados bidimensional não linear que contém nós com dois ou mais links.

i) Uma pilha é referida como uma estrutura de dados _____ porque o último nó inserido é o primeiro nó removido.

j) Os nós de uma árvore _____ contém dois membros de link.

k) O primeiro nó de uma árvore é o nó de _____.

l) Cada link em um nó de árvore refere-se a um(a) _____ ou _____ desse nó.

m) Um nó de árvore que não tem filhos é chamado de nó _____.

n) Os três algoritmos de percorrer que mencionamos no texto para árvores de pesquisa binária são _____, _____ e _____.

22.2 Quais são as diferenças entre uma lista vinculada e uma pilha?

22.3 Quais são as diferenças entre uma pilha e uma fila?

22.4 Talvez um título mais apropriado para este capítulo fosse "Estruturas de dados reutilizáveis". Comente como cada uma das seguintes entidades ou conceitos contribuem para a capacidade de reutilização das estruturas de dados:

a) classes

b) herança

c) composição

22.5 Forneça manualmente os percursos na ordem, pré-ordem e pós-ordem da árvore de pesquisa binária da Figura 22.20

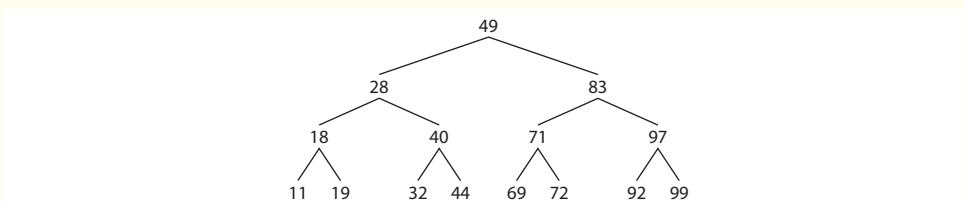


Figura 22.20 | Árvore de pesquisa binária com 15 nós.

Respostas dos exercícios de autorrevisão

22.1 a) referencial. b) pilha. c) predicado. d) primeiro a entrar, primeiro a sair (first in, first out — FIFO). e) link. f) coleta de lixo. g) fila. h) árvore i) último a entrar, primeiro a sair (LIFO). j) binário. k) raiz. l) filho ou subárvore. m) folha. n) na ordem, pré-ordem, pós-ordem.

22.2 É possível inserir um nó em qualquer lugar de uma lista vinculada e remover um nó de qualquer lugar de uma lista vinculada. Os nós em uma pilha podem ser inseridos somente na parte superior da pilha e removidos somente a partir da parte superior.

22.3 Uma fila é uma estrutura de dados FIFO que tem referências tanto para sua cabeça como para sua cauda, de modo que os nós podem ser inseridos na cauda e excluídos da cabeça. Uma pilha é uma estrutura de dados LIFO que tem uma única referência ao topo da pilha, no qual a inserção e a exclusão são realizadas.

22.4 a) As classes nos permitem instanciar quantos objetos de estrutura de dados de certo tipo (isto é, classe) quisermos.

b) A herança permite que uma subclasse reutilize as funcionalidades de uma superclasse. Os métodos públicos e protegidos de uma superclasse podem ser acessados por uma subclasse para eliminar a lógica duplicada.

c) A composição permite que uma classe reutilize o código armazenando uma referência a uma instância de outra classe em um campo. Os métodos públicos da instância podem ser chamados pelos métodos na classe que contém a referência.

22.5 O percurso na ordem é

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

O percurso na pré-ordem é

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

O percurso na pós-ordem é

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercícios

22.6 (***Concatenando listas***) Escreva um programa que concatene dois objetos de lista vinculada de caracteres. A classe `ListConcatenate` deve incluir um método `static concatenate` que aceite referências tanto para objetos de lista como para argumentos e concatene a segunda lista com a primeira lista.

22.7 (***Inserindo em uma lista ordenada***) Escreva um programa que insira 25 inteiros aleatórios de 0 a 100 na ordem em um objeto de lista vinculada. Para esse exercício, você precisará modificar a classe `List<T>` (Figura 22.3) para manter uma lista ordenada. Nomeie a nova versão da classe `SortedList`.

22.8 (***Mesclando listas ordenadas***) Modifique a classe `SortedList` do Exercício 22.7 para incluir um método `merge` que possa mesclar a `SortedList` que ele recebe como um argumento com a `SortedList` que chama o método. Escreva um aplicativo para testar o método `merge`.

22.9 (***Copiando uma lista de trás para frente***) Escreva um método `static reverseCopy` que recebe uma `List<T>` como um argumento e retorna uma cópia dessa `List<T>` com seus elementos invertidos. Teste esse método em um aplicativo.

22.10 (***Imprimindo uma frase na ordem inversa usando uma pilha***) Escreva um programa que insere uma linha de texto e utiliza uma pilha para exibir as palavras da linha em ordem reversa.

22.11 (***Testador de palíndromo***) Escreva um programa que utilize uma pilha para determinar se uma string é um palíndromo (isto é, a string é escrita identicamente de trás para frente). O programa deve ignorar espaços e pontuação.

22.12 (***Conversor de infixo para pós-fixo***) Pilhas são utilizadas por compiladores para ajudar no processo de avaliar expressões e gerar código de linguagem de máquina. Nesse e no próximo exercício, investigamos como os compiladores avaliam expressões aritméticas que consistem apenas de constantes, operadores e parênteses.

Os humanos geralmente escrevem expressões como `3 + 4` e `7 / 9` em que o operador (`+` ou `/` aqui) é escrito entre seus operandos — isso é chamado *notação infixa*. Os computadores "preferem" *notação pós-fixa*, na qual o operador é escrito à direita de seus dois operandos. As expressões infixas precedentes apareceriam na notação pós-fixa como `3 4 +` e `7 9 /`, respectivamente.

Para avaliar uma expressão infixa complexa, um compilador primeiro converteria a expressão em notação pós-fixa e avaliaria a versão. Cada um desses algoritmos requer apenas uma única passagem da esquerda para a direita pela expressão. Cada algoritmo utiliza um objeto pilha em suporte de sua operação, mas cada um utiliza a pilha para um propósito diferente.

Nesse exercício, você escreverá uma versão Java do algoritmo de conversão de infixo para pós-fixo. No próximo exercício, escreverá uma versão Java do algoritmo de avaliação da expressão pós-fixa. Em um exercício posterior, você descobrirá que o código que escrever nesse exercício pode ajudá-lo a implementar um compilador completo.

Escreva a classe `InfixToPostfixConverter` para converter uma expressão aritmética infixa comum (assuma que uma expressão válida foi inserida) com inteiros de único dígito como

```
(6 + 2) * 5 - 8 / 4
```

para uma expressão pós-fixa. A versão pós-fixa da expressão infixa precedente é (observe que nenhum parêntese é necessário)

```
6 2 + 5 * 8 4 / -
```

O programa deve ler a expressão no `StringBuffer infix` e utilizar uma das classes de pilha implementadas neste capítulo para ajudar a criar a expressão pós-fixa no `StringBuffer postfix`. O algoritmo para criar uma expressão pós-fixa é o seguinte:

a) Adicionar um parêntese esquerdo `'('` à pilha.

b) Acrescentar um parêntese direito `')'` ao final de `infix`.

c) Enquanto a pilha não estiver vazia, leia `infix` da esquerda para a direita e faça o seguinte:

Se o caractere atual em `infix` for um dígito, acrescente-o a `postfix`.

Se o caractere atual em `infix` for um parêntese esquerdo, adicione-o à pilha.

Se o caractere atual em `infix` for o operador:

Remova os operadores (se houver um) na parte superior da pilha enquanto eles tiverem precedência igual ou mais alta que o operador atual e acrescente os operadores removidos a `postfix`.

Adicione o caractere atual a `infix` na pilha.

Se o caractere atual em `infix` for um parêntese direito:

Remova operadores da parte superior da pilha e acrescente-os a `postfix` até que um parêntese esquerdo esteja na parte superior da pilha.

Remova (e descartar) o parêntese esquerdo da pilha.

As seguintes operações aritméticas são permitidas em uma expressão:

`+` adição
`-` subtração
`*` multiplicação
`/` divisão
`^` exponenciação
`%` resto

A pilha deve ser mantida com nós de pilha que cada um contém uma variável de instância e uma referência ao próximo nó de pilha. Alguns métodos que você pode querer fornecer são apresentados a seguir:

a) O método `convertToPostfix`, que converte a expressão infixa em notação pós-fixa.

b) O método `isOperator`, que determina se `c` é o operador.

c) O método `precedence`, que determina se a precedência do `operator1` (da expressão infixa) é menor, igual ou maior que a do `operator2` (da pilha). O método retorna `true` se `operator1` tiver precedência mais baixa que `operator2`. Caso contrário, `false` é retornado.

d) O método `peek` (que deve ser adicionado à classe de pilha), que retorna o valor superior da pilha sem estourar a pilha.

22.13 (***Avaliador de pós-fixo***) Escreva a classe `PostfixEvaluator` que avalia uma expressão *pós-fixa* como

```
6 2 + 5 * 8 4 / -
```

O programa deve ler uma expressão pós-fixa consistindo em dígitos e operadores em um `StringBuffer`. Utilizando versões modificadas dos métodos de pilha implementados anteriormente neste capítulo, o programa deve varrer a expressão e avaliá-la (supõe que ela seja válida). O algoritmo é como segue:

a) Acrescente um parêntese direito `')'` no fim da expressão pós-fixa. Quando o caractere do parêntese direito for encontrado, mais nenhum processamento é necessário.

b) Enquanto o caractere do parêntese direito não for encontrado, leia a expressão da esquerda para a direta.

Se o caractere atual for um dígito, faça o seguinte:

Adicione seu valor de inteiro à pilha (o valor de inteiro de um caractere de dígito é seu valor no conjunto de caracteres Unicode menos o valor `'0'` em Unicode).

Caso contrário, se o caractere atual for um *operador*:

Remova os dois elementos superiores da pilha para variáveis `x` e `y`.

Calcule `y` *operador* `x`.

Adicione o resultado do cálculo à pilha.

c) Quando o parêntese direito for encontrado na expressão, remova o valor da parte superior da pilha. Esse é o resultado da expressão pós-fixa.

[*Nota*: Em (b) acima (com base na expressão de exemplo no início deste exercício), se o operador for `'/'`, o topo da pilha é `4` e o próximo elemento na pilha é `40`, então remova `4` para `x`, remova `40` para `y`, avalie `40 / 4` e adicione o resultado, `10`, de volta à pilha. Essa nota também se aplica ao operador `'-'`.] As operações aritméticas permitidas em uma expressão são:

`+` adição
`-` subtração
`*` multiplicação
`/` divisão
`^` exponenciação
`%` resto

A pilha deve ser mantida com uma das classes de pilha introduzidas neste capítulo. Você pode querer fornecer os seguintes métodos:

a) O método `evaluatePostfixExpression`, que avalia a expressão pós-fixa.

b) O método `calculate`, que avalia a expressão `op1 operator op2`.

22.14 (***Modificação do avaliador de pós-fixo***) Modifique o programa avaliador de pós-fixo do Exercício 22.13 de modo que ele possa processar os operandos de inteiros maiores que 9.

22.15 (***Simulação de Supermercado***) Escreva um programa que simula uma fila de caixa em um supermercado. A fila é um objeto fila. Os clientes (isto é, os objetos cliente) chegam em intervalos aleatórios inteiros de 1 a 4 minutos. Além disso, cada cliente é atendido em intervalos aleatórios

inteiros de 1 a 4 minutos. Obviamente, as taxas precisam ser equilibradas. Se a taxa média de chegada for maior que a taxa média de atendimento, a fila crescerá infinitamente. Mesmo com taxas "equilibradas", a aleatoriedade ainda pode provocar filas longas. Execute a simulação de supermercado para um dia de 12 horas (720 minutos) utilizando o seguinte algoritmo:

a) Escolha um inteiro aleatório entre 1 e 4 para determinar o minuto em que o primeiro cliente chega.

b) Na hora da chegada do primeiro cliente, faça o seguinte:
Determine o tempo de atendimento do serviço ao cliente (inteiro aleatório de 1 a 4).
Comece atendendo o cliente.
Agenda a hora de chegada do próximo cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

c) Para cada minuto simulado do dia, considere o seguinte:
Se o próximo cliente chegar, prosseguir da seguinte maneira:
Expresse isso.
O enfileiramento do cliente.
Agende a hora de chegada do próximo cliente.
Se o atendimento do último cliente tiver sido concluído, faça o seguinte:
Expresse isso.
Desenfileire o próximo cliente a ser atendido.
Determine o tempo de atendimento do cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

Agora execute sua simulação para 720 minutos e responda a cada uma das seguintes perguntas:

a) Qual é o número máximo de clientes na fila a qualquer hora?

b) Qual é a espera mais longa que qualquer cliente experimenta?

c) O que acontece se o intervalo de chegada é alterado de 1 a 4 minutos para 1 a 3 minutos?

22.16 **(*Permitindo duplicatas em uma árvore binária*)** Modifique as figuras 22.17–22.18 para permitir que a árvore binária contenha valores duplicados.

22.17 **(*Processando uma árvore de pesquisa binária de Strings*)** Escreva um programa com base no programa das figuras 22.17–22.18 que insira uma linha do texto, tokenize (divida) a frase em palavras separadas, insira as palavras em uma árvore de pesquisa binária e imprima os percursos na ordem, pré-ordem e pós-ordem da árvore.

22.18 **(*Eliminação de duplicata*)** Neste capítulo, vimos que a eliminação de duplicata é simples e direta quando se cria uma árvore de pesquisa binária. Descreva como você realizaria a eliminação de duplicatas ao utilizar apenas um array unidimensional. Compare o desempenho da eliminação de duplicata baseada em array com o desempenho da eliminação de duplicata baseada na pesquisa de árvore binária.

22.19 **(*Profundidade de uma árvore binária*)** Modifique as figuras 22.17–22.18 para que a classe `Tree` forneça um método `getDepth` que determina quantos níveis estão na árvore. Teste o método em um aplicativo que insere 20 números inteiros aleatórios em uma `Tree`.

22.20 **(*Imprimir recursivamente uma lista de trás para frente*)** Modifique a classe `List<T>` da Figura 22.3 para incluir o método `printListBackward` que gera recursivamente saída dos itens em um objeto de lista encadeada na ordem inversa. Escreva um programa de teste que crie uma lista de inteiros e imprima a lista em ordem inversa.

22.21 **(*Pesquisar recursivamente uma lista*)** Modifique a classe `List<T>` da Figura 22.3 para incluir o método `search` que pesquisa recursivamente um valor especificado em um objeto de lista encadeada. O método deve retornar uma referência ao valor se ele for localizada; caso contrário, ele deve retornar `null`. Utilize seu método em um programa de teste que crie uma lista de inteiros. O programa deve solicitar ao usuário um valor para localizar na lista.

22.22 **(*Exclusão de árvore binária*)** Neste exercício, discutimos a exclusão de itens de árvores de pesquisa binária. O algoritmo de exclusão não é tão simples e direto quanto o algoritmo de inserção. Três casos são encontrados ao excluir-se um item — o item está contido em um nó de folha (isto é, não tem filhos), o item está contido em um nó que tem um filho ou está em um nó que tem dois filhos.
Se o item a ser excluído está contido em um nó de folha, o nó é excluído e a referência no nó pai é configurada como nulo.
Se o item a ser excluído está contido em um nó com um filho, a referência no nó pai é configurada para referenciar o nó filho e o nó contendo o item de dados é excluído. Isso faz com que o nó filho tome o lugar do nó excluído na árvore.
O último caso é o mais difícil. Quando um nó com dois filhos é excluído, outro nó na árvore deve tomar seu lugar. Entretanto, a referência no nó pai simplesmente não pode ser atribuída para referenciar um dos filhos do nó a ser excluído. Na maioria dos casos, a árvore de pesquisa binária resultante não incorporaria seguinte característica das árvores de pesquisa binária (sem valores duplicados): *Os valores em qualquer subárvore esquerda são menores que o valor no nó pai e os valores em qualquer subárvore direita são maiores que o valor no nó pai.*
Qual é o nó utilizado como um *nó substituto* para manter essa característica? É o nó contendo o maior valor na árvore menor que o valor no nó que está sendo excluído, ou o nó contendo o menor valor na árvore maior que o valor no nó que está sendo excluído? Vamos considerar o nó com o menor valor. Em uma árvore de pesquisa binária, o valor maior que um valor do pai encontra-se na subárvore esquerda do nó pai e seguramente estará contido no nó mais à direita da subárvore. Esse nó é encontrado percorrendo a subárvore esquerda pela direita até que a referência ao filho direito do nó atual seja nula. Agora estamos referenciando o nó substituto que é um nó de folha ou um nó com um filho à sua esquerda. Se o nó substituto for um nó de folha, os passos para realizar a exclusão são os seguintes:

a) Armazene a referência ao nó a ser excluído em uma variável de referência temporária.

b) Configure a referência no pai do nó sendo excluído para referenciar o nó substituto.

c) Configure a referência no pai do nó substituto como `null`.

d) Configure a referência como a subárvore direita no nó substituto para referenciar a subárvore direita do nó a ser excluído.

e) Configure a referência como a subárvore esquerda no nó substituto para referenciar a subárvore esquerda do nó a ser excluído.

Os passos de exclusão para um nó substituto com um filho esquerdo são semelhantes àqueles para um nó substituto sem filhos, mas o algoritmo também deve mover o filho para a posição do nó substituto na árvore. Se o nó substituto for um nó com um filho esquerdo, os passos a realizar a exclusão são como segue:

a) Armazene a referência ao nó a ser excluído em uma variável de referência temporária.

b) Configure a referência no pai do nó sendo excluído para referenciar o nó substituto.

c) Configure a referência no pai do nó substituto para referenciar o filho esquerdo do nó substituto.

d) Configure a referência como a subárvore direita no nó substituto para referenciar a subárvore direita do nó a ser excluído.

e) Configure a referência como a subárvore esquerda no nó substituto para referenciar a subárvore esquerda do nó a ser excluído.

Escreva o método `deleteNode`, que aceita como seu argumento o valor a ser excluído. O método `deleteNode` deve localizar na árvore o nó que contém o valor a ser excluído e utilizar os algoritmos discutidos aqui para excluir o nó. Se o valor não for localizado na árvore, o método deve exibir uma mensagem informando isso. Modifique o programa das figuras 22.17–22.18 para utilizar esse método. Depois de excluir um item, chame os métodos `inorderTraversal`, `preorderTraversal` e `postorderTraversal` para confirmar que a operação de exclusão foi realizada corretamente.

22.23 (***Pesquisa de árvore binária***) Modifique a classe `Tree` da Figura 22.17 para incluir o método `contains`, que tenta localizar um valor especificado em um objeto de árvore da pesquisa binária. O método deve aceitar como um argumento uma chave de pesquisa a ser localizada. Se o nó contendo a chave de pesquisa for localizado, o método deve retornar uma referência aos dados desse nó; caso contrário, deve retornar `null`.

22.24 (***Travessia na ordem de nível de árvore binária***) O programa das figuras 22.17–22.18 ilustrou os três métodos recursivos de atravessar uma árvore binária — travessias na ordem, pré-ordem e pós-ordem. Esse exercício apresenta o percurso *na ordem de nível* de uma árvore binária, em que os valores de nó são impressos nível por nível, iniciando no nível do nó raiz. Os nós em cada nível são impressos da esquerda para a direta. O percurso na ordem de nível não é um algoritmo recursivo. Ela utiliza um objeto fila para controlar a saída dos nós. O algoritmo é como segue:

a) Insira o nó raiz na fila.

b) Enquanto houver nós esquerdos na fila, faça o seguinte:

```
Obtenha o próximo nó na fila.
Imprima o valor do nó.
Se a referência ao filho esquerdo do nó não for nula:
        Insira o nó filho esquerdo na fila.
Se a referência ao filho direito do nó não é nula:
        Insira o nó filho direito na fila.
```

Escreva o método `levelOrder` para realizar um percurso na ordem de nível de um objeto de árvore binária. Modifique o programa das figuras 22.17–22.18 para utilizar esse método. [*Nota*: Você também precisará utilizar os métodos de processamento de fila da Figura 22.13 nesse programa.

22.25 (***Imprimindo árvores***) Modifique a classe `Tree` da Figura 22.17 para incluir um método `outputTree` recursivo para exibir um objeto de árvore binária. O método deve gerar saída da árvore linha por linha com o topo da árvore na parte esquerda da tela e a parte inferior da árvore em direção à parte direita da tela. Cada linha é enviada para a saída verticalmente. Por exemplo, a árvore binária ilustrada na Figura 22.20 é enviada para a saída como mostrado na Figura 22.21.

```
            99
        97
            92
    83
            72
        71
            69
49
            44
        40
            32
    28
            19
        18
            11
```

Figura 22.21 | Saída de exemplo do método recursivo `outputTree`.

O nó mais à direita da folha aparece na parte superior da saída na coluna mais à direita e o nó raiz aparece à esquerda da saída. Cada coluna inicia cinco espaços à direita da coluna precedente. O método `outputTree` deve receber um argumento `totalSpaces` para representar o número de espaços que precedem o valor a ser enviado para a saída. (Essa variável deve iniciar em zero de modo que o nó raiz seja enviado para a saída à esquerda da tela.) O método utiliza uma travessia na ordem modificada para dar a saída à árvore — ele inicia no nó mais à direita na árvore e segue para a esquerda. O algoritmo é como segue:

```
Enquanto a referência ao nó atual não for nula, faça o seguinte:
Chame recursivamente outputTree com a subárvore direita do nó atual e totalSpaces + .
Utilize uma estrutura for para contar de 1 a totalSpaces e envie os espaços para saída.
Envie para a saída o valor no nó atual.
Configure a referência ao nó atual para referenciar a subárvore esquerda do nó atual.
Incremente totalSpaces por 5.
```

22.26 (***Inserção/Exclusão em qualquer lugar de uma lista vinculada***) Nossa classe de lista vinculada permitiu inserções e exclusões no início e no fim da lista vinculada. Essas capacidades foram convenientes para nós quando utilizamos herança ou composição para produzir uma classe de pilha e uma classe de fila com uma quantidade mínima de código simplesmente reutilizando a classe de lista. As listas vinculadas são normalmente mais gerais que aquelas que fornecemos. Modifique a classe da lista vinculada que desenvolvemos neste capítulo para tratar inserções e exclusões em qualquer lugar da lista.

22.27 (***Listas e filas sem referências de fim***) Nossa implementação de uma lista vinculada (Figura 22.3) utilizou tanto `firstNode` como `lastNode`. O `lastNode` foi útil para os métodos `insertAtBack` e `removeFromBack` da classe `List`. O método `insertAtBack` equivale ao método `enqueue` da classe `Queue`.
Reescreva a classe `List` de modo que ela não utilize um `lastNode`. Portanto, quaisquer operações no fim de uma lista devem começar pesquisando no início da lista. Isso afeta nossa implementação da classe `Queue` (Figura 22.13)?

22.28 (***Desempenho da classificação e da pesquisa de árvore binária***) Um problema com a classificação de árvore binária é que a ordem em que os dados são inseridos afeta a forma da árvore — para a mesma coleção de dados, ordens diferentes podem produzir árvores binárias de formas significativamente diferentes. O desempenho dos algoritmos de classificação e pesquisa de árvore binária é sensível à forma da árvore binária. Que forma teria uma árvore binária se seus dados fossem inseridos na ordem crescente? na ordem decrescente? Que forma a árvore deveria ter para alcançar desempenho máximo de pesquisa?

22.29 (***Listas indexadas***) Como apresentado no texto, as listas vinculadas devem ser pesquisadas sequencialmente. Para listas grandes, isso pode resultar em desempenho pobre. Uma técnica comum para aprimorar o desempenho de pesquisa de lista é criar e manter um índice para a lista. Um índice é um conjunto de referências para lugares-chave na lista. Por exemplo, um aplicativo que pesquisa uma lista grande de nomes pode aprimorar seu desempenho criando um índice com 26 entradas — uma para cada letra do alfabeto. Uma operação de pesquisa de um sobrenome iniciando com 'Y' iria primeiro pesquisar o índice para determinar onde as entradas 'Y' iniciaram e, então, "saltaria" na lista nesse ponto e pesquisaria linearmente até que o nome desejado fosse localizado. Isso seria muito mais rápido que pesquisar a lista vinculada desde o início. Utilize a classe `List` da Figura 22.3 como a base de uma classe `IndexedList`.
Escreva um programa que demonstre a operação de listas indexadas. Certifique-se de incluir os métodos `insertInIndexedList`, `searchIndexedList` e `deleteFromIndexedList`.

22.30 (***Classe de pilha que herda de uma classe `List`***) Na Seção 22.5, criamos uma classe de pilha da classe `List` com a herança (Figura 22.10) e a composição (Figura 22.12). Na Seção 22.6, criamos uma classe queue a partir da classe `List` com composição (Figura 22.13). Crie uma classe queue herdando da classe `List`. Quais as diferenças entre essa classe e aquela criada com a composição?

Seção especial: construindo seu próprio compilador

Nos exercícios 7.35–7.37, introduzimos a Simpletron Machine Language (SML) e implementamos um simulador de computador Simpletron para executar programas SML. Nos exercícios 22.31–22.35, construímos um compilador que converte programas escritos em uma linguagem de programação de alto nível em SML. Esta seção "amarra" o processo de programação inteiro. Você escreverá programas nessa nova linguagem de alto nível, compilará esses programas no compilador que construir e os executará no simulador construído no Exercício 7.36. Você deve se esforçar o máximo possível para implementar seu compilador de uma maneira orientada a objetos. [*Nota*: Por causa do tamanho das descrições dos exercícios 22.31–22.35, nós as postamos em um documento PDF em inglês localizado em `www.deitel.com/books/jhtp8/`.]

23

Aja com prudência, a ocasião é o fator mais importante entre todos.
— **Hesíodo**

A pintura é apenas uma ponte ligando a mente do pintor com a do observador.
— **Eugene Delacroix**

A direção em que a educação conduz um homem determinará sua vida futura.
— **Platão**

Applets e Java Web Start

Objetivos

Neste capítulo, você aprenderá:

- O que são applets e como eles são utilizados em páginas Web.
- A observar alguns das fantásticas capacidades do Java por meio de applets de demonstração do JDK.
- A escrever applets simples.
- A escrever um documento XHTML (Extensible HyperText Markup Language) simples para carregar um applet em um contêiner de applets e executar o applet.
- Os cinco métodos chamados automaticamente por um contêiner de applets durante o ciclo de vida de um applet.
- O que é o modelo de segurança de caixa de areia e como ele permite executar de uma maneira segura código baixado.
- O que é o Java Web Start e como utilizá-lo para fazer o download, instalar e executar applets fora do navegador Web.

Sumário

23.1 Introdução
23.2 Applets de exemplo fornecidas com o JDK
23.3 Applet Java simples: desenhando uma string
23.3.1 Executando `WelcomeApplet` no appletviewer
23.3.2 Executando um applet em um navegador da Web
23.4 Métodos de ciclo de vida de applet
23.5 Inicializando uma variável de instância com o método `init`
23.6 Modelo de segurança da caixa de areia
23.7 Java Web Start e o Java Network Launch Protocol (JNLP)
23.7.1 Empacotando o applet `DrawTest` para uso com Java Web Start
23.7.2 Documento JNLP para o applet `DrawTest`
23.8 Conclusão

Resumo | Terminologia | Exercício de autorrevisão | Respostas do exercício de autorrevisão | Exercícios

23.1 Introdução

[*Nota*: Este capítulo é intencionalmente pequeno e simples para leitores que desejam estudar applets depois de ler apenas os primeiros capítulos do livro. Apresentamos applets mais complexos no Capítulo 24, Multimídia: applets e aplicativos, e no Capítulo 27, Redes. Além disso, os exemplos neste capítulo exigem um pouco de conhecimento de XHTML para criar uma página Web que carrega um applet. Com cada exemplo fornecemos documentos XHTML de exemplo que você pode modificar para seus próprios objetivos. Para aprender mais sobre a XHTML, acesse o link XMTML Tutorials no nosso XHTML Resource Center (`www.deitel.com/xhtml/`).]

Este capítulo introduz **applets** — programas Java que são tipicamente incorporados **a documentos XHTML (Extensible HyperText Markup Language)** — também chamados páginas Web. Quando um navegador Web compatível com Java carrega uma página Web que contém um applet, o applet é baixado no navegador e executa.

Contêineres de applet

O aplicativo no qual um applet executa é conhecido como o **contêiner de applets**. É responsabilidade do contêiner de applets carregar a(s) classe(s) do applet, criar uma instância do applet e gerenciar seu ciclo de vida (que discutimos mais detalhadamente na Seção 23.4). O Java Development Kit (JDK) inclui um chamado **`appletviewer`** para testar applets à medida que você os desenvolve e antes de incorporá-los a páginas Web. Demonstramos applets que utilizam tanto o `appletviewer` como navegadores Web, que executam applets Java via o **Plug-in Java**. Alguns navegadores não vêm com o plug-in por padrão. Você pode visitar `java.com` para verificar se seu navegador está pronto para executar applets Java. Se não estiverem, clique no botão Free Java Download para instalar o Java no seu navegador. Muitos navegadores populares são utilizados. Testamos nossos applets no Firefox 3 da Mozilla, Internet Explorer da Microsoft 7, Chrome do Google, Opera 9 e Safari da Apple 3.

Java Web Start e o Java Network Launch Protocol (JNLP)

Este capítulo termina com uma introdução ao Java Web Start e ao Java Network Launch Protocol (JNLP). Juntos, eles permitem empacotar seus applets e aplicativos para que possam ser instalados na área de trabalho do usuário. Como veremos no Capítulo 24, o Java Web Start também permite dar ao usuário controle sobre se um applet ou aplicativo baixado da Web pode ter acesso limitado aos recursos no sistema de arquivos local. Por exemplo, se você criar em Java um programa editor de textos que pode ser baixado, é provável que os usuários talvez queiram armazenar os documentos nos seus próprios computadores.

23.2 Applets de exemplo fornecidas com o JDK

Antes de discutirmos nossos applets, vamos considerar vários applets de exemplo fornecidos no JDK. Cada applet de exemplo vem com o código-fonte. Alguns programadores acham interessante ler esse código-fonte para aprender os fantásticos recursos Java.

Os programas de demonstração fornecidos no JDK estão localizados em um diretório chamado `demo`. Para o Windows, a localização padrão do diretório `demo` do JDK 6.0 é

```
C:\Program Files\Java\jdk1.6.0_##\demo
```

onde `_##` representa o número da atualização do JDK. No UNIX/Linux, a localização padrão é o diretório em que você instala o JDK seguido por `jdk1.6.0_##/demo` — por exemplo,

```
/usr/local/jdk1.6.0_##/demo
```

Para outras plataformas, haverá uma estrutura semelhante de diretório (ou pasta). Este capítulo supõe que o JDK esteja instalado em `C:\Program Files\Java\jdk1.6.0_11` no Windows ou no diretório inicial em `~/jdk1.6.0_11` no UNIX/Linux. Talvez seja necessário atualizar as localizações especificadas aqui para que elas correspondam ao seu diretório de instalação e unidade de disco selecionados, ou uma versão diferente do JDK.

Se estiver utilizando uma ferramenta de desenvolvimento Java que não é distribuída com as demos do Sun Java, baixe o JDK atual em `java.sun.com/javase/downloads/index.jsp`. Usuários do Mac OS X devem acessar `developer.apple.com/java` para obter informações sobre o Java no Mac, ou utilizar um software de virtualização para executar as versões do Windows ou Linux do Java em uma máquina virtual.

Visão geral dos applets de demonstração

Abra uma janela de comando e utilize o comando `cd` para mudar para o diretório `demo` do JDK. O diretório `demo` contém vários subdiretórios. Você pode listá-los emitindo o comando `dir` no Windows ou o comando `ls` no UNIX/Linux/Mac OS X. Discutimos programas de exemplo nos subdiretórios `applets` e `jfc`. O diretório `applets` contém applets de demonstração. O diretório `jfc` (Java Foundation Classes) contém applets e aplicativos que demonstram os recursos gráficos e as capacidades GUI do Java.

Mude para o diretório `applets` e liste seu conteúdo para ver os nomes dos diretórios para os applets de demonstração. A Figura 23.1 fornece uma breve descrição de cada um. Se seu navegador suportar Java, você poderá testar um applet abrindo o respectivo documento XHTML no diretório do applet. Demonstraremos três desses applets utilizando o comando `appletviewer` em uma janela de comando.

Exemplo	Descrição
`Animator`	Realiza uma de quatro animações separadas.
`ArcTest`	Demonstra como desenhar arcos. Você pode interagir com o applet para alterar atributos do arco que é exibido.
`BarChart`	Desenha um gráfico de barras simples.
`Blink`	Exibe texto intermitente em diferentes cores.
`CardTest`	Demonstra vários componentes GUI e layouts.
`Clock`	Desenha um relógio com ponteiros giratórios, a data e a hora atuais. O relógio é atualizado uma vez por segundo.
`DitherTest`	Demonstra desenhos com uma técnica gráfica conhecida como pontilhamento que permite uma transformação gradual de uma cor para outra.
`DrawTest`	Permite ao usuário arrastar o mouse para desenhar linhas e pontos em diferentes cores.
`Fractal`	Desenha um fractal. Os fractais em geral requerem cálculos complexos para determinar como eles são exibidos. Discutimos fractais na Seção 18.8.
`GraphicsTest`	Desenha formas para ilustrar as capacidades gráficas.
`GraphLayout`	Desenha um gráfico que consiste em muitos nós (representados como retângulos) conectados por linhas. Arraste um nó para ver os outros nós no gráfico se ajustarem na tela e para demonstrar interações gráficas complexas.
`ImageMap`	Demonstra uma imagem com pontos ativos. Posicionar o ponteiro do mouse sobre certas áreas da imagem destaca essa área e uma mensagem é exibida no canto esquerdo inferior da janela do contêiner de applets. Posicione sobre a boca na imagem para ouvir o applet dizer "hi".
`JumpingBox`	Move um retângulo aleatoriamente pela tela. Tente pegá-lo clicando nele com o mouse!
`MoleculeViewer`	Apresenta uma visualização tridimensional de várias moléculas químicas. Arraste o mouse para ver a molécula de ângulos diferentes.
`NervousText`	Desenha texto que se movimenta pela applet.
`SimpleGraph`	Desenha uma curva complexa.
`SortDemo`	Compara três técnicas de classificação. A classificação (descrita no Capítulo 19) organiza as informações em ordem — como palavras em ordem alfabética. Ao executar esse exemplo com o `appletviewer`, três janelas aparecem. Ao executá-lo em um navegador, as três demos aparecem lado a lado. Clique em cada demo para iniciar a classificação. Observe que todas as classificações operam em diferentes velocidades.

Exemplo	Descrição
`SpreadSheet`	Demonstra uma planilha simples de linhas e colunas.
`TicTacToe`	Permite ao usuário jogar o jogo da velha contra o computador.
`WireFrame`	Desenha uma forma tridimensional como um aramado. Arraste o mouse para ver a forma de ângulos diferentes.

Figura 23.1 | Exemplos do diretório `applets`.

Applet `TicTacToe`

Esse applet de demonstração `TicTacToe` permite jogar o jogo da velha contra o computador. Mude para o subdiretório `TicTacToe`, no qual você localizará o arquivo `example1.html` que carrega o applet. Na janela de comando, digite o comando

```
appletviewer example1.html
```

e pressione *Enter*. Isso executa o contêiner de applets `appletviewer`, o qual carrega o documento XHTML `example1.html` especificado como seu argumento de linha de comando. O `appletviewer` determina no documento qual applet será carregado e executado. A Figura 23.2 mostra as várias capturas de tela do jogo Tic-Tac-Toe (jogo da velha) com esse applet. Você pode abrir o documento XHTML no navegador para executar o applet nele.

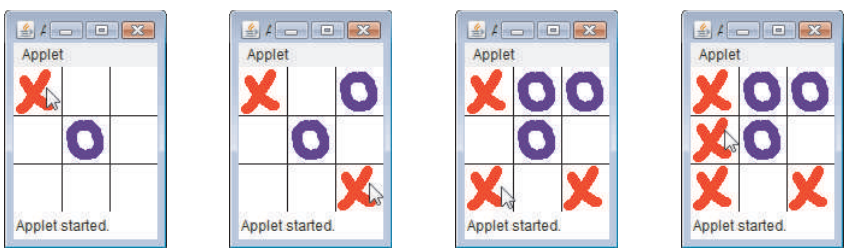


Figura 23.2 | Execução de exemplo do applet `TicTacToe`.

Você é o jogador **X**. Para interagir com o applet, aponte o mouse para o quadrado em que você quer colocar um **X** e clique com o botão do mouse. O applet emite um som e coloca um **X** no quadrado, se ele estiver aberto. Se o quadrado estiver ocupado, esse é um movimento inválido e o applet emite um som diferente indicando que você não fez o movimento especificado. Depois de fazer um movimento válido, o applet responde fazendo seu próprio movimento.

Para reproduzir novamente, clique no menu **Applet** do `appletviewer` e selecione o item **Reload** no menu (Figura 23.3), ou clique no applet novamente quando o jogo terminar. Para terminar o `appletviewer`, clique no menu **Applet** do `appletviewer` e selecione o item de menu **Quit**.

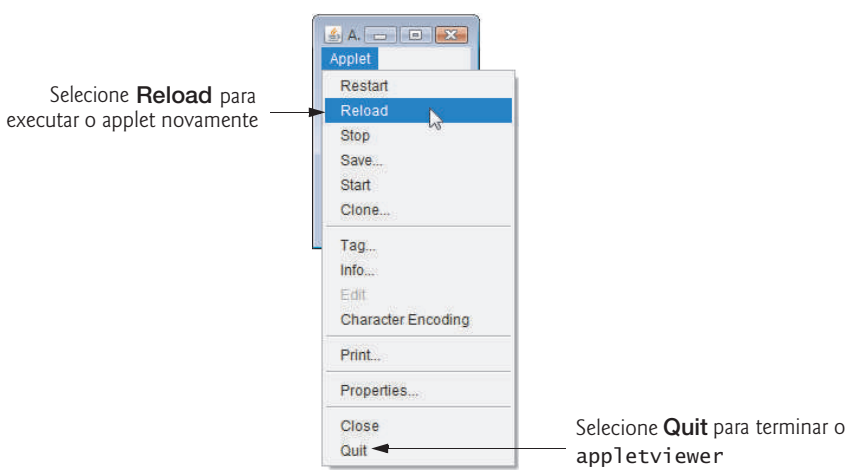


Figura 23.3 | Menu **Applet** no `appletviewer`.

Applet `DrawTest`

O applet `DrawTest` permite desenhar linhas e pontos em cores diferentes. Na janela de comando, mude para o diretório `applets` e então para subdiretório `DrawTest`. Você pode mover-se para cima na árvore de diretórios incrementalmente em direção a `demo` emitindo o comando "`cd ..`" na janela de comando. O diretório `DrawTest` contém o documento `example1.html` que é utilizado para executar o applet. Na janela de comando, digite o comando

```
appletviewer example1.html
```

e pressione *Enter*. O `appletviewer` carrega `example1.html`, determina a partir do documento o applet que deve carregar e o executa. A Figura 23.4 mostra uma captura de tela depois que algumas linhas e pontos foram desenhados.

Por padrão, o applet permite desenhar linhas pretas arrastando o mouse pelo applet. Ao arrastar o mouse, observe que o ponto de partida da linha sempre permanece no mesmo lugar e sua extremidade segue o ponteiro do mouse na applet. A linha não será permanente até você liberar o botão do mouse.

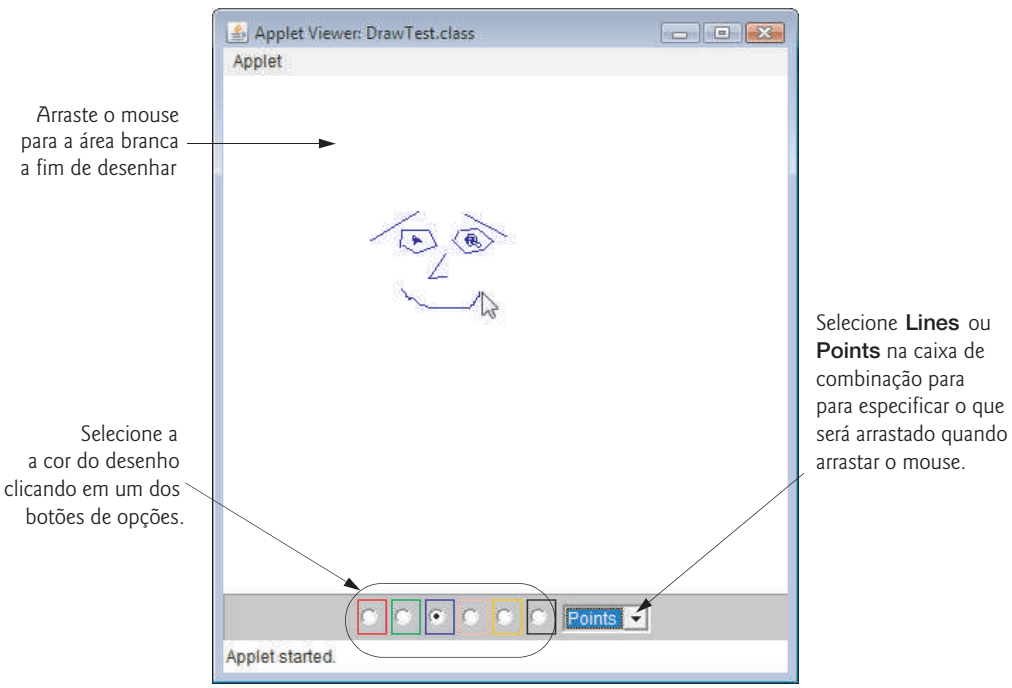


Figura 23.4 | Execução de exemplo do applet `DrawTest`.

Selecione uma cor clicando em um dos botões de opção na parte inferior do applet. Você pode selecionar vermelho, verde, azul, cor-de-rosa, alaranjado e preto. Altere a forma para desenhar de **Lines** para **Points** selecionando **Points** na caixa de combinação. Para iniciar um novo desenho, selecione **Reload** do menu **Applet** do `appletviewer`.

Applet `Jav2D`

O applet `Java2D` demonstra muitos recursos da Java 2D API (que introduzimos no Capítulo 15). Mude para o diretório `jfc` no diretório `demo` do JDK, e depois para o diretório `Java2D`. Na janela de comando, digite o comando

```
appletviewer Java2Demo.html
```

e pressione *Enter*. O `appletviewer` carrega `Java2Demo.html`, determina a partir do documento o applet que deve carregar e o executa. A Figura 23.5 mostra uma captura de tela de uma das muitas demonstrações desse applet das capacidades gráficas bidimensionais do Java.

Na parte superior do applet estão guias que parecem pastas suspensas em um armário de arquivos. Essa demonstração fornece 12 guias, cada uma delas demonstrando os recursos da Java 2D API. Para mudar para uma parte diferente da demo, simplesmente clique em uma guia diferente. Além disso, tente alterar as opções no canto superior direito do applet. Algumas delas afetam a velocidade com que o applet desenha as imagens gráficas. Por exemplo, clique na caixa de seleção à esquerda da palavra **Anti Aliasing** para desativar a suavização, ou o antialiasing (uma técnica gráfica para produzir imagens mais suaves na tela em que as bordas da imagem são desfocadas). Formas que não são suavizadas (antialiased) são menos complexas de desenhar. Consequentemente, quando o recurso antialiasing está desativado, a velocidade de animação aumenta para as formas animadas na parte inferior da demo (Figura 23.5).

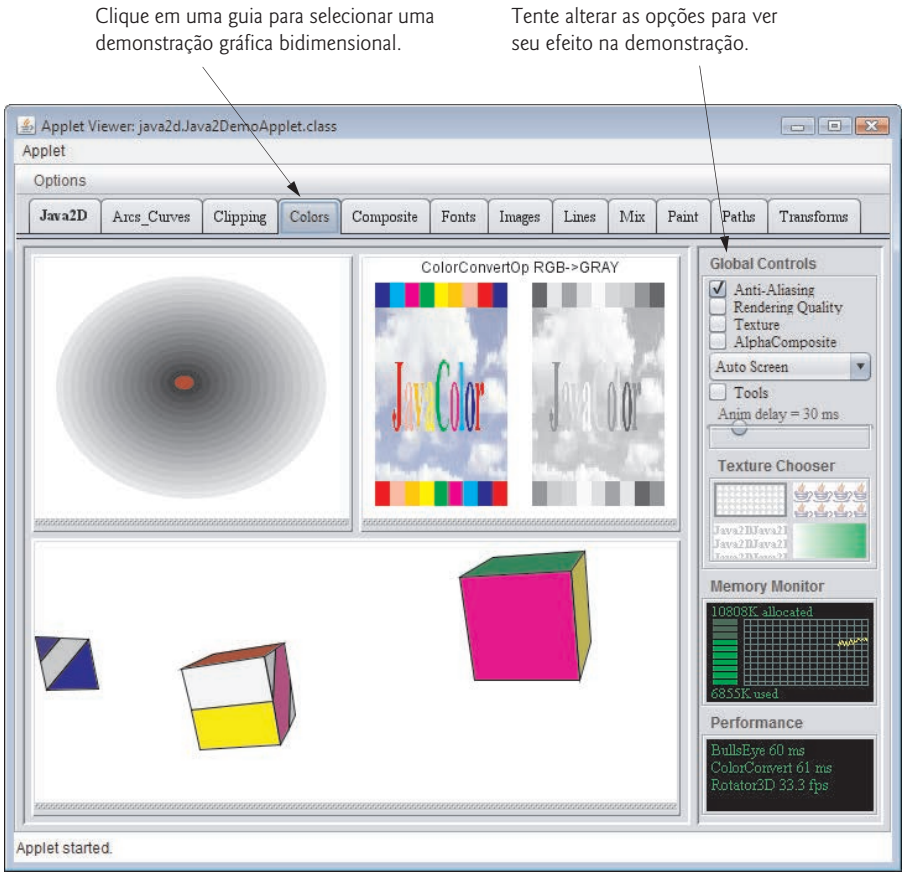


Figura 23.5 | Execução de exemplo do applet `Java2D`.

23.3 Applet Java simples: desenhando uma string

Cada applet Java é uma interface gráfica com o usuário em que você pode colocar componentes GUI utilizando as técnicas apresentadas no Capítulo 14 ou desenhar utilizando as técnicas demonstradas no Capítulo 15. Neste capítulo, demonstraremos como desenhar em um applet. Os exemplos nos capítulos 24 e 27 demonstram como criar uma interface gráfica com o usuário do applet.

Agora vamos criar nosso próprio applet. Iniciamos com um applet simples (Figura 23.6) que desenha `"Welcome to Java Programming!"` no applet. Mostramos esse applet em execução em dois contêineres de applet — o `appletviewer` e o navegador Web Mozilla Firefox. No fim desta seção, você aprenderá a executar o applet em um navegador da Web.

```
1  // Figura 23.6: WelcomeApplet.java
2  // Applet que desenha uma String.
3  import TML.awt.Graphics;   // o programa utiliza a classe Graphics
4  import javax.swing.JApplet; // o programa utiliza a classe JApplet
5
6  public class WelcomeApplet extends JApplet
7  {
8     // desenha texto sobre o fundo do applet
9     public void paint( Graphics g )
10    {
11       // chama a versão da superclasse do método paint
12       super.paint( g );
13
14       // desenha uma String nas coordenadas x 25 e y 25
15       g.drawString( "Welcome to Java Programming!", 25, 25 );
16    } // fim do método paint
17 } // fim da classe WelcomeApplet
```

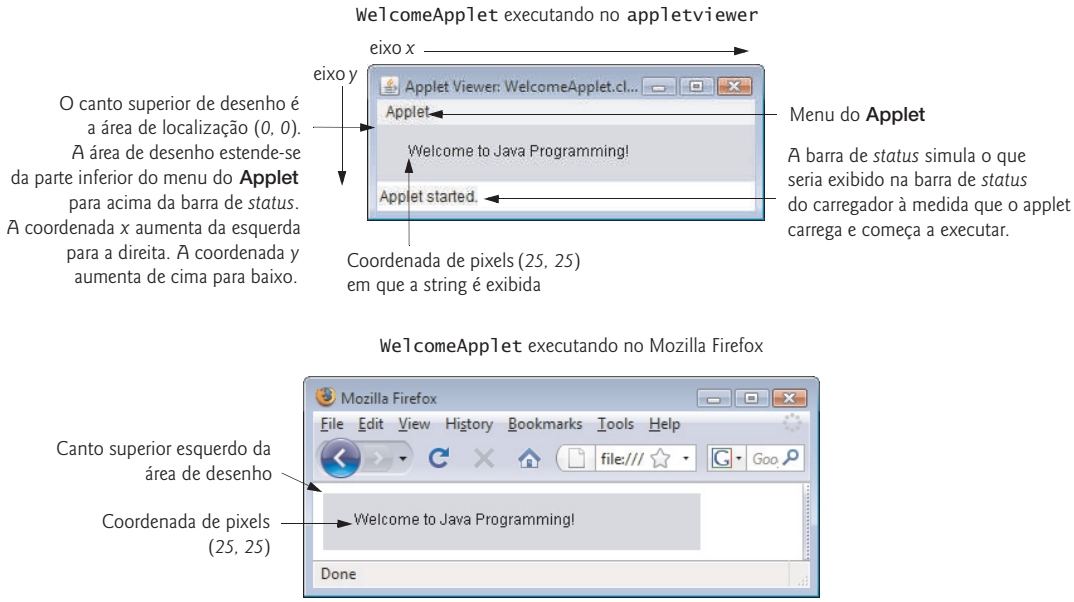


Figura 23.6 | Applet que desenha uma `String`.

Criando a classe Applet

A linha 3 importa a classe `Graphics` para permitir ao applet desenhar imagens gráficas como linhas, retângulos, ovais e strings de caracteres. A classe **`JApplet`** (importada na linha 4) do pacote `javax.swing` é utilizada para criar applets. Como com aplicativos, cada applet Java contém pelo menos uma declaração de classe `public`. Um contêiner de applets só pode criar objetos de classes que são `public` e `JApplet` estendidas (ou sua superclasse `Applet`). Por essa razão, a classe `WelcomeApplet` (linhas 6–17) estende a classe `JApplet`.

Um contêiner de applets espera que cada applet Java tenha os métodos **`init`**, **`start`**, **`paint`**, **`stop`** e **`destroy`**, cada um dos quais é declarado na classe `JApplet`. Cada nova classe de applet que você cria herda as implementações padrão desses métodos da classe `JApplet`. Esses métodos podem ser sobrescritos (redefinidos) para realizar tarefas específicas ao seu applet. A Seção 23.4 discute cada um desses métodos em mais detalhes.

Quando um contêiner de applets carrega a classe `WelcomeApplet`, o contêiner cria um objeto `WelcomeApplet`, então chama seus métodos `init`, `start` e `paint` na sequência. Se você não declarar esses métodos no seu applet, o contêiner de applets chamará as versões herdadas. Os métodos `init` e `start` da superclasse têm corpos vazios, portanto eles não realizam nenhuma tarefa. O método `paint` da superclasse não desenha nada no applet.

Talvez você pergunte por que é necessário herdar os métodos `init`, `start` e `paint` se suas implementações padrão não realizam tarefas. Alguns applets não utilizam todos esses três métodos. Entretanto, o contêiner de applets não sabe disso. Portanto, ele espera que cada applet tenha esses métodos, assim pode fornecer uma sequência inicial consistente. Isso é semelhante ao fato de os aplicativos sempre começarem a executar com `main`. Herdar as versões "padrão" desses métodos garante que o contêiner de applets pode executar cada applet uniformemente. Além disso, herdar implementações padrão desses métodos permite que o programador se concentre apenas na definição dos métodos requeridos por um applet particular.

Sobrescrevendo o método `paint` para desenhar

Para permitir que nosso applet desenhe, a classe `WelcomeApplet` sobrescreve o método `paint` (linhas 9–16) colocando instruções no corpo do `paint` que desenha uma mensagem na tela. O método `paint` recebe um parâmetro do tipo `Graphics` (chamado `g` por convenção), que é utilizado para desenhar imagens gráficas no applet. Você não chama o método `paint` explicitamente em um applet. Em vez disso, o contêiner de applets chama `paint` para dizer ao applet quando desenhar e o contêiner de applet é responsável por passar um objeto de `Graphics` como um argumento.

A linha 12 chama a versão da superclasse do método `paint` que foi herdado de `JApplet`. Essa deve ser a primeira instrução no método `paint` de cada applet. Omiti-lo pode causar erros sutis de desenho em applets que combinam componentes de desenho e GUIs.

A linha 15 utiliza o método `Graphics` `drawString` para desenhar `Welcome to Java Programming!` No applet. O método recebe como argumentos a `String` a desenhar e as coordenadas *x* e y em que o canto inferior esquerdo da `String` deve aparecer na área de desenho. Quando a linha 15 executa, ela desenha a `String` sobre o applet nas coordenadas `25` e `25`.

23.3.1 Executando `WelcomeApplet` no `appletviewer`

Depois de criar a classe `WelcomeApplet` e salvá-la no arquivo `WelcomeApplet.java`, abra uma janela de comando, mude para o diretório em que você salvou a declaração da classe de applet e compile a classe `WelcomeApplet`.

Lembre-se de que applets são incorporados a páginas Web para execução em um contêiner de applets (`appletviewer` ou um navegador). Para poder executar o applet, antes você deve criar um documento XHTML que especifica qual applet executar do contêiner de applets. Em geral, um documento XHTML termina com uma extensão de arquivo **`.html`** ou **`.htm`**. A Figura 23.7 mostra um documento XHTML simples — `WelcomeApplet.html` — que carrega o applet definido na Figura 23.6 em um contêiner de applets.

A maioria dos elementos XHTML é delimitada por **tags** — por exemplo, as linhas 1 e 6 delimitam o início e o fim do documento XHTML, respectivamente. Cada tag é colocado entre colchetes angulares (< e >). As linhas 2–5 especificam o elemento do **elemento `body`** do documento — isso representa os elementos que serão exibidos na página Web. As linhas especificam um **elemento `applet`** que instrui o contêiner de applets a carregar um applet específico e define o tamanho de sua área de exibição (sua largura e altura em pixels) no contêiner de applets.

```
1  <html>
2  <body>
3  <applet code = "WelcomeApplet.class" width = "300" height = "45">
4  </applet>
5  </body>
6  </html>
```

Figura 23.7 | `WelcomeApplet.html` carrega `WelcomeApplet` (Figura 23.6) em um contêiner de applets.

O applet e seu documento XHTML correspondente normalmente são armazenados no mesmo diretório do disco. Em geral, um navegador carrega um documento XHTML de um computador (além do seu) conectado à Internet. Entretanto, documentos XHTML também podem residir no seu computador (como visto na Seção 23.2). Quando um contêiner de applets encontra um elemento `applet` em um documento XHTML, ele carrega o arquivo (ou arquivos) `.class` do applet a partir da mesma localização que contém o documento XHTML.

O elemento `applet` tem vários **atributos**. O primeiro atributo na linha 3, `code = "WelcomeApplet.class"`, indica que o arquivo `WelcomeApplet.class` contém a classe compilada de applet. O segundo e terceiro atributos na linha 3 indicam a largura, ou **`width`**, (300) e a altura, ou **`height`**, (45) do applet em pixels. O tag `</applet>` (linha 4) termina o elemento `applet` que iniciou na linha 2. O tag `</html>` (linha 6) termina o documento XHTML.

Observações sobre a aparência e comportamento 23.1

Geralmente, um applet deve ter menos de 1.024 pixels de largura e 768 pixels de altura — dimensões suportadas pela maioria das telas de computador.

Erro comum de programação 23.1

Esquecer o tag `</applet>` de fechamento impede que o applet seja executado em alguns contêineres de applets. O `appletviewer` termina sem indicar um erro. Alguns navegadores da Web simplesmente ignoram o elemento `applet` incompleto.

Dica de prevenção de erro 23.1

Se você receber uma mensagem de erro `MissingResourceException` ao carregar um applet no `appletviewer` ou em um navegador, verifique cuidadosamente erros de sintaxe no tag `<applet>` do documento XHTML, como vírgulas (`,`) entre os atributos.

O `appletviewer` só entende os tags XHTML `<applet>` e `</applet>` e ignora todos os outros tags do documento. O `appletviewer` é um lugar ideal para testar um applet e assegurar que ele execute adequadamente. Depois que a execução do applet for verificada, você pode adicionar os tags XHTML a uma página Web para que outras pessoas possam visualizar nos seus navegadores da Web.

Para executar `WelcomeApplet` no `appletviewer`, abra uma janela de comando, mude para o diretório contendo seu applet e o documento XHTML e, então, digite

```
appletviewer WelcomeApplet.html
```

Dica de prevenção de erro 23.2

Teste seus applets no `appletviewer` antes de executá-los em um navegador da Web. Frequentemente, os navegadores salvam uma cópia de um applet na memória até que todas as janelas do navegador tenham sido fechadas. Se alterar um applet, recompile e então recarregue-o no seu navegador. O navegador talvez continue a executar a versão original do applet.

Dica de prevenção de erro 23.3

Teste seus applets em cada navegador da Web em que eles serão executados para assegurar que eles operam corretamente.

23.3.2 Executando um applet em um navegador da Web

As execuções do programa de exemplo na Figura 23.6 demonstram `WelcomeApplet` executando no `appletviewer` e no navegador da Web Mozilla Firefox. Para executar um applet no Firefox, siga os passos a seguir:

1. Selecione Open File no menu File.
2. Na caixa de diálogo que aparece, localize o diretório contendo o documento XHTML do applet que você deseja executar.
3. Selecione o documento XHTML.
4. Clique no botão Open.

Os passos para executar applets em outros navegadores da Web são semelhantes. Na maioria dos navegadores, você pode simplesmente pressionar *Ctrl* + *O* para abrir uma caixa de diálogo que permite selecionar um documento XHTML a partir do disco rígido local.

Dica de prevenção de erro 23.4

Se seu applet executar no `appletviewer`, mas não executar no seu navegador da Web, o Java não poderá ser instalado e configurado no seu navegador. Nesse caso, visite o site Web `Java.com` e clique no botão Free Java Download *para instalar o Java no seu navegador.*

23.4 Métodos de ciclo de vida de applet

Agora que você criou um applet, vamos considerar os cinco métodos de applet que são chamados pelo contêiner de applets a partir do momento em que o applet é carregado no navegador até que ele seja encerrado pelo navegador. Esses métodos correspondem a vários aspectos do ciclo de vida de um applet. A Figura 23.8 lista esses métodos, que são herdados para suas classes de applet da classe `JApplet`. A tabela especifica quando cada método é chamado e explica o seu propósito. Além do método `paint`, os corpos desses métodos permanecem vazios por padrão. Se quiser declarar qualquer um deles nos seus applets para que sejam chamados pelo contêiner de applets, você deverá utilizar os cabeçalhos de método mostrados na Figura 23.8.

Método	Descrição
`public void init()`	Chamado uma vez pelo contêiner de applets quando um applet é carregado para execução. Esse método inicializa um applet. Ações típicas realizadas aqui são: inicializar campos, criar componentes GUI, carregar sons para tocar, carregar e exibir imagens (ver Capítulo 24) e criar threads (ver Capítulo 26).
`public void start()`	Chamado pelo contêiner de applets depois de o método `init` completar a execução. Além disso, se o usuário for para outro site da Web e depois retornar à página XHTML do applet, o método `start` é chamado novamente. O método realiza todas as tarefas que devem ser completadas quando o applet é carregado pela primeira vez e isso deve ser feito todas as vezes que a página XHTML do applet for revisitada. As ações realizadas aqui poderiam incluir: iniciar uma animação ou iniciar outros threads da execução.
`public void paint( Graphics g )`	Chamado pelo contêiner de applets depois dos métodos `init` e `start`. O método `paint` também é chamado quando o applet precisa ser repintado. Por exemplo, se o usuário encobrir o applet com outra janela aberta na tela e mais tarde exibi-lo, o método `paint` será chamado. Ações típicas realizadas aqui envolvem desenhar com o objeto `g` de `Graphics` que é passado para o método `paint` pelo contêiner de applets.
`public void stop()`	Esse método é chamado pelo contêiner de applets quando o usuário sai da página Web do applet indo para outra página Web. Como é possível que o usuário retorne à página Web contendo o applet, o método `stop` realiza as tarefas que seriam necessárias para suspender a execução do applet, de modo que o applet não utilize o tempo de processamento do computador quando não é exibido na tela. Ações típicas realizadas aqui interromperiam a execução de animações e threads.

Método	Descrição
`public void destroy()`	Esse método é chamado pelo contêiner de applets quando o applet é removido da memória. Isso ocorre quando o usuário encerra a sessão de navegação fechando todas as janelas do navegador e também pode ocorrer sem que o navegador saiba quando o usuário tiver ido para outras páginas Web. O método realiza quaisquer tarefas necessárias para limpar recursos alocados ao applet.

Figura 23.8 | Métodos de ciclo de vida do `JApplet` chamados por um contêiner de applets durante a execução de um applet.

Erro comum de programação 23.2

Declarar os métodos `init`, `start`, `paint`, `stop` ou `destroy` com cabeçalhos de método que diferem daqueles mostrados na Figura 23.8 resulta em métodos que não serão chamados pelo contêiner de applets. O código especificado nas suas versões dos métodos não os executará. A notação `@Override` pode ser aplicada a cada método para evitar esse problema.

23.5 Inicializando uma variável de instância com o método `init`

Nosso próximo applet (Figura 23.9) calcula a soma dos dois valores inseridos em diálogos de entrada pelo usuário e exibe o resultado desenhando uma `String` dentro de um retângulo no applet. A soma é armazenada em uma variável de instância da classe `AdditionApplet`, de modo que possa ser utilizada no método `init` e no método `paint`. O documento XHTML que você pode utilizar para carregar esse applet em um contêiner de applets (isto é, o `appletviewer` ou um navegador Web) é mostrado na Figura 23.10.

```
1  // Figura 23.9: AdditionApplet.java
2  // Applet que adiciona dois valores double inseridos via caixas de diálogo de entrada.
3  import java.awt.Graphics;       // o programa utiliza a classe Graphics
4  import javax.swing.JApplet;     // o programa utiliza a classe JApplet
5  import javax.swing.JOptionPane; // o programa utiliza a classe JOptionPane
6
7  public class AdditionApplet extends JApplet
8  {
9     private double sum; // soma dos valores inseridos pelo usuário
10
11    // inicializa um applet obtendo os valores inseridos pelo usuário
12    public void init()
13    {
14       // obtém do usuário o primeiro número
15       String firstNumber = JOptionPane.showInputDialog(
16          "Enter first floating-point value" );
17
18       // obtém do usuário o segundo número
19       String secondNumber = JOptionPane.showInputDialog(
20          "Enter second floating-point value" );
21
22       // converte os números de tipo String para tipo duplo
23       double number1 = Double.parseDouble( firstNumber );
24       double number2 = Double.parseDouble( secondNumber );
25
26       sum = number1 + number2; // soma os números
27    } // fim do método init
28
29    // desenha os resultados em um retângulo sobre o fundo do applet
30    public void paint( Graphics g )
31    {
32       super.paint( g ); // chama a versão da superclasse do método paint
33
34       // desenha um retângulo iniciando em (15, 10) que tem 270
35       // pixels de largura e 20 pixels de altura
36       g.drawRect( 15, 10, 270, 20 );
37
38       // desenha os resultados como uma String em (25, 25)
```

```
39          g.drawString( "The sum is " + sum, 25, 25 );
40       } // fim do método paint
41    } // fim da classe AdditionApplet
```

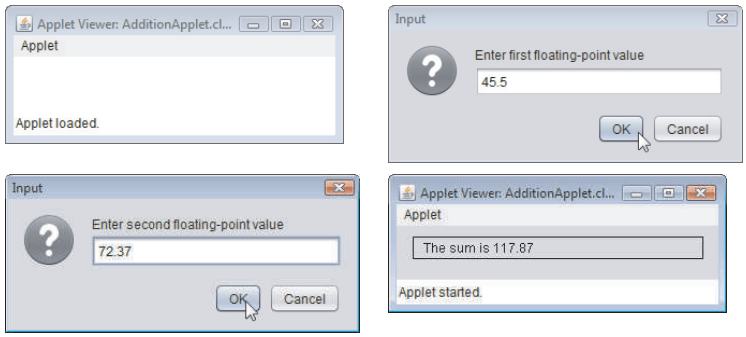


Figura 23.9 | Applet que adiciona dois valores `double` inseridos via caixas de diálogo de entrada.

```
1  <html>
2  <body>
3  <applet code = "AdditionApplet.class" width = "300" height = "50">
4  </applet>
5  </body>
6  </html>
```

Figura 23.10 | `AdditionApplet.html` carrega a classe `AdditionApplet` da Figura 23.9 em um contêiner de applets.

O applet solicita que o usuário insira dois números de ponto flutuante. Na Figura 23.9, a linha 9 declara a variável de instância `sum` do tipo `double`. O applet contém dois métodos — `init` (linhas 12–27) e `paint` (linhas 30–40). Quando um contêiner de applets carrega esse applet, o contêiner cria uma instância da classe `AdditionApplet` e chama seu método `init` — isso ocorre apenas uma vez durante a execução de um applet. O método `init` normalmente inicializa os campos do applet (se precisarem ser inicializados com valores além dos valores padrão) e realiza outras tarefas que devem ocorrer apenas uma vez quando o applet inicia a execução. A primeira linha do `init` sempre aparece como mostrado na linha 12, a qual indica que `init` é um método `public` que não recebe nenhum argumento e não retorna nenhuma informação depois de completar.

As linhas 15–24 declaram as variáveis para armazenar os valores inseridos pelo usuário, obtém a entrada do usuário e converte as `String`s inseridas pelo usuário em valores `double`. A linha 26 adiciona os valores armazenados nas variáveis `number1` e `number2`, e atribui o resultado à variável de instância `sum`. Nesse ponto, o método `init` do applet retorna o controle do programa ao contêiner de applets, que então chama o método `start` do applet. Não declaramos `start` nesse applet, portanto aquele herdado da classe `JApplet` é chamado aqui.

Em seguida, o contêiner de applets chama o método `paint` do applet, que desenha um retângulo (linha 36) em que o resultado da adição aparecerá. A linha 39 chama o método `drawString` do objeto `Graphics` para exibir os resultados. A instrução concatena o valor da variável de instância `sum` da `String` `"The sum is "` e exibe a `String` concatenada.

Observação de engenharia de software 23.1

As únicas instruções que devem ser colocadas no método `init` do applet são aquelas que devem ser executadas somente uma vez quando o applet é inicializado.

23.6 Modelo de segurança da caixa de areia

Por razões de segurança, geralmente é considerado perigoso permitir que applets ou qualquer outro programa que você executa a partir de um navegador Web acessem seu computador local. Desse modo, você deve decidir se *confia na fonte*. Por exemplo, se optar por fazer o download de uma nova versão do navegador Web Firefox a partir do site do Mozilla, `firefox.com`, você deverá decidir se confia no site do Mozilla. Afinal de contas, o programa instalador irá modificar seu sistema e inserir os arquivos para executar o Firefox no seu computador. Depois de instalado, o Firefox poderá acessar arquivos e outros recursos locais. A maioria das coisas que você faz via navegadores Web — como compras, navegação na Web e download de softwares — exige que você confie nos sites que você visita e também nas organizações que mantêm esses sites. Se você não for cuidadoso, um programa mal-intencionado baixado poderá ganhar controle sobre o computador,

acessar informações pessoais armazenadas aí, corromper seus dados e possivelmente até ser utilizado para atacar outros computadores na Internet — assim como muitas vezes acontece com vírus de computador hoje em dia.

Evitando applets mal-intencionados

Applets costumam ser descarregados da Internet. O que aconteceria se você fizesse o download de um applet mal-intencionado? Considere o fato de que um navegador baixa e executa um applet Java automaticamente — o usuário *não é solicitado a dar sua aprovação*. Na verdade, um applet é baixado *sem o conhecimento do usuário* — é apenas outro elemento da página Web que o usuário acessa.

Os projetistas do Java consideraram essa questão a fundo, uma vez que o Java foi concebido para uso em ambientes em rede. Para combater códigos mal-intencionados, a plataforma Java utiliza o chamado **modelo de segurança de caixa de areia** que fornece um mecanismo para executar código descarregado de uma maneira segura. Esse código executa na "caixa de areia" e não tem permissão de "rodar fora da caixa de areia". Por padrão, o código descarregado não pode acessar recursos do sistema local, e um applet só pode interagir com o servidor a partir do qual o applet foi descarregado.

Applets assinados digitalmente

Infelizmente, a execução em uma caixa de areia dificulta para os applets realizar tarefas úteis. É possível, porém, para uma organização que deseja criar applets com acesso ao sistema local obter um certificado de segurança (também chamado certificado digital) de uma entre várias autoridades certificadoras (ver `en.wikipedia.org/wiki/Certificate_Authority` para uma lista de autoridades e informações adicionais sobre autoridades certificadoras). A organização então pode utilizar as ferramentas fornecidas com o JDK para "assinar digitalmente" um applet que requer acesso aos recursos do sistema local. Quando um usuário baixa um applet digitalmente assinado, uma caixa de diálogo pergunta se o usuário confia na origem do applet. Nessa caixa de diálogo, o usuário pode visualizar o certificado de segurança da organização e ver qual autoridade certificadora o emitiu. Se o usuário indicar que ele confia na fonte, só então o applet será capaz de acessar os recursos do computador local.

Na próxima seção, introduzimos o Java Web Start e o Java Network Launch Protocol (JNLP). Essas tecnologias permitem a applets ou aplicativos interagir com o usuário para solicitar acesso a recursos específicos do sistema local. Com a permissão do usuário, isso permite a programadores Java estender a caixa de areia, mas ele não dá acesso aos programas a todos os recursos locais do usuário — assim os princípios da caixa de areia ainda estão em efeito. Por exemplo, seria útil para um programa de editor de textos descarregável armazenar os arquivos do usuário em uma pasta no computador do usuário. O editor de textos pode solicitar que o usuário dê permissão para fazer isso. Se o usuário der permissão de acesso a um diretório específico no disco, o programa então só poderá acessar esse diretório local e seus subdiretórios.

Para obter informações adicionais sobre applets digitalmente assinados, visite `java.sun.com/developer/onlineTraining/Programming/JDCBook/signed.html`. Para informações sobre o modelo de segurança da Java Platform, visite `java.sun.com/javase/6/docs/technotes/guides/security/`.

23.7 Java Web Start e o Java Network Launch Protocol (JNLP)

O **Java Web Start** é uma framework para executar applets e aplicativos descarregados fora do navegador. Tipicamente, esses programas são armazenados em um servidor Web para acesso via Internet, mas eles também podem ser armazenados na rede de uma organização para distribuição interna, ou até mesmo em CDs, DVDs ou outra mídia. Como você aprenderá no Capítulo 24, o Java Web Start permite que você pergunte ao usuário se um programa descarregado pode ter acesso aos recursos do computador do usuário.

Recursos do Java Web Start

Alguns recursos-chave do Java Web Start incluem:

- ***Integração com a área de trabalho:*** os usuários podem inicializar applets e aplicativos robustos clicando em um hiperlink em uma página Web e podem instalar rápida e facilmente os programas nos computadores. O Java Web Start pode ser configurado para perguntar ao usuário se um ícone na área de trabalho deve ser criado para que o usuário possa inicializar o programa diretamente da área de trabalho. Programas descarregados também podem ter um "modo off-line" para a execução se o computador não estiver conectado à Internet.
- ***Atualização automática:*** ao executar um programa via Java Web Start, o programa é descarregado e armazenado no cache (área de armazenamento temporário) no computador do usuário. Na próxima vez que o usuário executar esse programa, o Java Web Start o inicializa a partir do cache. Se o programa foi atualizado desde a última vez em que foi carregado, o Java Web Start pode baixar automaticamente as atualizações para que um usuário sempre tenha a versão mais recente. Isso torna simples e contínua a instalação e atualização do software para o usuário.
- ***Applets arrastáveis:*** esse é um novo recurso do Java SE 6 Update 10. Com uma pequena modificação no elemento `applet` que invoca um applet a partir de um documento XHTML, você pode permitir que os usuários executem um applet em uma janela própria mantendo pressionada a tecla *Alt* e arrastando o applet para fora do navegador Web. O applet continua a executar mesmo depois que o navegador Web fecha.

Java Network Launch Protocol (JNLP)

Um documento **Java Network Launch Protocol (JNLP)** fornece as informações que o Java Web Start precisa para fazer o download e executar um programa. Além disso, você deve empacotar seu programa em um ou mais repositórios de arquivo Java (JAR) que contêm o código e os recursos do programa e (por exemplo, imagens, arquivos de mídia, arquivos de texto).

Por padrão, programas inicializados via Java Web Start são executados utilizando o modelo de segurança de caixa de areia. Se o usuário der permissão, esses programas podem acessar o sistema de arquivos local, a área de transferência e outros serviços via JNLP APIs do pacote `javax.jnlp`. Discutiremos alguns desses recursos no Capítulo 24. Programas digitalmente assinados podem ter maior acesso ao sistema local se o usuário confiar na fonte.

23.7.1 Empacotando o applet `DrawTest` para uso com Java Web Start

Vamos empacotar o applet `DrawTest` de demonstração do JDK (discutido na Seção 23.2) para que você possa executá-lo via Java Web Start. Para fazer isso, você deve primeiro empacotar os arquivos e recursos `.class` do applet que ele utiliza (se houver algum) em um repositório de arquivos Java (JAR). Em uma janela de comando, mude para o diretório `DrawTest`, como você fez na Seção 23.2. Nessa pasta, execute o seguinte comando:

```
jar cvf DrawTest.jar *.class
```

que cria um arquivo JAR no diretório atual nomeado `DrawTest.jar` contendo os arquivos `.class` do applet — `DrawControls.class`, `DrawPanel.class` e `DrawTest.class`. Se o programa tivesse outros recursos, você simplesmente adicionaria os nomes de arquivo ou os nomes de pasta em que esses recursos são armazenados no fim do comando precedente. As letras `cvf` são opções de linha de comando para o **comando `jar`**. A **opção `c`** indica que o comando deve criar um novo arquivo JAR. A **opção `v`** indica que o comando deve produzir uma saída detalhada para que você possa ver a lista de arquivos e diretórios que estão sendo incluídos no arquivo JAR. A **opção `f`** indica que o próximo argumento na linha de comando (`DrawTest.jar`) é o novo nome do arquivo JAR. A Figura 23.11 mostra a saída detalhada do comando precedente, que mostra os arquivos que foram inseridos no JAR.

```
added manifest
adding: DrawControls.class(in = 2611) (out= 1488)(deflated 43%)
adding: DrawPanel.class(in = 2703) (out= 1406)(deflated 47%)
adding: DrawTest.class(in = 1170) (out= 706)(deflated 39%)
```

Figura 23.11 | Saída do comando `jar`.

23.7.2 Documento JNLP para o applet `DrawTest`

Em seguida, você deve criar um documento JNLP que descreve o conteúdo do arquivo JAR e especifica que arquivo no JAR é o chamado **`main-classe`** que inicia a execução do programa. Para um applet, o `main-class` é aquele que estende `JApplet` (isto é, `DrawTest` nesse exemplo). Para um aplicativo, o `main-class` é aquele que contém o método `main`. Um documento JNLP básico para o applet `DrawTest` é mostrado na Figura 23.12. Descreveremos os elementos desse documento mais adiante.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp
3     codebase=CaminhoParaOArquivoJNLP
4     href="DrawTest.jnlp"     >
5
6     <information>
7        <title>DrawTest Applet</title>
8     <vendor>Sun Microsystems, Inc.</vendor>
9        <shortcut>
10          <desktop/>
11       </shortcut>
12       <offline-allowed/>
13    </information>
14
15    <resources>
16       <java version="1.6+"/>
17       <jar href="DrawTest.jar" main="true"/>
18    </resources>
19
```

```
20      <applet-desc
21         name="DrawTest"
22         main-class="DrawTest"
23         width="400"
24         height="400">
25      </applet-desc>
26   </jnlp>
```

Figura 23.12 | Documento `DrawTest.jnlp` para inicializar o applet `DrawTest`.

Visão geral da XML

Os documentos de JNLP são escritos na **Extensible Markup Language (XML)** — um padrão amplamente suportado para descrever dados. A XML é comumente utilizada na troca de dados entre aplicativos pela Internet, e muitos aplicativos agora utilizam a XML para também especificar informações sobre a configuração — como é o caso com documentos JNLP para o Java Web Start. A XML permite criar marcações para praticamente qualquer tipo de informações. Isso permite criar linguagens de marcação inteiramente novas para descrever qualquer tipo de dados, como fórmulas matemáticas, instruções de configuração de software, estruturas químicas moleculares, músicas, notícias, receitas e relatórios financeiros. A XML descreve dados de uma maneira que tanto os seres humanos como os computadores podem entender. O JNLP é um vocabulário XML que descreve as informações que o Java Web Start precisa para inicializar um programa.

Documentos XML contêm **elementos** que especificam a estrutura do documento, como `title` (linha 7), e texto que representa o conteúdo (isto é, dados), como `DrawTest Applet` (linha 7). Documentos XML delimitam os elementos com **tags de abertura e fechamento**. Um tag de abertura consiste no nome do elemento entre **colchetes angulares** (por exemplo, `<title>` e `vendor>` nas linhas 7 e 8). Tags de abertura também podem conter atributos da forma *nome=valor* — por exemplo, o tag de início `jnlp` contém o atributo `href= "DrawTest.jnlp"`. Um tag de fechamento consiste no nome do elemento precedido por uma **barra (/)** entre colchetes angulares (por exemplo, `</title>` e `</vendor>` nas linhas 7–8). Os tags de abertura e fechamento de um elemento cercam o texto que representa um dado (por exemplo, o `vendor` (fabricante) do programa — `Sun Microsystems, Inc.` — na linha 8, que é cercado pelo tag de abertura `<vendor>` e pelo tag de fechamento `</vendor>`) ou outros elementos (por exemplo, os elementos `title`, `vendor`, `shortcut` e `offline-allowed` no elemento `information` das linhas 6–13). Cada documento XML deve ter exatamente um único elemento-raiz que contém todos os outros elementos. No Figura 23.12, o **elemento `jnlp`** (linhas 2–26) é o **elemento-raiz**.

Documento JNLP: Elemento `jnlp`

O tag de abertura do elemento `jnlp` (linhas 2–4) tem dois atributos — `codebase` e `href`. O valor do atributo `codebase` é um URL que especifica o caminho em que o documento JNLP e o arquivo JAR estão armazenados — isso é especificado na Figura 23.12 como *CaminhoParaOArquivoJNLP*, uma vez que esse valor depende da localização a partir da qual o applet é carregado. O atributo `href` especifica o arquivo JNLP que inicializa o programa. Salvamos o arquivo JNLP e o arquivo JAR no diretório do applet de demonstração `DrawTest` dentro da estrutura de diretórios do JDK. Utilizamos o seguinte URL do sistema de arquivos local como o `codebase`:

```
file:.
```

que indica que o código está no diretório atual (`.`). Em geral, o `codebase` refere-se a um diretório em um servidor Web com um URL `http://`. Se quiser disponibilizar seu applet ou aplicativo a partir de um servidor Web para que os usuários possam acessá-lo on-line, você precisará configurar o servidor Web corretamente, como descrito em `java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/setup.html`.

Documento JNLP: Elemento `information`

O **elemento `information`** (linhas 6–13) fornece detalhes sobre o programa. O **elemento `title`** especifica um título para o programa. O **elemento `vendor`** especifica quem criou o programa. Os valores desses elementos aparecem nos erros e alertas de segurança do Java Web Start e são apresentados ao usuário. O valor de `title` também aparece na barra de título da janela em que o programa executa.

O **elemento `desktop`**, aninhado no **elemento `shortcut`** (linhas 9–11) instrui o Java Web Start a perguntar se o usuário deseja instalar um atalho na área de trabalho. Se o usuário aceitar, um ícone aparecerá na área de trabalho. O usuário pode então inicializar o programa em janela própria dando um clique duplo no ícone na área de trabalho. Observe a sintaxe do elemento `<desktop/>` — elemento XML vazio. Quando nada aparece entre os tags de abertura e fechamento de um elemento, o elemento pode ser escrito utilizando um tag de fechamento que termina com `/>`.

O **elemento `offline-allowed`** (linha 12) indica que depois de o programa ser instalado no computador do usuário, ele pode ser inicializado via Java Web Start — mesmo quando o computador não está conectado à Internet. Isso é especialmente útil para qualquer programa que pode ser utilizado com arquivos armazenados no computador do usuário.

Documento JNLP: elemento `resources`

O **elemento `resources`** (linhas 15–18) contém dois elementos aninhados. O **elemento `java`** lista a versão mínima do Java necessária para executar o programa (linha 16) e o **elemento `jar`** (linha 17) especifica a localização do arquivo JAR que contém o programa e se esse arquivo JAR contém a classe que inicializa o programa. Pode haver múltiplos elementos `jar`, como veremos no próximo capítulo.

Documento JNLP: elemento `applet-desc`

O **elemento `applet-desc`** (linhas 20–25) é semelhante ao elemento `applet` na XHTML. O atributo `name` especifica o nome do applet. O atributo `main-class` especifica a classe de applet principal (aquela que estende `JApplet`). Os atributos `width` e `height` especificam a largura e altura em pixels, respectivamente, da janela em que o applet executará. O Capítulo 24 discute um elemento semelhante para aplicativos — `application-desc`.

Inicializando o applet com o Java Web Start

Você agora está pronto para inicializar o applet via Java Web Start. Há várias maneiras de fazer isso. Você pode utilizar o **comando `javaws`** em uma janela de comando a partir da pasta que contém o documento JNLP, como em

```
javaws DrawTest.jnlp
```

Você também pode utilizar o gerenciador de arquivos do seu sistema operacional para localizar o JNLP no computador e dar um clique duplo no nome de arquivo. Normalmente, o arquivo JNLP é referenciado a partir de uma página Web via um hiperlink. O documento `DrawTestWebPage.html` na Figura 23.13 (que foi salvo no mesmo diretório que o arquivo JNLP) contém um elemento âncora (`a`) (linha 4), que é vinculado ao arquivo `DrawTest.jnlp`. Clicar nesse hiperlink na página Web descarrega o arquivo JNLP (nesse caso, ele é carregado do sistema de arquivos local) e executa o applet correspondente.

```
1  <html>
2     <head><title>DrawTest Launcher Page</title></head>
3     <body>
4        <a href="DrawTest.jnlp">Launch DrawTest via Java Web Start</a>
5     </body>
6  </html>
```

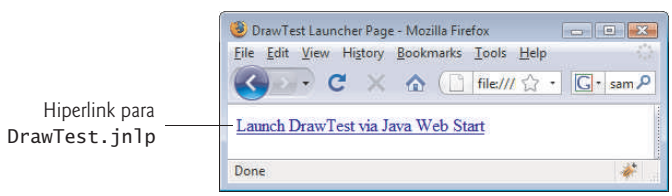


Figura 23.13 | O documento XHTML que inicializa o applet `DrawTest` quando o usuário clica no link.

Ao executar o applet pela primeira vez via Java Web Start, você verá uma caixa de diálogo como a da Figura 23.14. Essa caixa de diálogo permite que o usuário decida se um ícone será instalado na área de trabalho. Se o usuário clicar em **OK**, um novo ícone rotulado com o título especificado no documento JNLP aparece na área de trabalho do usuário. O applet também é armazenado em cache para uso futuro. Depois de o usuário clicar em **OK** ou **Skip** nessa caixa de diálogo, o programa executa (Figura 23.15).

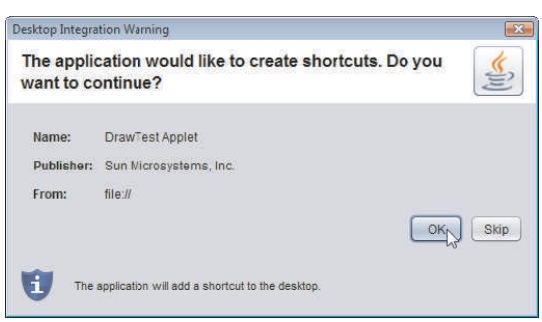


Figura 23.14 | A caixa de diálogo pergunta se o usuário deseja instalar um atalho na área de trabalho.

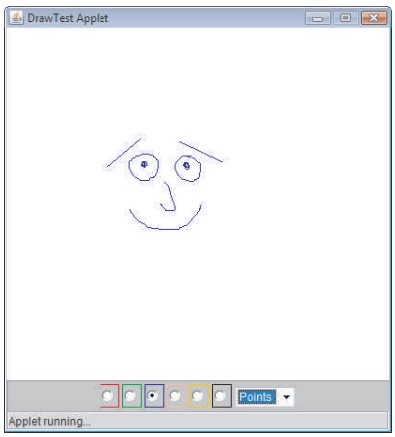


Figura 23.15 | Applet `DrawTest` em execução e com o Java Web Start.

Visualizando programas instalados via Java Web Star

Você pode visualizar os programas instalados via Java Web Start no **Java Cache Viewer** digitando o seguinte comando em uma janela de comandos:

```
javaws -viewer
```

Isso exibe a janela na Figura 23.16. O **Java Cache Viewer** permite gerenciar os programas Java Web Start no seu sistema. Você pode executar um programa selecionado, criar um atalho na área de trabalho para um programa (se não houver um ainda), excluir programas instalados etc.

Para obter informações adicionais sobre o Java Web Start , visite `java.sun.com/javase/6/docs/technotes/guides/javaws/`. Esse site fornece uma visão geral do Java Web Start e inclui links para o Developer's Guide, um FAQ, a JNLP Specification e a documentação da API para o pacote `javax.jnlp`.

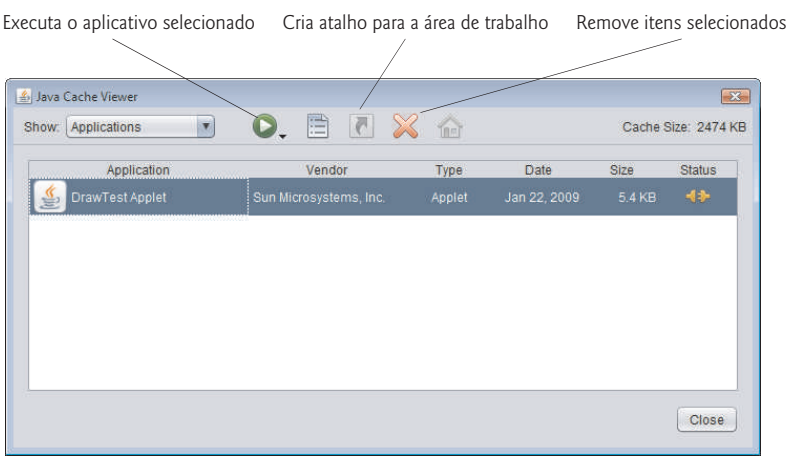


Figura 23.16 | Visualizando programas instalados via Java Web Start no **Java Cache Viewer**.

23.8 Conclusão

Neste capítulo, você aprendeu os fundamentos dos applets Java e do Java Web Start. Conheceu os conceitos básicos da XHTML para incorporar um applet a uma página Web e executá-lo em um contêiner de applets como o `appletviewer` ou um navegador Web. Além disso, elencamos os cinco métodos que são chamados automaticamente pelo contêiner de applets durante o ciclo de vida de um applet. Discutimos o modelo de segurança de caixa de areia do Java para executar código descarregado. Introduzimos então o Java Web Start e o Java Network Launch Protocol (JNLP). Você aprendeu a empacotar um programa em um arquivo JAR para que ele possa ser executado via Java Web Start. Também apontamos os elementos básicos de um documento JNLP. No próximo capítulo, você verá vários applets adicionais à medida que apresentamos os recursos de multimídia básicos. Também aprenderá mais recursos do Java Web Start e JNLP. No Capítulo 27, demonstraremos como personalizar um applet via parâmetros que são especificados em um elemento XHTML `applet`.

Resumo

Seção 23.1 Introdução

- Applets são programas Java que costumam ser incorporados a documentos XHTML (Extensible HyperText Markup Language) — também chamados páginas Web. Quando um navegador Web compatível com Java carrega uma página Web que contém um applet, o applet é baixado no navegador e executa.
- O aplicativo no qual um applet executa é conhecido como contêiner de applets. É responsabilidade do contêiner de applets inicializar a(s) classe(s) do applet, criar uma instância do applet e gerenciar seu ciclo de vida. O JDK inclui um contêiner de applets chamado `appletviewer` para testar applets à medida que você os desenvolve e antes de incorporá-los a páginas Web.
- Navegadores Web executam applets Java via Plug-in Java.

Seção 23.2 Applets de exemplo fornecidos com o JDK

- Para reexecutar um applet no `appletviewer`, clique no menu **Applet** do `appletviewer` e selecione o item de menu **Reload**.
- Para encerrar o `appletviewer`, selecione **Quit** no menu **Applet** do `appletviewer`.

Seção 23.3 Applet Java simples: desenhando uma string

- Cada applet Java é uma interface gráfica com o usuário em que você pode desenhar ou colocar componentes GUI.
- A classe `JApplet` do pacote `javax.swing` é utilizada para criar applets.
- Um contêiner de applets pode criar apenas objetos de classes `public` e estender a `JApplet` (ou a classe `Applet` nas versões anteriores do Java).
- Um contêiner de applets espera que cada applet Java tenha os métodos `init`, `start`, `paint`, `stop` e `destroy`, cada um dos quais é declarado na classe `JApplet`. Cada nova classe de applet que você cria herda as implementações padrão desses métodos da classe `JApplet`.
- Quando um contêiner de applets carrega um applet, ele cria um objeto do tipo de applet e, então, chama os métodos `init`, `start` e `paint` do applet. Se você não declarar esses métodos no seu applet, o contêiner de applets chamará as versões herdadas.
- Os métodos `init` e `start` da superclasse têm corpos vazios, portanto eles não realizam nenhuma tarefa. O método `paint` da superclasse não desenha nada no applet.
- Para permitir que um applet desenhe, sobrescreva o método `paint`. Você não chama o método `paint` explicitamente em um applet. Em vez disso, o contêiner de applets chama `paint` para dizer ao applet quando desenhar, e o contêiner de applets é responsável por passar um objeto `Graphics` como um argumento.
- A primeira instrução no método `paint` deve ser uma chamada ao método `paint` da superclasse. Omitir isso pode causar erros sutis de desenho em applets que combinam desenho e componentes GUI.
- Para poder executar um applet, antes você deve criar um documento XHTML (Extensible Hypertext Markup Language) que especifica qual applet executar no contêiner de applets. Em geral, um documento XHTML termina com uma extensão de nome de arquivo "`.html`" ou "`.htm`".
- A maioria dos elementos XHTML é delimitada por pares de `tags`. Todos os tags XHTML iniciam com um sinal de menor, <, e terminam com um sinal de maior, >.
- Um elemento `applet` instrui o contêiner de applets a carregar um applet específico e define sua área de exibição (sua largura e altura em pixels) no contêiner de applets.
- Normalmente, um applet e seu documento XHTML correspondente são armazenados no mesmo diretório.
- Quando um contêiner de applets encontra um documento XHTML que contém um applet, o contêiner de applets carrega automaticamente o(s) arquivo(s) `.class` do applet do mesmo diretório no computador em que o documento XHTML reside.
- O `appletviewer` só entende os tags XHTML `<applet>` e `</applet>` e ignora todos os outros tags do documento.

Seção 23.4 Métodos de ciclo de vida de applet

- Cinco métodos do ciclo de vida de um applet são chamados pelo contêiner de applets entre o momento em que o applet é carregado no navegador e o momento em que ele é fechado pelo navegador.
- O método `init` é chamado uma vez pelo contêiner de applets para inicializar um applet quando ele é carregado.
- O método `start` é chamado pelo contêiner de applets depois de o método `init` completar a execução. Além disso, se o usuário for para outro site da Web e depois retornar à página XHTML do applet, o método `start` é chamado novamente.
- O método `paint` é chamado pelo contêiner de applets depois dos métodos `init` e `start`. O método `paint` também é chamado quando o applet precisa ser repintado.
- O método `stop` é chamado pelo contêiner de applets quando o usuário sai da página Web do applet indo para outra página Web.
- O método `destroy` é chamado pelo contêiner de applets quando o applet é removido da memória. Isso ocorre quando o usuário encerra a sessão de navegação fechando todas as janelas do navegador e também pode ocorrer sem que o navegador saiba quando o usuário tiver ido para outras páginas Web.

Seção 23.5 Inicializando uma variável de instância com o método `init`

- O método `Graphics` `drawString` desenha uma `String` em uma localização especificada.
- O método `Graphics` `drawRect` desenha um retângulo no canto superior esquerdo com a largura e altura especificadas.

Seção 23.6 Modelo de segurança da caixa de areia

- É considerado perigoso permitir que applets ou outros programas que você executa a partir de um navegador Web acessem seu computador local. Você deve decidir se *confia na fonte*.
- Um navegador baixa um applet *sem o conhecimento do usuário* — é apenas outro elemento da página Web que o usuário acessa.
- Para combater códigos mal-intencionados, a plataforma Java utiliza um modelo de segurança de caixa de areia que fornece um mecanismo para executar códigos descarregados de uma maneira segura. O código baixado não pode acessar recursos do sistema local, e um applet só pode interagir com o servidor de onde foi baixado.
- Você pode "assinar digitalmente" um applet que requer acesso aos recursos do sistema local. Se o usuário indicar que ele confia na origem do applet, só então o applet será capaz de acessar os recursos do computador local.

Seção 23.7 Java Web Start e o Java Network Launch Protocol (JNLP)

- O Java Web Start é um framework para executar programas descarregados fora do navegador. Em geral, esses programas são armazenados em um servidor Web, mas eles também podem ser armazenados na rede de uma organização para distribuição interna, ou até mesmo em CDs, DVDs ou outra mídia.
- Os usuários podem inicializar applets e aplicativos robustos clicando em um hiperlink em uma página Web, e podem instalar rápida e facilmente os programas nos computadores.
- O Java Web Start pode ser configurado para perguntar ao usuário se um ícone na área de trabalho deve ser criado para que o usuário possa inicializar o programa diretamente da área de trabalho. Programas descarregados também podem ter um "modo off-line" para a execução se o computador não estiver conectado à Internet.
- Ao executar um programa via Java Web Start, o programa é descarregado e armazenado no cache (área de armazenamento temporário) no computador do usuário. Na próxima vez que o usuário executar esse programa, o Java Web Start o inicializa a partir do cache.
- Se o programa foi atualizado desde a última vez em que foi carregado, o Java Web Start pode baixar automaticamente as atualizações para que um usuário sempre tenha a versão mais recente.
- Um documento Java Network Launch Protocol (JNLP) fornece as informações de que o Java Web Start precisa para baixar e executar um programa.
- Programas inicializados via o Java Web Start são executados utilizando o modelo de segurança de caixa de areia. O usuário pode permitir acesso ao sistema de arquivos local, à área de transferência e a outros serviços via JNLP APIs.

Seção 23.7.1 Empacotando o applet `DrawTest` para uso com o Java Web Start

- O comando `jar` é utilizado para criar arquivos JAR. A opção `c` indica que o comando deve criar um novo arquivo JAR. A opção `v` indica que o comando deve produzir uma saída prolixa. A opção `f` indica que o próximo argumento na linha de comandos é o novo nome do arquivo JAR.

Seção 23.7.2 Documento JNLP para o applet `DrawTest`

- Um documento JNLP descreve o conteúdo do arquivo JAR e especifica qual arquivo no JAR é o assim chamado `main-class` que inicia a execução do programa.
- Os documentos de JNLP são escritos na Extensible Markup Language (XML) — um padrão amplamente suportado para descrever dados.
- O JNLP é um vocabulário XML que descreve as informações que o Java Web Start precisa para inicializar um programa.
- Documentos XML contêm elementos que especificam a estrutura do documento. Documentos XML delimitam os elementos com tags de abertura e fechamento. Um tag de abertura consiste no nome do elemento entre colchetes angulares. Um tag de abertura também pode conter atributos da forma *nome=valor*. Um tag de fechamento consiste no nome do elemento precedido por uma barra (`/`) entre colchetes angulares.
- Os tags de abertura e fechamento de um elemento cercam o texto que representa um dado ou outros elementos.
- Cada documento XML deve ter exatamente um único elemento-raiz que contém todos os outros elementos.
- O atributo `codebase` do elemento `jnlp` especifica o caminho em que o documento JNLP e o arquivo JAR são armazenados. O atributo `href` especifica o arquivo JNLP que inicializa o programa.
- Em geral, o `codebase` refere-se a um diretório em um servidor Web com um URL `http://`.
- O elemento `information` fornece detalhes sobre o programa.
- O elemento `title` especifica um título para o programa.
- O elemento `vendor` especifica quem criou o programa.
- O elemento `desktop`, aninhado no elemento `shortcut`, instrui o Java Web Start a perguntar aos usuários se eles desejam instalar um atalho na área de trabalho.
- O elemento `offline-allowed` indica que um programa pode ser inicializado via Java Web Start mesmo quando o computador não está conectado à Internet.

- O elemento `resources` contém um elemento `java` que lista a versão mínima do Java necessária para executar o programa e um elemento `jar` que especifica a localização do arquivo JAR.
- O atributo `name` do elemento `applet-desc` especifica o nome do applet. O atributo `main-class` especifica a classe de applet principal. Os atributos `width` e `height` especificam a largura e altura em pixels, respectivamente, da janela em que o applet executará.
- Para inicializar o applet via Java Web Start, utilize o comando `javaws`. Você também pode utilizar o gerenciador de arquivos do seu sistema operacional para localizar o JNLP no computador e dar um clique duplo no nome de arquivo. Normalmente, um arquivo JNLP é referenciado a partir de uma página Web via um hiperlink.
- Ao executar um applet via Java Web Start pela primeira vez e o documento JNLP especifica que um ícone deve ser instalado na área de trabalho, você verá uma caixa de diálogo que permite decidir se é preciso instalar o ícone na área de trabalho. Se clicar em **OK**, um novo ícone rotulado com o título especificado no documento JNLP aparece na área de trabalho.
- Você pode visualizar os programas instalados via Java Web Start no **Java Cache Viewer** digitando o comando `javaws -viewer` em uma janela de comandos:
- O **Java Cache Viewer** permite gerenciar os programas instalados via Java Web Start. Você pode executar um programa selecionado, criar um atalho na área de trabalho, excluir programas instalados etc.

Terminologia

/, barras em tags de fechamento, 735
<>, colchetes angulares para elementos XML, 735
`.htm`, extensão de nome de arquivo, 729
`.html`, extensão de nome de arquivo, 729
applet, 723
`applet-desc`, elemento de um documento JNLP, 736
`applet`, elemento XHTML, 729
`appletviewer`, contêiner de applets, 723
atributo de um elemento XHTML, 729
`body`, elemento XHTML, 729
`c`, opção do comando `jar`, 734
caractere de barra (/) em tags de fechamento, 735
colchete angular (<>) para elementos XML, 735
contêiner de applets, 723
`desktop`, elemento de um documento JNLP, 735
`destroy`, método da classe `JApplet`, 728
elemento-raiz (XML), 735
elemento (XML), 735
Extensible Markup Language (XML), 735
`f`, opção do comando `jar`, 734
`height` de um applet em pixels, 729
`information`, elemento de um documento JNLP, 735
`init`, método da classe `JApplet`, 728
`JApplet`, classe, 728
`jar`, comando, 734
`jar`, elemento de um documento JNLP, 736
`java`, elemento de um documento JNLP, 736
Java Network Launch Protocol (JNLP), 734
Java Plug-in, 723
Java Web Start, 733
`javaws`, comando, 736
`jnlp`, elemento de um documento JNLP, 735
`main-class` especificada em um documento JNLP, 734
modelo de segurança de caixa de areia, 733
`offline-allowed`, elemento de um documento JNLP, 735
`paint`, método da classe `JApplet`, 728
`resources`, elemento de um documento JNLP, 736
`shortcut`, elemento de um documento JNLP, 735
sinais de maior e menor (<>) para elementos XML, 735
`start`, método da classe `JApplet`, 728
`stop`, método da classe `JApplet`, 728
tag de abertura, 735
tag de fechamento, 735
tag (em um documento de XHTML), 729
`title`, elemento de um documento JNLP, 735
`v`, opção do comando `jar`, 734
`vendor`, elemento de um documento JNLP, 735
`width` de um applet em pixels, 729
XHTML (Extensible HyperText Markup Language), documento, 723

Exercício de autorrevisão

23.1 Preencha as lacunas em cada uma das seguintes sentenças:

a) Os applets Java começam a execução com uma série de três chamadas de método: _____, _____ e _____.

b) O método _____ é invocado para um applet toda vez que o usuário de um navegador deixa a página XHTML em que o applet reside.

c) Cada applet deve estender a classe _____.

d) O(A) _____ ou um navegador pode ser utilizado para executar um applet Java.

e) O método _____ é chamado toda vez que o usuário de um navegador revisita a página XHTML em que um applet reside.

f) Para carregar um applet em um navegador você deve primeiro definir um arquivo _____.

g) O método _____ é chamado uma vez quando um applet inicia a execução.

h) O método _____ é invocado para desenhar em um applet.

i) O método _____ é invocado para um applet quando o navegador o remove da memória.

j) Os tags XHTML _____ e _____ especificam que um applet deve ser carregado em um contêiner de applets e executado.

k) _____ é uma estrutura para executar programas baixado fora do navegador.

l) Um documento _____ fornece as informações que a Java Web Start precisa para baixar e executar um programa.

m) O _____ permite gerenciar os programas Java Web Start no seu sistema.

Respostas do exercício de autorrevisão

23.1 a) `init`, `start`, `paint`. b) `stop`. c) `JApplet` (ou `Applet`). d) `appletviewer`. e) `start`. f) XHTML. g) `init`. h) `paint`. i) `destroy`. j) `<applet>`, `</applet>`. k) Java Web Start. l) Java Network Launch Protocol (JNLP). m) Java Cache Viewer.

Exercícios

23.2 Escreva um applet que pede que o usuário insira dois números de ponto flutuante, obtém do usuário os dois números e desenha sua soma, produto (multiplicação), diferença e quociente (divisão). Utilize as técnicas mostradas na Figura 23.9.

23.3 Escreva um applet que pede que o usuário insira dois números de ponto flutuante, obtém do usuário os números e exibe os dois números e, então, exibe o maior número seguido pelas palavras `"is larger"` como uma string no applet. Se os números forem iguais, o applet deve imprimir a mensagem `"These numbers are equal"`. Utilize as técnicas mostradas na Figura 23.9.

23.4 Escreva um applet que insere três números de ponto flutuante do usuário e exibe a soma, média, produto, maior e menor desses números como strings no applet. Utilize as técnicas mostradas na Figura 23.9.

23.5 Escreva um applet que pede que o usuário insira o raio de um círculo como um número de ponto flutuante e desenha o diâmetro do círculo, circunferência e área. Utilize o valor 3.14159 para π. Utilize as técnicas mostradas na Figura 23.9. [*Nota:* Você também pode utilizar a constante `Math.PI` predefinida para o valor de π. Essa constante é mais precisa que o valor 3,14159. A classe `Math` é definida no pacote `java.lang`, então você não precisa `importá`-la.] Utilize as seguintes fórmulas (*r* é o raio):
diâmetro = $2r$
circunferência = $2\pi r$
área = πr^2

23.6 Escreva um applet que lê cinco inteiros, determina e imprime o maior e o menor inteiro no grupo. Desenhe os resultados no applet.

23.7 Escreva um applet que desenha um padrão de tabuleiro de damas como segue:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

23.8 Escreva um applet que desenha retângulos de diferentes tamanhos e localizações.

23.9 Escreva um applet que permite ao usuário inserir os quatro valores para os argumentos requeridos pelo método `drawRect`, então desenhe um retângulo utilizando os quatro valores de entrada.

23.10 A classe `Graphics` contém o método `drawOval`, que recebe como argumentos os mesmos quatro argumentos do método `drawRect`. Os argumentos do método `drawOval` especificam o "quadro delimitador" da oval — os lados do quadro delimitador são os limites da oval. Escreva um applet Java que desenha uma oval e um retângulo com os mesmos quatro argumentos. A oval tocará o retângulo no centro de cada lado.

23.11 Modifique a solução do Exercício 23.10 para gerar a saída de ovais de diferentes formas e tamanhos.

23.12 Escreva um applet que permite ao usuário inserir os quatro argumentos requeridos pelo método `drawOval`, então desenhe uma oval utilizando os quatro valores de entrada.

23.13 Empacote o applet de demonstração `TicTacToe` do JDK (discutido na Seção 23.2) para uso com o Java Web Start, copie então o documento JNLP na Figura 23.12 e modifique-o para que ele inicialize o applet `TicTacToe`.

24

A roda que faz mais barulho ... é a que obtém a graxa.
— **John Billings (Henry Wheeler Shaw)**

Utilizaremos um sinal que eu testei e considerei de grande alcance e fácil de gritar. Uaa-huu!
— **Zane Grey**

Há uma dança natural no movimento de um peixe de aquário.
— **Walt Disney**

Entre o movimento e o ato está a sombra.
— **Thomas Stearns Eliot**

Multimídia: applets e aplicativos

Objetivos

Neste capítulo, você aprenderá:

- A obter, exibir e redimensionar imagens.
- A criar animações de sequências de imagens.
- A criar mapas de imagem.
- A obter, reproduzir, repetir (loop) e interromper sons, utilizando um `AudioClip`.
- A reproduzir vídeo utilizando a interface `Player`.

Sumário

24.1 Introdução
24.2 Carregando, exibindo e dimensionando imagens
24.3 Animação de uma série de imagens
24.4 Mapas de imagem
24.5 Carregando e reproduzindo clipes de áudio
24.6 Reproduzindo vídeo e outros tipos de mídia com o Java Media Framework
24.7 Conclusão
24.8 Recursos da Web

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Seção especial: projetos de multimídia desafiadores | Fazendo a diferença

24.1 Introdução

A **multimídia** — o uso de som, imagens, elementos gráficos, animação e vídeo — faz aplicativos "ganharem vida". Embora a maior parte da multimídia em aplicativos Java seja bidimensional, você pode utilizar a **Java 3D API** para criar aplicativos gráficos 3D (`java.sun.com/javase/technologies/desktop/java3d/`).

A maioria dos computadores novos vendidos atualmente é multimídia, pois já são equipados com unidades de DVD e capacidades de reprodução de áudio e vídeo. Os computadores desktop e laptop são tão poderosos que podem armazenar e reproduzir som e vídeo com qualidade de DVD (e muitas vezes, com qualidade de HD).

Entre os usuários que querem imagens gráficas, muitos agora querem imagens gráficas tridimensionais coloridas e de alta resolução. Verdadeiras imagens tridimensionais já estão disponíveis. Espera-se que a televisão tridimensional e de alta resolução com a experiência de um "teatro arena" (em que a audiência circunda o palco) acabe se tornando comum. Eventos esportivos e de entretenimento parecerão acontecer dentro da sua sala! Estudantes de medicina em todo o mundo verão operações sendo realizadas a milhares de milhas de distância, como se estivessem ocorrendo no mesmo lugar. As pessoas aprenderão como guiar com simuladores de direção incrivelmente realistas em suas casas antes de pegarem no volante. As possibilidades são infinitas e empolgantes.

A multimídia demanda um extraordinário poder de computação. Até recentemente, computadores economicamente acessíveis com esse tipo de poder não estavam amplamente disponíveis. Os processadores ultrapoderosos atuais tornam a multimídia efetivamente econômica. Os usuários estão ávidos por processadores mais poderosos, mais memória e canais de comunicação mais rápidos que suportem os exigentes aplicativos multimídia. Ironicamente, essas capacidades aprimoradas podem não custar mais — a concorrência feroz continua abaixando os preços.

As Java APIs fornecem recursos multimídia que permitem começarmos a desenvolver imediatamente poderosos aplicativos multimídia. Este capítulo apresenta vários exemplos, incluindo:

1. Princípios básicos de manipulação de imagens.
2. Criação de animações suaves.
3. Reprodução de arquivos de áudio com a interface `AudioClip`.
4. Criação de mapas de imagem que podem perceber quando o cursor está sobre eles, mesmo sem o clique de mouse.
5. Reprodução de arquivos de vídeo com a interface `Player`.

O capítulo também introduz recursos de JNLP adicionais que, com a permissão do usuário, permitem que um applet ou aplicativo acesse arquivos no computador local do usuário. Os exercícios sugerem dezenas de projetos desafiadores e interessantes. Quando estávamos escrevendo-os, as ideias simplesmente não paravam de fluir. A multimídia estimula a criatividade de uma maneira que não experimentamos com as capacidades de computadores "convencionais". [*Nota*: As capacidades multimídia do Java vão bem além das apresentadas neste capítulo. Elas incluem a **Java Media Framework (JMF) API** (para adicionar mídia de áudio e vídeo a um aplicativo), a **Java Sound API** (para reproduzir, gravar e modificar áudio), Java 3D API (para criar e modificar imagens gráficas 3D), a **Java Advanced Imaging API** (para capacidades de processamento de imagens, como cortar e redimensionar), a **Java Speech API** (para inserir comandos de voz do usuário ou gerar saída de comandos de voz para o usuário), a Java 2D API (para criar e modificar imagens gráficas 2D, discutidas no Capítulo 15) e a **Java Image I/O API** (para ler e gerar a saída de imagens para arquivos). A Seção 24.8 fornece links Web para essas APIs.]

24.2 Carregando, exibindo e dimensionando imagens

Começamos nossa discussão com imagens. Utilizaremos várias imagens diferentes neste capítulo. Você pode criar suas próprias imagens com softwares, como Adobe® Photoshop®, Corel® Paint Shop Pro®, Microsoft® Paint e G.I.M.P. (`gimp.org`).

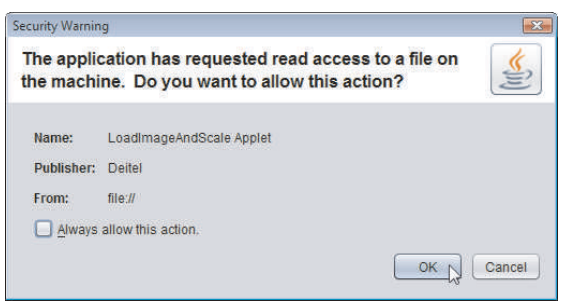
O applet da Figura 24.1 utiliza o Java Web Start e o JNLP `FileOpenService` (pacote `javax.jnlp`) para permitir ao usuário selecionar uma imagem, exibi-la e redimensioná-la. Depois de o usuário selecionar uma imagem, o applet obtém os bytes do arquivo e os passa para o

construtor `ImageIcon` (pacote `javax.swing`) para criar a imagem que será exibida. Os construtores da classe `ImageIcon` podem receber argumentos de vários formatos diferentes, incluindo um array `byte` que contém os bytes de uma imagem, uma **`Image`** (pacote `java.awt`) já carregada na memória, ou uma `String` ou um `URL` representando a localização da imagem. O Java suporta vários formatos de imagem, incluindo **Graphics Interchange Format** (**GIF**), **Joint Photographic Experts Group** (**JPEG**) e **Portable Network Graphics** (**PNG**). Os nomes de arquivo desses tipos geralmente terminam com a extensão **`.gif`**, **`.jpg`** (ou **`.jpeg`**) e **`.png`**, respectivamente.

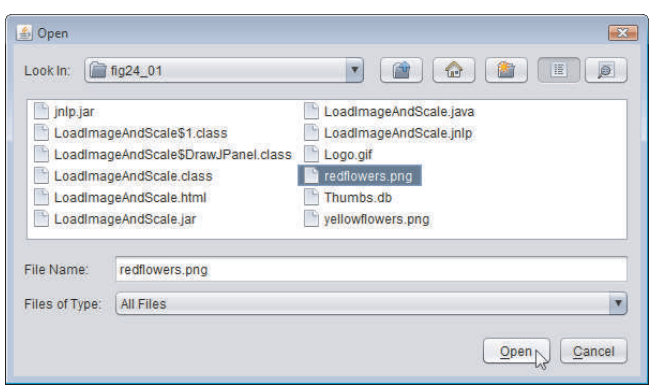
```
1  // Figura 24.1: LoadImageAndScale.java
2  // Carregando, exibindo e redimensionando uma imagem em um applet.
3  import java.awt.BorderLayout;
4  import java.awt.Graphics;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.jnlp.FileContents;
8  import javax.jnlp.FileOpenService;
9  import javax.jnlp.ServiceManager;
10 import javax.swing.ImageIcon;
11 import javax.swing.JApplet;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JOptionPane;
16 import javax.swing.JPanel;
17 import javax.swing.JTextField;
18
19 public class LoadImageAndScale extends JApplet
20 {
21    private ImageIcon image; // referencia imagem para exibição
22    private JPanel scaleJPanel; // JPanel contendo o selecionador de escala
23    private JLabel percentJLabel; // rótulo para JTextField
24    private JTextField scaleInputJTextField; // obtém a entrada de usuário
25    private JButton scaleChangeJButton; // inicia o redimensionamento de imagem
26    private double scaleValue = 1.0;  // redimensiona porcentagem para imagem
27
28    // carrega a imagem quando o applet é carregado
29    public void init()
30    {
31       scaleJPanel = new JPanel();
32       percentJLabel = new JLabel( "scale percent:" );
33       scaleInputJTextField = new JTextField( "100" );
34       scaleChangeJButton = new JButton( "Set Scale" );
35
36       // adiciona componentes e coloca scaleJPanel na região NORTH do applet
37       scaleJPanel.add( percentJLabel );
38       scaleJPanel.add( scaleInputJTextField );
39       scaleJPanel.add( scaleChangeJButton );
40       add( scaleJPanel, BorderLayout.NORTH );
41
42       // registra a rotina de tratamento de evento para scaleChangeJButton
43       scaleChangeJButton.addActionListener(
44          new ActionListener()
45          {
46             // quando JButton é pressionado, configura scaleValue e redesenha
47             public void actionPerformed( ActionEvent e )
48             {
49                scaleValue = Double.parseDouble(
50                   scaleInputJTextField.getText() ) / 100.0;
51                repaint(); // faz com que a imagem seja reexibida na nova escala
52             } // fim do método actionPerformed
53          } // fim da classe interna anônima
54       ); // fim da chamada para addActionListener
55
56       // utiliza serviços JNLP para abrir um arquivo de imagem que o usuário seleciona
57       try
58       {
59          // obtém uma referência ao FileOpenService
60          FileOpenService fileOpenService =
61             (FileOpenService) ServiceManager.lookup(
62                "javax.jnlp.FileOpenService" );
```

```
63
64             // obtém conteúdo de arquivo a partir do FileOpenService
65             FileContents contents =
66                fileOpenService.openFileDialog( null, null );
67
68             // array de bytes para armazenar dados da imagem
69             byte[] imageData = new byte[ (int) contents.getLength() ];
70             contents.getInputStream().read( imageData ); // lê bytes da imagem
71             image = new ImageIcon( imageData ); // cria a imagem
72
73             // se a imagem for carregada com sucesso, cria e adiciona DrawJPanel
74             add( new DrawJPanel(), BorderLayout.CENTER );
75          } // fim do try
76          catch( Exception e )
77          {
78             e.printStackTrace();
79          } // fim do catch
80       } // fim do método init
81
82       // DrawJPanel utilizado para exibir a imagem carregada
83       private class DrawJPanel extends JPanel
84       {
85          // exibe a imagem
86          public void paintComponent( Graphics g )
87          {
88             super.paintComponent( g );
89
90             // os seguintes valores são utilizados para centralizar a imagem
91             double spareWidth =
92                getWidth() - scaleValue * image.getIconWidth();
93             double spareHeight =
94                getHeight() - scaleValue * image.getIconHeight();
95
96             // desenha imagem com a largura e a altura redimensionada
97             g.drawImage( image.getImage(),
98                (int) ( spareWidth ) / 2, (int) ( spareHeight ) / 2,
99                (int) ( image.getIconWidth() * scaleValue ),
100               (int) ( image.getIconHeight() * scaleValue ), this );
101         } // fim do método paint
102      } // fim da classe DrawJPanel
103   } // fim da classe LoadImageAndScale
```

(a) Diálogo de segurança do Java Web Start que aparece porque esse applet está pedindo acesso a um arquivo no computador local.



(b) Diálogo **Open** que aparece se o usuário clicar em **OK** no diálogo de segurança



(c) Dimensionando a imagem.

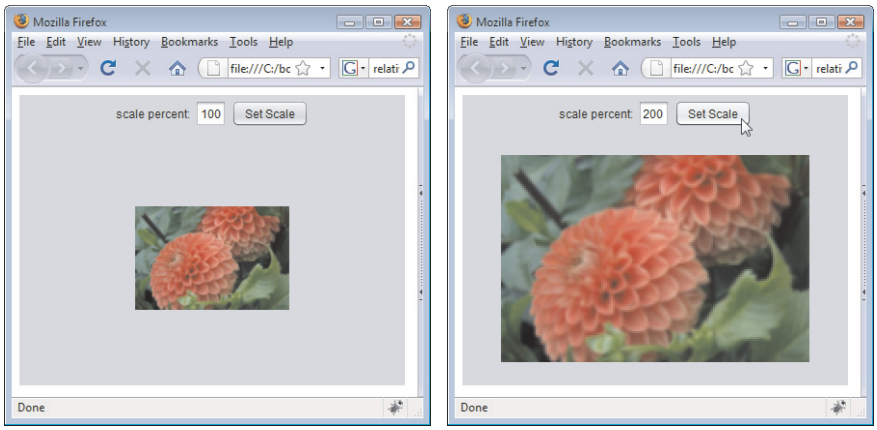


Figura 24.1 | Carregando, exibindo e dimensionando uma imagem em um applet.

Configurando a GUI e a rotina de tratamento de evento `JBUTTON`

O método `init` do applet (linhas 29–80) configura a GUI e uma rotina de tratamento de evento. Ele também utiliza serviços JNLP para permitir ao usuário selecionar uma imagem para exibição a partir do computador local. A linha 31 cria o `JPanel` que conterá o `JLabel`, `JTextField` e `JButton` criados nas linhas 32–34. As linhas 37–39 adicionam esses componentes ao `FlowLayout` padrão do `JPANEL`. A linha 40 coloca esse `JPanel` na região `NORTH` do `BorderLayout` padrão do `JApplet`.

As linhas 43–54 criam a rotina de tratamento de evento para `scaleChangeJButton`. Quando o usuário clica nesse `JButton`, as linhas 49–50 obtêm a entrada do usuário do `scaleInputJTextField` e a dividem por `100.0` a fim de calcular a porcentagem de escala e atribuir o resultado a `scaleValue`. Esse valor será utilizado em cálculos posteriores para redimensionar a imagem. Por exemplo, se o usuário inserir `50`, o valor da escala será `0.5` e a imagem será exibida com a metade do tamanho original. A linha 51 então redesenha o applet para exibir a imagem em sua nova escala.

Abrindo o arquivo de imagem com `FileOpenService` do JNLP

Como mencionamos na Seção 23.7, com a permissão do usuário, os programas Java Web Start podem acessar o sistema de arquivos local por meio das JNLP APIs do pacote `javax.jnlp`. Nesse exemplo, gostaríamos que o usuário selecionasse uma imagem do computador local para exibir no applet. (Fornecemos duas imagens no diretório desse exemplo com o código-fonte.) Você pode utilizar o `FileOpenService` do JNLP para solicitar acesso limitado ao sistema de arquivos local.

As linhas 7–9 importam as interfaces e a classe de que precisamos para utilizar o `FileOpenService`. As linhas 60–62 utilizam o método `static` **`lookup`** da classe JNLP **`ServiceManager`** para obter uma referência ao `FileOpenService`. O JNLP fornece vários serviços, por isso esse método retorna uma referência `Object`, da qual você deve fazer coerção (converter) para o tipo apropriado. Em seguida, as linhas 65–66 utilizam o método **`openFileDialog`** do `FileOpenService` para exibir um diálogo de seleção de arquivo. O Java Web Start abre um prompt para o usuário (Figura 24.1 (a)) permitir a solicitação do applet de obter acesso ao sistema de arquivos local. Se o usuário permitir, o diálogo **Open** (Figura 24.1 (b)) é exibido. O método `openFileDialog` tem dois parâmetros — uma `String` para sugerir que um diretório seja aberto e um array `String` de extensões de arquivo aceitáveis (como `"png"` e `"jpg"`). Para simplificar, esse programa não faz o uso de nenhum desses parâmetros, em vez disso passa valores `null`, que exibem um diálogo de seleção de arquivo para abrir o diretório padrão do usuário e permitir que qualquer tipo de arquivo seja selecionado.

Quando o usuário seleciona um arquivo de imagem e clica no botão **Open** do diálogo, o método `openFileDialog` retorna um objeto `FileContents`, que, por razões de segurança não dá ao programa acesso à localização exata do arquivo no disco. Em vez disso, o programa pode obter um `InputStream` e ler os bytes do arquivo. A linha 69 cria um array `byte` no qual os dados da imagem serão armazenados. O método `FileContents` **`getLength`** retorna o número de bytes (como um `long`) no arquivo. A linha 70 obtém o `InputStream`, depois invoca o seu método **`read`** para preencher o array `imageData` `byte`. A linha 71 cria um `ImageIcon` utilizando o array `byte` como a fonte dos dados da imagem. Por fim, a linha 74 adiciona um novo `DrawJPanel` ao `CENTER` do `BorderLayout` do applet. Quando o applet é exibido, os métodos `paintComponent` de seus componentes são chamados, o que faz com que o `DrawJPanel` exiba a imagem. Você pode aprender mais sobre as JNLP APIs em `java.sun.com/javase/6/docs/jre/api/javaws/jnlp`.

Exibindo a imagem com o método `paintComponent` da classe `DrawJPanel`

Para separar a GUI da área em que a imagem é exibida, utilizamos uma subclasse de `JPanel` chamada `DrawJPanel` (linhas 83–102). Seu método `paintComponent` (linhas 86–101) exibe a imagem. Gostaríamos de centralizar a imagem em `DrawJPanel`, portanto, as linhas 91–94 calculam a diferença entre a largura do `DrawJPanel` e a da imagem redimensionada e, depois, a altura do `DrawJPanel` e a da

imagem redimensionada. Os métodos **`getWidth`** e **`getHeight`** da subclasse `DrawJPanel` (herdados indiretamente da classe `Component`) retornam a largura e a altura de `DrawJPanel`, respectivamente. Os métodos **`getIconWidth`** e **`getIconHeight`** retornam a altura e a largura da imagem, respectivamente. O `scaleValue` é configurado como `1.0` por padrão (linha 26), e é alterado quando o usuário clica no `JButton` Set Scale.

As linhas 97–100 utilizam um dos métodos **`drawImage`** sobrecarregados da classe `Graphics` para exibir uma versão redimensionada do `ImageIcon`. O primeiro argumento invoca o método **`getImage`** do `ImageIcon` para obter a `Image` a ser desenhada. O segundo e terceiro argumentos representam as coordenadas do canto superior esquerdo da `Image` em relação ao canto superior esquerdo do `DrawJPanel`. O quarto e quinto argumentos especificam a largura e a altura da `Image`, respectivamente. A linha 99 redimensiona a largura da imagem invocando o método `getIconWidth` do `ImageIcon` e multiplicando seu valor de retorno pelo `scaleValue`. De modo semelhante, a linha 100 redimensiona a altura da imagem invocando o método `getIconHeight` da classe `ImageIcon` e multiplicando seu valor de retorno pelo `scaleValue`. O último argumento é uma referência a um **`ImageObserver`** — uma interface implementada pela classe `Component`. Como a classe `DrawJPanel` estende `Component` indiretamente, um `DrawJPanel` *é um* `ImageObserver`. Esse argumento é importante ao exibir imagens grandes que exigem bastante tempo para carregar (ou download da Internet). É possível que um programa tente exibir a imagem antes que tenha sido carregada (ou feito o seu download) completamente. Enquanto `Image` é carregada, `ImageObserver` recebe notificações e atualiza a imagem na tela conforme necessário. Nesse exemplo, as imagens estão sendo carregadas a partir do computador do usuário, portanto, é provável que a imagem inteira seja imediatamente exibida.

Compilando o applet

Compilar e executar esse applet requer o arquivo `jnlp.jar` que contém as JNLP APIs. Esse arquivo pode ser encontrado no diretório de instalação do JDK sob os diretórios

```
sample
   jnlp
      servlet
```

Para compilar o applet, utilize o seguinte comando:

```
javac -classpath CaminhoParaOArquivoJnlpJar LoadImageAndScale.java
```

onde *CaminhoParaOArquivoJnlpJar* inclui tanto o caminho como o nome de arquivo `jnlp.jar`. Por exemplo, no nosso computador Windows Vista, o *CaminhoParaOArquivoJnlpJar* é

```
"C:\Program Files\Java\jdk1.6.0_11\sample\jnlp\servlet\jnlp.jar"
```

Empacotando o applet para usar com Java Web Start

Para empacotar o applet para uso com o Java Web Start, você deve criar um arquivo JAR que contenha o código do applet e o arquivo `jnlp.jar`. Para fazer isso, utilize o comando

```
jar cvf LoadImageAndScale.jar *.class CaminhoParaOArquivoJnlpJar
```

onde *CaminhoParaOArquivoJnlpJar* inclui tanto o caminho como o nome de arquivo `jnlp.jar`. Isso colocará todos os arquivos `.class` do applet e uma cópia do arquivo `jnlp.jar` no novo arquivo JAR `LoadImageAndScale.jar`.

Documento JNLP para o applet `LoadImageAndScale`

O documento JNLP na Figura 24.2 é semelhante ao introduzido na Figura 23.12. O único recurso novo nesse documento é que o elemento `resources` (linhas 10–14) contém um segundo elemento `jar` (linha 13) que referencia o arquivo `jnlp.jar`, que está incorporado no arquivo `LoadImageAndScale.jar`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp codebase="file:." href="LoadImageAndScale.jnlp">
3
4     <information>
5        <title>LoadImageAndScale Applet</title>
6        <vendor>Deitel</vendor>
7        <offline-allowed/>
8     </information>
9
10    <resources>
11       <java version="1.6+"/>
12       <jar href="LoadImageAndScale.jar" main="true"/>
13       <jar href="jnlp.jar"/>
14    </resources>
```

```
15
16       <applet-desc
17          name="LoadImageAndScale"
18          main-class="LoadImageAndScale"
19          width="400"
20          height="300">
21       </applet-desc>
22    </jnlp>
```

Figura 24.2 | Documento JNLP para o applet `LoadImageAndScale`.

Tornando o applet arrastável fora da janela do navegador

O documento XHTML na Figura 24.3 carrega o applet em um navegador da Web. Nesse exemplo, utilizamos um elemento `applet` para especificar a classe do applet e fornecer dois elementos `param` entre os tags do elemento `applet`. O primeiro (linha 4) especifica que esse applet deve ser **arrastável**. Isto é, o usuário pode manter a tecla *Alt* e utilizar o mouse para arrastar o applet fora da janela do navegador. O applet então continuará executando, mesmo se o navegador for fechado. Clicar na caixa de fechamento do applet quando ele estiver executando fora do navegador faz com que o applet volte para a janela de navegador se ela ainda estiver aberta, ou termine, caso contrário. O segundo elemento `param` mostra uma maneira alternativa de especificar o arquivo JNLP que carrega um applet. Discutiremos parâmetros de applets em mais detalhes na Seção 27.2.

```
1  <html>
2  <body>
3  <applet code="LoadImageAndScale.class" width="400" height="300">
4      <param name="draggable" value="true">
5      <param name="jnlp_href" value="LoadImageAndScale.jnlp">
6  </applet>
7  </body>
8  </html>
```

Figura 24.3 | O documento XHTML para carregar o applet `LoadImageAndScale` o torna arrastável para fora da janela de navegador.

24.3 Animação de uma série de imagens

A seguir, animamos uma série de imagens que estão armazenadas em um array de `ImageIcons`. Nesse exemplo, utilizamos o JNLP `FileOpenService` para permitir ao usuário escolher um grupo de imagens que serão animadas exibindo uma imagem por vez em intervalos de 50 milissegundos. A animação apresentada nas figuras 24.4–24.5 é implementada utilizando uma subclasse de `JPanel` chamada `LogoAnimatorJPanel` (Figura 24.4) que pode ser anexada a uma janela de aplicativo ou um `JApplet`. A classe `LogoAnimator` (Figura 24.5) declara um método `main` (linhas 8–20 da Figura 24.5) para executar a animação como um aplicativo. O método `main` declara uma instância da classe `JFrame` e anexa um objeto `LogoAnimatorJPanel` ao `JFrame` para exibir a animação.

```
1  // Figura 24.4: LogoAnimatorJPanel.java
2  // Animando uma série de imagens.
3  import java.awt.Dimension;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.awt.Graphics;
7  import javax.jnlp.FileContents;
8  import javax.jnlp.FileOpenService;
9  import javax.jnlp.ServiceManager;
10 import javax.swing.ImageIcon;
11 import javax.swing.JPanel;
12 import javax.swing.Timer;
13
14 public class LogoAnimatorJPanel extends JPanel
15 {
16    protected ImageIcon images[]; // array de imagens
17    private int currentImage = 0; // índice de imagem atual
18    private final int ANIMATION_DELAY = 50; // retardo em milissegundos
19    private int width; // largura da imagem
```

```
20       private int height; // altura da imagem
21
22       private Timer animationTimer; // O timer guia a animação
23
24       // construtor inicializa LogoAnimatorJPanel carregando imagens
25       public LogoAnimatorJPanel()
26       {
27          try
28          {
29             // obtém a referência para FileOpenService
30             FileOpenService fileOpenService =
31                (FileOpenService) ServiceManager.lookup(
32                   "javax.jnlp.FileOpenService" );
33
34             // exibe diálogo que permite que o usuário selecione vários arquivos
35             FileContents[] contents =
36                fileOpenService.openMultiFileDialog( null, null );
37
38             // cria o array para armazenar referências de ImageIcons
39             images = new ImageIcon[ contents.length ];
40
41             // carrega as imagens selecionadas
42             for ( int count = 0; count < images.length; count++ )
43             {
44                // cria o array de bytes para armazenar dados de uma imagem
45                byte[] imageData =
46                    new byte[ (int) contents[ count ].getLength() ];
47
48                // obtém dados da imagem e cria a imagem
49                contents[ count ].getInputStream().read( imageData );
50                images[ count ] = new ImageIcon( imageData );
51             } // for final
52
53             // esse exemplo assume que todas as imagens têm a mesma largura e altura
54             width = images[ 0 ].getIconWidth();   // obtém a largura de ícone
55             height = images[ 0 ].getIconHeight(); // obtém a altura de ícone
56          } // fim do try
57          catch( Exception e )
58          {
59             e.printStackTrace();
60          } // fim do catch
61       } // fim do construtor LogoAnimatorJPanel
62
63       // exibe a imagem atual
64       public void paintComponent( Graphics g )
65       {
66          super.paintComponent( g ); // chama a superclasse paintComponent
67
68          images[ currentImage ].paintIcon( this, g, 0, 0 );
69
70          // configura a próxima imagem a ser desenhada apenas se o timer estiver executando
71          if (animationTimer.isRunning() )
72             currentImage = ( currentImage + 1 ) % images.length;
73       } // fim do método paintComponent
74
75       // inicia a animação ou reinicia se a janela for reexibida
76       public void startAnimation()
77       {
78          if ( animationTimer == null )
79          {
80             currentImage = 0; // exibe a primeira imagem
81
82             // cria o timer
83             animationTimer =
84                new Timer( ANIMATION_DELAY, new TimerHandler() );
85
86             animationTimer.start(); // inicia o timer
87          } // fim do if
88          else // animationTimer já existe, reinicia animação
89          {
```

```
90            if ( ! animationTimer.isRunning() )
91               animationTimer.restart();
92         } // fim de else
93      } // fim do método startAnimation
94
95      // para o timer da animação
96      public void stopAnimation()
97      {
98         animationTimer.stop();
99      } // fim do método stopAnimation
100
101     // retorna o tamanho mínimo de animação
102     public Dimension getMinimumSize()
103     {
104        return getPreferredSize();
105     } // fim do método getMinimumSize
106
107     // retorna tamanho preferido da animação
108     public Dimension getPreferredSize()
109     {
110        return new Dimension( width, height );
111     } // fim do método getPreferredSize
112
113     // classe interna para tratar eventos de ação do Timer
114     private class TimerHandler implements ActionListener
115     {
116        // responde ao evento do Timer
117        public void actionPerformed( ActionEvent actionEvent )
118        {
119           repaint(); // pinta o animator novamente
120        } // fim do método actionPerformed
121     } // fim da classe TimerHandler
122  } // fim da classe LogoAnimatorJPanel
```

Figura 24.4 | Animando uma série de imagens.

```
1  // Figura 24.5: LogoAnimator.java
2  // Exibindo imagens animadas em um JFrame.
3  import javax.swing.JFrame;
4
5  public class LogoAnimator
6  {
7     // executa a animação em um JFrame
8     public static void main( String args[] )
9     {
10       LogoAnimatorJPanel animation = new LogoAnimatorJPanel();
11
12       JFrame window = new JFrame( "Animator test" ); // configura a janela
13       window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
14       window.add( animation ); // adiciona painel ao frame
15
16       window.pack();  // aumenta a janela apenas o suficiente para sua GUI
17       window.setVisible( true );   // exibe a janela
18
19       animation.startAnimation();  // inicia a animação
20    } // fim de main
21 } // fim da classe LogoAnimator
```

Figura 24.5 | Exibindo imagens animadas em um `JFrame`.

Classe LogoAnimatorPanel

A classe LogoAnimatorJPanel (Figura 24.4) mantém um array de ImageIcons (declarados na linha 16) que são carregados no construtor (linhas 25–61). O construtor começa utilizando o método **openMultiFileDialog** do FileOpenService JNLP para exibir um diálogo de seleção de arquivo que permite ao usuário selecionar vários arquivos ao mesmo tempo. Nomeamos nossas imagens de exemplo de tal modo que todas tenham o mesmo nome básico ("deitel") seguido por um número de dois dígitos de 00-29. Isso assegura que as imagens estão na ordem adequada para a animação. Como no primeiro exemplo deste capítulo, primeiro o usuário é solicitado a dar a permissão, e então o diálogo Open aparece se a permissão tiver sido concedida. O método FileOpenService openMultiFileDialog aceita os mesmos argumentos do método openFileDialog, mas retorna um array de objetos FileContents representando o conjunto de arquivos selecionados pelo usuário. Ao executar esse aplicativo, navegue até a pasta que contém as imagens que você deseja utilizar e selecione-as. Se quiser, você pode utilizar as 30 imagens que fornecemos no subdiretório chamado images desse exemplo.

A linha 39 cria o array de ImageIcons, então as linhas 42–51 preenchem o array criando um array byte (linhas 45–46) para os dados da imagem atual, lendo os bytes da imagem no array (linha 49) e criando um objeto ImageIcon do array byte. As linhas 54–55 determinam a largura e a altura da animação a partir do tamanho da primeira imagem no array images — supomos que todas as imagens tenham a mesma largura e altura.

Método startAnimation

Depois que o construtor LogoAnimatorJPanel carrega as imagens, o método main da Figura 24.5 configura a janela em que a animação aparecerá (linhas 12–17) e a linha 19 chama o método startAnimation do LogoAnimatorJPanel (declarado nas linhas 76–93 da Figura 24.4). Esse método inicia a animação do programa pela primeira vez ou reinicia a animação que o programa parou anteriormente. A animação é controlada por uma instância da classe **Timer** (do pacote javax.swing).

Quando o programa é executado pela primeira vez, o método startAnimation é chamado para começar a animação. Embora forneçamos as funcionalidades para esse método reiniciar a animação se ela tiver sido parada, o exemplo não chama o método para esse propósito. Mas adicionamos as funcionalidades para o caso de o leitor escolher se acrescenta componentes GUI que permitam ao usuário iniciar e parar a animação.

Um Timer gera ActionEvents a um intervalo fixo em milissegundos (normalmente especificado como um argumento para o construtor do Timer) e notifica todos seus ActionListeners sempre que ocorrer um ActionEvent. A linha 78 determina se a referência animationTimer de Timer é null. Se for, o método startAnimation está sendo chamado pela primeira vez, e um Timer precisa ser criado para que a animação possa começar. A linha 80 configura currentImage como 0, o que indica que a animação deve iniciar com o primeiro elemento do array images. As linhas 83–84 atribuem um novo objeto Timer a animationTimer. O construtor Timer recebe dois argumentos — o retardo em milissegundos (ANIMATION_DELAY é 50, como especificado na linha 18) e o ActionListener que responderá aos ActionEvents do Timer. Quanto ao segundo argumento, um objeto da classe TimerHandler é criado. Essa classe, que implementa ActionListener, é declarada nas linhas 114–121. A linha 86 chama o método **start** do objeto Timer para iniciar o Timer. Uma vez iniciado, animationTimer gerará um ActionEvent a cada 50 milissegundos e chamará a rotina de tratamento de evento actionPerformed de Timer (linhas 117–120). A linha 119 chama o método repaint de LogoAnimatorJPanel para agendar uma chamada para o método paintComponent de LogoAnimatorJPanel (linhas 64–73). Lembre-se de que qualquer subclasse de JComponent que desenhe deve fazer isso em seu método paintComponent. Lembre-se de que a primeira instrução em qualquer método paintComponent deve ser uma chamada ao método paintComponent da superclasse, para assegurar que os componentes Swing sejam exibidos corretamente.

Se a animação já tiver iniciado antes, nosso Timer foi criado e a condição na linha 78 é avaliada como false. O programa continua com as linhas 90–91, que reiniciam a animação que o programa parou anteriormente. A condição if na linha 90 utiliza o método Timer **isRunning** para determinar se o Timer está executando (isto é, gerando eventos). Se não estiver executando, a linha 91 chama o método Timer **restart** para indicar que Timer deve começar a gerar eventos novamente. Uma vez que isso ocorre, o método actionPerformed (o handler de evento do Timer) é novamente chamado em intervalos regulares.

Método paintComponent

A linha 68 chama o método **paintIcon** do ImageIcon para exibir a imagem armazenada no elemento currentImage no array. Os argumentos representam o Component em que desenhar (this), o objeto Graphics que realiza o desenho (g) e as coordenadas do canto superior esquerdo da imagem. As linhas 71–72 determinam se o animationTimer está executando e, se estiver, preparam a próxima imagem a ser exibida incrementando currentImage por 1. O cálculo restante assegura que o valor de currentImage está configurado como 0 (para repetir a sequência de animação) quando ele é incrementado depois do último índice de elemento no array. A instrução if assegura que a mesma imagem seja exibida se paintComponent for chamado enquanto o Timer é parado. Isso pode ser útil se for fornecida uma GUI que permita ao usuário iniciar e parar a animação. Por exemplo, se a animação for interrompida e o usuário cobri-la com outra janela, e depois descobri-la, o método paintComponent será chamado. Nesse caso, não queremos que a animação exiba a próxima imagem (porque a animação foi interrompida). Simplesmente queremos que a janela exiba a mesma imagem até a animação ser reiniciada.

Método `stopAnimation`

O método `stopAnimation` (linhas 96–99) para a animação chamando o método `Timer` **`stop`** para indicar que o `Timer` deve parar de gerar os eventos. Isso impede que `actionPerformed` chame `repaint` para iniciar a pintura da próxima imagem no array. Assim como com o exemplo de reiniciar a animação, esse exemplo define, mas não utiliza, o método `stopAnimation`. Fornecemos esse método para propósitos de demonstração, ou para permitir ao usuário modificar esse exemplo a fim de parar e reiniciar a animação.

Observação de engenharia de software 24.1

Ao criar uma animação para uso em um applet, forneça um mecanismo para desativar a animação quando o usuário navegar para uma nova página Web diferente daquela em que o applet de animação reside.

Métodos `getPreferredSize` e `getMinimumSize`

Estendendo a classe `JPanel`, estamos criando um novo componente GUI. Portanto, devemos assegurar que ele funciona como os outros componentes para propósitos de layout. Em geral, os gerenciadores de layout utilizam o método **`getPreferredSize`** de um componente (herdado da classe `java.awt.Component`) para determinar a largura e a altura preferidas do componente. Se um novo componente tiver uma largura e altura preferidas, ele deve anular o método `getPreferredSize` (linhas 108–111) para retornar essa largura e altura como um objeto da classe **`Dimension`** (pacote `java.awt`). A classe `Dimension` representa a largura e altura de um componente GUI. Nesse exemplo, as imagens que fornecemos têm 160 pixels de largura e 80 pixels de altura, portanto, o método `getPreferredSize` retornaria um objeto `Dimension` contendo os números 160 e 80 (se você utilizasse essas imagens).

Observações sobre a aparência e comportamento 24.1

O tamanho padrão de um objeto `JPanel` é de 10 pixels de largura e 10 pixels de altura.

Observações sobre a aparência e comportamento 24.2

Ao subclassificar `JPanel` (ou qualquer outro `JComponent`), anule o método `getPreferredSize` se o novo componente precisar ter uma largura e uma altura específicas preferidas.

As linhas 102–105 sobrescrevem o método **`getMinimumSize`**. Esse método determina a largura e altura mínima do componente. Como com o método `getPreferredSize`, novos componentes devem sobrescrever o método `getMinimumSize` (também herdado da classe `Component`). O método `getMinimumSize` simplesmente chama `getPreferredSize` (uma prática de programação comum) para indicar que o tamanho mínimo e o preferido são os mesmos. Alguns gerenciadores de layout ignoram as dimensões especificadas por esses métodos. Por exemplo, as regiões `NORTH` e `SOUTH` de um `BorderLayout` utilizam apenas a altura preferida do componente.

Observações sobre a aparência e comportamento 24.3

Se um novo componente GUI tiver uma largura e altura mínimas (isto é, se as pequenas dimensões resultarem em um componente ineficiente na tela), sobrescreva o método `getMinimumSize` para retornar a largura e a altura mínimas como uma instância da classe `Dimension`.

Observações sobre a aparência e comportamento 24.4

Para muitos componentes GUI o método `getMinimumSize` é implementado para retornar o resultado de uma chamada ao método `getPreferredSize` desse componente.

Compilando o aplicativo

A compilação e a execução desse aplicativo requerem o arquivo `jnlp.jar` que contém as JNLP APIs. Para compilar o aplicativo, utilize o seguinte comando:

```
javac -classpath CaminhoParaOArquivoJnlpJar *.java
```

onde *CaminhoParaOArquivoJnlpJar* inclui tanto o caminho como o nome de arquivo `jnlp.jar`.

Empacotando o aplicativo para uso com Java Web Start

Para empacotar o aplicativo para uso com Java Web Start, você deve criar um arquivo JAR que contém o código e o arquivo `jnlp.jar` do applet. Para fazer isso, utilize o comando

```
jar cvf LogoAnimator.jar *.class CaminhoParaOArquivoJnlpJar
```

onde *CaminhoParaOArquivoJnlpJar* inclui tanto o caminho como o nome de arquivo `jnlp.jar`.

Documento JNLP para o applet `LoadImageAndScale`

O documento JNLP da Figura 24.6 é semelhante ao da Figura 24.2. O único recurso novo nesse documento é o elemento `application-desc` (linhas 16–19), que especifica o nome do aplicativo e sua classe principal. Para executar esse aplicativo, utilize o comando

```
javaws LogoAnimator.jnlp
```

Lembre-se de que você também pode executar aplicativos Java Web Start via um link em uma página Web, como mostramos na Figura 23.13.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp codebase="file:." href="LogoAnimator.jnlp">
3
4     <information>
5        <title>LogoAnimator</title>
6        <vendor>Deitel</vendor>
7        <offline-allowed/>
8     </information>
9
10    <resources>
11       <java version="1.6+"/>
12       <jar href="LogoAnimator.jar" main="true"/>
13       <jar href="jnlp.jar"/>
14    </resources>
15
16    <application-desc
17       name="LogoAnimator"
18       main-class="LogoAnimator">
19    </application-desc>
20 </jnlp>
```

Figura 24.6 | Documento JNLP para o applet `LoadImageAndScale`.

24.4 Mapas de imagem

Os **mapas de imagem** são comumente utilizados para criar páginas Web interativas. Um mapa de imagem é uma imagem com **áreas ativas (hot areas)** em que o usuário pode clicar para realizar uma tarefa, como carregar uma página Web diferente em um navegador. Quando o usuário posiciona o ponteiro do mouse sobre uma área ativa, normalmente uma mensagem descritiva aparece na área de status do navegador ou em uma dica de ferramenta.

A Figura 24.7 carrega uma imagem contendo vários dos ícones de dica de programação utilizados neste livro. O programa permite ao usuário posicionar o ponteiro do mouse sobre um ícone para exibir uma mensagem descritiva associada com ele. O handler de evento `mouseMoved` (linhas 39–43) aceita as coordenadas do mouse e as passa para o método `translateLocation` (linhas 58–69). O método `translateLocation` testa as coordenadas para determinar o ícone sobre o qual o mouse foi posicionado quando o evento `mouseMoved` ocorreu — o método então retorna uma mensagem que indica o que o ícone representa. Essa mensagem é exibida na barra de status do contêiner de applets utilizando o método **`showStatus`** da classe `Applet`.

```
1  // Figura 24.7: ImageMap.java
2  // Mapa de imagem.
3  import java.awt.event.MouseAdapter;
4  import java.awt.event.MouseEvent;
5  import java.awt.event.MouseMotionAdapter;
6  import java.awt.Graphics;
7  import javax.swing.ImageIcon;
```

```
8  import javax.swing.JApplet;
9
10 public class ImageMap extends JApplet
11 {
12    private ImageIcon mapImage;
13
14    private static final String captions[] = { "Common Programming Error",
15       "Good Programming Practice", "Look-and-Feel Observation",
16       "Performance Tip", "Portability Tip",
17       "Software Engineering Observation", "Error-Prevention Tip" };
18
19    // configura ouvintes de mouse
20    public void init()
21    {
22       addMouseListener(
23
24          new MouseAdapter() // classe interna anônima
25          {
26             // indica quando o ponteiro do mouse sai da área do applet
27             public void mouseExited( MouseEvent event )
28             {
29                showStatus( "Pointer outside applet" );
30             } // mouseExited fim do método
31          } // fim da classe interna anônima
32       ); // fim da chamada para addMouseListener
33
34       addMouseMotionListener(
35
36          new MouseMotionAdapter() // classe interna anônima
37          {
38             // determina ícone sobre o qual o mouse aparece
39             public void mouseMoved( MouseEvent event )
40             {
41                showStatus( translateLocation(
42                   event.getX(), event.getY() ) );
43             } // fim do método mouseMoved
44          } // fim da classe interna anônima
45       ); // fim da chamada para addMouseMotionListener
46
47       mapImage = new ImageIcon( "icons.png" ); // obtém a imagem
48    } // fim do método init
49
50    // exibe mapImage
51    public void paint( Graphics g )
52    {
53       super.paint( g );
54       mapImage.paintIcon( this, g, 0, 0 );
55    } // fim do método paint
56
57    // retorna legenda de dica com base nas coordenadas do mouse
58    public String translateLocation( int x, int y )
59    {
60       // se coordenar fora da imagem, retorna imediatamente
61       if ( x >= mapImage.getIconWidth() || y >= mapImage.getIconHeight() )
62          return "";
63
64       // determina o número de ícone (0 - 6)
65       double iconWidth = ( double ) mapImage.getIconWidth() / 7.0;
66       int iconNumber = ( int )( ( double ) x / iconWidth );
67
68       return captions[ iconNumber ]; // retorno à legenda de ícone apropriada
69    } // fim do método translateLocation
70 } // fim da classe ImageMap
```

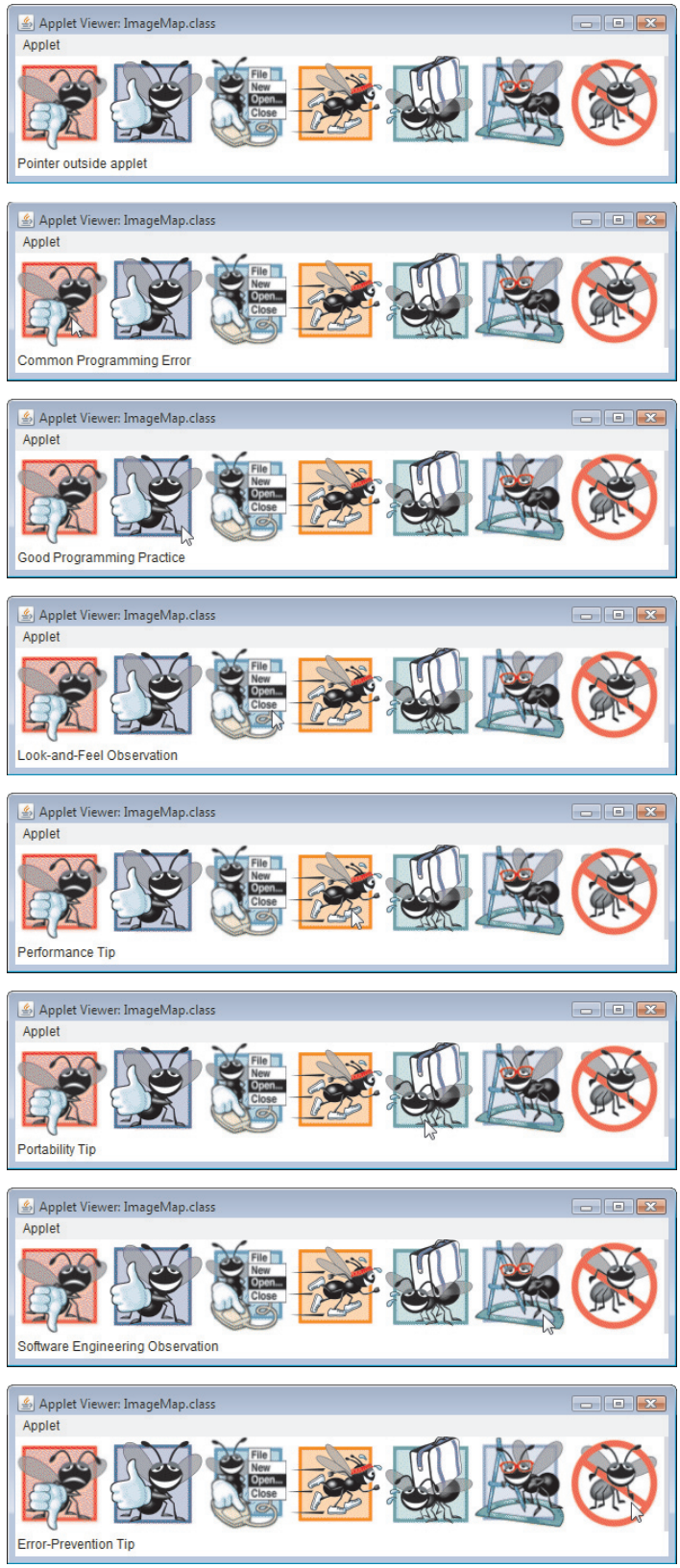


Figura 24.7 | Mapa de imagem.

Clicar no applet da Figura 24.7 não produzirá nenhuma ação. No Capítulo 27, Rede, discutimos as técnicas para carregar outra página Web em um navegador via `URL`s e a interface `AppletContext`. Utilizando essas técnicas, esse applet poderia associar cada ícone com um `URL` que o navegador exibiria quando o usuário clicasse no ícone.

24.5 Carregando e reproduzindo clipes de áudio

Programas Java podem manipular e reproduzir **clipes de áudio**. Os usuários podem capturar seus próprios clipes de áudio e muitos clipes estão disponíveis em produtos de software e Internet. Seu sistema precisa ser equipado com hardware de áudio (alto-falantes e uma placa de som) para ser capaz de reproduzir os clipes de áudio.

O Java fornece vários mecanismos para reproduzir sons em um applet. Os dois mais simples são o método **`play`** de `Applet` e o método **`play`** da interface **`AudioClip`**. As capacidades adicionais de áudio estão disponíveis nas APIs Java Media Framework e Java Sound. Se você quiser reproduzir um som uma vez em um programa, o método `Applet` `play` carrega o som e o reproduz uma vez, depois o som é marcado para coleta de lixo. O método `Applet` `play` tem duas versões:

```
public void play( URL location, String soundFileName );
public void play( URL soundURL );
```

A primeira versão carrega o clipe de áudio armazenado no arquivo `soundFileName` a partir da `location` e reproduz o som. O primeiro argumento é normalmente uma chamada ao método `getDocumentBase` ou **`getCodeBase`** do applet. O método `getDocumentBase` retorna a localização do arquivo HTML que carregou o applet. (Se o applet estiver em um pacote, o método retorna a localização do pacote ou o arquivo JAR que contém o pacote.) O método `getCodeBase` indica a localização do arquivo `.class` do applet. A segunda versão de método `play` aceita um `URL` que contém a localização e o nome de arquivo do clipe de áudio. A instrução

```
play( getDocumentBase(), "hi.au" );
```

carrega o clipe de áudio no arquivo `hi.au` e reproduz o clipe uma vez.

O **mecanismo de som** que reproduz os clipes de áudio suporta vários formatos de arquivo de áudio, incluindo **formato de arquivo Sun Audio** (extensão **.au**), **formato de arquivo Windows Wave** (extensão **.wav**), **formato de arquivo Macintosh AIFF** (extensões **.aif ou .AIFF**) e o **formato de arquivo Musical Instrument Digital Interface** (**MIDI**) (extensões **.mid** ou **.rmi**). A API Java Media Framework (JMF) e a Java Sound API suportam formatos adicionais.

O programa da Figura 24.8 demonstra carregamento e reprodução de um `AudioClip` (pacote `java.applet`). Essa técnica é mais flexível que o método `Applet` `play`. Um applet pode utilizar um `AudioClip` para armazenar o áudio para uso repetido em toda execução de um programa. O método `Applet` **`getAudioClip`** tem duas formas que aceitam os mesmos argumentos que o método `play` descrito anteriormente. O método `getAudioClip` retorna uma referência para um `AudioClip`. Um `AudioClip` tem três métodos — `play`, `loop` e `stop`. Como mencionado anteriormente, o método `play` reproduz o clipe de áudio uma vez. O método **`loop`** faz loops continuamente pelo clipe de áudio no segundo plano. O método **`stop`** termina um clipe de áudio que atualmente está reproduzindo. No programa, cada um desses métodos é associado com um botão no applet.

```
1  // Figura 24.8: LoadAudioAndPlay.java
2  // Carregando e executando um AudioClip.
3  import java.applet.AudioClip;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.FlowLayout;
9  import javax.swing.JApplet;
10 import javax.swing.JButton;
11 import javax.swing.JComboBox;
12
13 public class LoadAudioAndPlay extends JApplet
14 {
15    private AudioClip sound1, sound2, currentSound;
16    private JButton playJButton, loopJButton, stopJButton;
17    private JComboBox soundJComboBox;
18
19    // carrega a imagem quando o applet começa a executar
20    public void init()
21    {
22       setLayout( new FlowLayout() );
23
24       String choices[] = { "Welcome", "Hi" };
25       soundJComboBox = new JComboBox( choices ); // cria JComboBox
26
27       soundJComboBox.addItemListener(
28
29          new ItemListener() // classe interna anônima
30          {
```

```
31              // interrompe o som e muda para o som selecionado pelo usuário
32              public void itemStateChanged( ItemEvent e )
33              {
34                 currentSound.stop();
35                 currentSound = soundJComboBox.getSelectedIndex() == 0 ?
36                    sound1 : sound2;
37              } // fim do método itemStateChanged
38           } // fim da classe interna anônima
39        ); // fim da chamada de método addItemListener
40
41        add( soundJComboBox ); // adiciona JComboBox o applet
42
43        // configura o handler de evento de botão e botões
44        ButtonHandler handler = new ButtonHandler();
45
46        // cria o JButton Play
47        playJButton = new JButton( "Play" );
48        playJButton.addActionListener( handler );
49        add( playJButton );
50
51        // cria o JButton Loop
52        loopJButton = new JButton( "Loop" );
53        loopJButton.addActionListener( handler );
54        add( loopJButton );
55
56        // cria o JButton Stop
57        stopJButton = new JButton( "Stop" );
58        stopJButton.addActionListener( handler );
59        add( stopJButton );
60
61        // carrega sons e configura currentSound
62        sound1 = getAudioClip( getDocumentBase(), "welcome.wav" );
63        sound2 = getAudioClip( getDocumentBase(), "hi.au" );
64        currentSound = sound1;
65     } // fim do método init
66
67     // para o som quando o usuário muda de página Web
68     public void stop()
69     {
70        currentSound.stop(); // interrompe o AudioClip
71     } // fim do método stop
72
73     // classe interna private para tratar eventos de botão
74     private class ButtonHandler implements ActionListener
75     {
76        // processa, reproduz, faz loop e interrompe eventos de botão
77        public void actionPerformed( ActionEvent actionEvent )
78        {
79           if ( actionEvent.getSource() == playJButton )
80              currentSound.play(); // reproduz o AudioClip uma vez
81           else if ( actionEvent.getSource() == loopJButton )
82              currentSound.loop(); // reproduz o AudioClip continuamente
83           else if ( actionEvent.getSource() == stopJButton )
84              currentSound.stop(); // interrompe o AudioClip
85        } // fim do método actionPerformed
86     } // fim da classe ButtonHandler
87  } // fim da classe LoadAudioAndPlay
```

Figura 24.8 | Carregamento e reprodução de um `AudioClip`.

As linhas 62–63 no applet `init` utilizam o método `getAudioClip` para carregar dois arquivos de áudio — um Windows Wave (`welcome.wav`) e um Sun Audio (`hi.au`). O usuário pode selecionar qual clipe de áudio reproduzir a partir do `JComboBox` `soundJComboBox`. Observe que o método `stop` do applet é sobrescrito nas linhas 68–71. Quando o usuário alterna páginas Web, o contêiner de applets chama o método `stop` do applet. Isso permite que o applet pare de reproduzir o clipe de áudio. Caso contrário, continua a reproduzir no segundo plano — mesmo se o applet não for exibido no navegador. Isso não é necessariamente um problema, mas pode irritar o usuário se o clipe de áudio estiver fazendo um loop. O método `stop` é fornecido aqui como uma conveniência para o usuário.

Observações sobre a aparência e comportamento 24.5

Ao reproduzir clipes de áudio em um applet ou aplicativo, forneça um mecanismo para o usuário desativar o áudio.

24.6 Reproduzindo vídeo e outros tipos de mídia com o Java Media Framework

Um vídeo simples pode concisa e efetivamente transportar muitas informações. Reconhecendo o valor de introduzir capacidades multimídia extensíveis ao Java, a Sun Microsystems, a Intel e a Silicon Graphics se uniram para produzir o aplicativo Java Media Framework (JMF). Utilizando a JMF API, os programadores podem criar aplicativos Java que reproduzem, editam, transmitem por stream e capturam muitos tipos de mídia populares. Os recursos do JMF são bem extensos. Esta seção apresenta resumidamente alguns formatos de mídia populares e demonstra a reprodução de vídeo utilizando a JMF API.

A IBM e a Sun desenvolveram a última especificação JMF — a versão 2.0. A Sun também fornece uma implementação de referência da especificação JMF — JMF 2.1.1e — que suporta tipos de arquivo de mídia como **Microsoft Audio/Video Interleave** (`.avi`), **filmes Macromedia Flash 2** (`.swf`), **Future Splash** (`.spl`), **MPEG Layer 3 Audio** (`.mp3`), Musical Instrument Digital Interface (MIDI; extensões `.mid` ou `.rmi`), **MPEG-1 videos** (`.mpeg.mpg`), **QuickTime** `.mov`, formato de arquivo Sun Audio (extensão `.au`) e formato de arquivo AIFF da Macintosh (extensões `.aif` ou `.aiff`). Você já viu alguns desses tipos de arquivo.

Atualmente, a JMF está disponível como uma extensão separada do Java 2 Software Development Kit. O download da implementação JMF mais recente (2.1.1e) pode ser feito a partir de:

```
java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/
   download.html
```

[*Nota*: Preste atenção ao local em que você instala o Java Media Framework no computador. Para compilar e executar esse aplicativo, você deve incluir no caminho de classe o arquivo `jmf.jar` que é instalado com o Java Media Framework. Lembre-se de que você pode especificar o caminho de classe tanto com o comando `javac` quanto com o comando `java` por meio da opção de linha de comando `-classpath`.]

O site Web da JMF fornece versões da JMF que tiram proveito dos recursos de desempenho de certas plataformas. Por exemplo, o JMF Windows Performance Pack fornece extensa mídia e suporte de dispositivo para programas Java que executam em plataformas Microsoft Windows. O site do JMF (`java.sun.com/javase/technologies/desktop/media/jmf/`) tem informações e recursos para programadores JMF.

Criando um Media Player simples

A JMF oferece vários mecanismos para reproduzir a mídia. O mais simples é utilizar objetos que implementam a interface **`Player`** declarada no pacote **`javax.media`**. O pacote `javax.media` e seus subpacotes contêm as classes que compõem o Java Media Framework. Para reproduzir um clipe de mídia, você deve primeiro criar um objeto `URL` que o referencia. Então passe o `URL` como um argumento para o método `static` **`createRealizedPlayer`** da classe `Manager` a fim de obter um `Player` para o clipe de mídia. A classe **`Manager`** declara os métodos de utilitários para acessar os recursos do sistema a fim de reproduzir e manipular mídia. A Figura 24.9 declara um `JPanel` que demonstra alguns desses métodos.

```
1  // Figura 24.9: MediaPanel.java
2  // JPanel que reproduz um arquivo de mídia de um URL.
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.io.IOException;
6  import java.net.URL;
7  import javax.media.CannotRealizeException;
8  import javax.media.Manager;
9  import javax.media.NoPlayerException;
10 import javax.media.Player;
11 import javax.swing.JPanel;
12
13 public class MediaPanel extends JPanel
14 {
15    public MediaPanel( URL mediaURL )
16    {
17       setLayout( new BorderLayout() ); // utiliza um BorderLayout
18
19       // Utiliza componentes leves para compatibilidade Swing
20       Manager.setHint( Manager.LIGHTWEIGHT_RENDERER, true );
21
22       try
23       {
24          // cria um player para reproduzir a mídia especificada no URL
25          Player mediaPlayer = Manager.createRealizedPlayer( mediaURL );
26
27          // obtém os componentes para o vídeo e os controles de reprodução
28          Component video = mediaPlayer.getVisualComponent();
29          Component controls = mediaPlayer.getControlPanelComponent();
30
31          if ( video != null )
32             add( video, BorderLayout.CENTER ); // adiciona o componente de vídeo
33
34          if ( controls != null )
35             add( controls, BorderLayout.SOUTH ); // adiciona os controles
36
37          mediaPlayer.start(); // inicia a reprodução do clipe de mídia
38       } // fim do try
39       catch ( NoPlayerException noPlayerException )
40       {
41          System.err.println( "No media player found" );
42       } // fim do catch
43       catch ( CannotRealizeException cannotRealizeException )
44       {
45          System.err.println( "Could not realize media player" );
46       } // fim do catch
47       catch ( IOException iOException )
48       {
49          System.err.println( "Error reading from the source" );
50       } // fim do catch
51    } // fim do construtor MediaPanel
52 } // fim da classe MediaPanel
```

Figura 24.9 | `JPanel` que reproduz um arquivo de mídia de um URL.

O construtor (linhas 15–51) configura o `JPanel` para reproduzir o arquivo de mídia especificado por um parâmetro `URL` do construtor. `MediaPanel` utiliza um `BorderLayout` (linha 17). A linha 20 invoca o método `static` **`setHint`** para configurar o flag **`Manager.LIGHTWEIGHT_RENDERER`** como `true`. Isso instrui o `Manager` a utilizar um renderizador de peso leve que é compatível com os componentes Swing leves, em oposição ao renderizador pesado padrão. No bloco `try` (linhas 22–38), a linha 25 invoca o método `static` `createRealizedPlayer` da classe `Manager` para criar e realizar um `Player` que reproduz o arquivo de mídia. Quando um `Player` é realizado, ele identifica os recursos do sistema de que ele precisa para reproduzir a mídia. Dependendo do arquivo, realizar pode ser um processo que consome recursos e tempo. O método `createRealizedPlayer` lança três exceções verificadas, **`NoPlayerException`**, **`CannotRealizeException`** e `IOException`. Uma `NoPlayerException` indica que o sistema não pôde localizar um player que pudesse reproduzir o formato de arquivo. Uma `CannotRealizeException` indica que o sistema não pôde identificar adequadamente os recursos que o arquivo de mídia precisa. Uma `IOException` indica que houve um erro durante a leitura do arquivo. Essas exceções são tratadas no bloco `catch` nas linhas 39–50.

A linha 28 invoca o método **`getVisualComponent`** de `Player` para obter um `Component` que exibe o aspecto visual (geralmente vídeo) do arquivo de mídia. A linha 29 invoca o método **`getControlPanelComponent`** de `Player` para obter um `Component` que fornece controles de reprodução e mídia. Esses componentes são atribuídos às variáveis locais `video` e `controls`, respectivamente. As instruções `if` nas linhas 31–32 e linhas 34–35 adicionam o `video` e os `controls` se eles existirem. O `video Component` é adicionado à região `CENTER` (linha 32), então ele preenche qualquer espaço disponível no `JPanel`. O `controls Component`, que é adicionado à região `SOUTH`, em geral fornece os seguintes controles:

1. Um controle deslizante de posicionamento para pular para certos pontos no clipe de mídia.
2. Um botão de pausa.
3. Um botão de volume que fornece controle de volume clicando-se à direita e uma função de mudo clicando-se à esquerda.
4. Um botão de propriedades de mídia que fornece informações detalhadas de mídia clicando-se à esquerda e controle de velocidade de projeção clicando-se à direita.

A linha 37 chama o método `Player` **`start`** para iniciar a reprodução do arquivo de mídia. As linhas 39–50 tratam as várias exceções que o `createRealizedPlayer` lança.

O aplicativo na Figura 24.10 exibe um diálogo `JFileChooser` para o usuário escolher um arquivo de mídia. Ele, então, cria um `MediaPanel` que executa o arquivo selecionado e cria um `JFrame` para exibir o `MediaPanel`.

```
1  // Figura 24.10: MediaTest.java
2  // Aplicativo de teste que cria um MediaPanel de um arquivo selecionado pelo usuário.
3  import java.io.File;
4  import java.net.MalformedURLException;
5  import java.net.URL;
6  import javax.swing.JFileChooser;
7  import javax.swing.JFrame;
8
9  public class MediaTest
10 {
11    // carrega o aplicativo
12    public static void main( String args[] )
13    {
14       // cria um chooser de arquivo
15       JFileChooser fileChooser = new JFileChooser();
16
17       // mostra diálogo de arquivo aberto
18       int result = fileChooser.showOpenDialog( null );
19
20       if ( result == JFileChooser.APPROVE_OPTION ) // usuário escolheu um arquivo
21       {
22          URL mediaURL = null;
23
24          try
25          {
26             // obtém o arquivo como URL
27             mediaURL = fileChooser.getSelectedFile().toURI().toURL();
28          } // fim do try
29          catch ( MalformedURLException malformedURLException )
30          {
31             System.err.println( "Could not create URL for the file" );
32          } // fim do catch
33
34          if ( mediaURL != null ) // exibe somente se houver um URL válido
35          {
36             JFrame mediaTest = new JFrame( "Media Tester" );
37             mediaTest.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
38
39             MediaPanel mediaPanel = new MediaPanel( mediaURL );
40             mediaTest.add( mediaPanel );
41
42             mediaTest.setSize( 300, 300 );
43             mediaTest.setVisible( true );
44          } // fim do if interno
45       } // fim do if externo
46    } // fim de main
47 } // fim da classe MediaTest
```

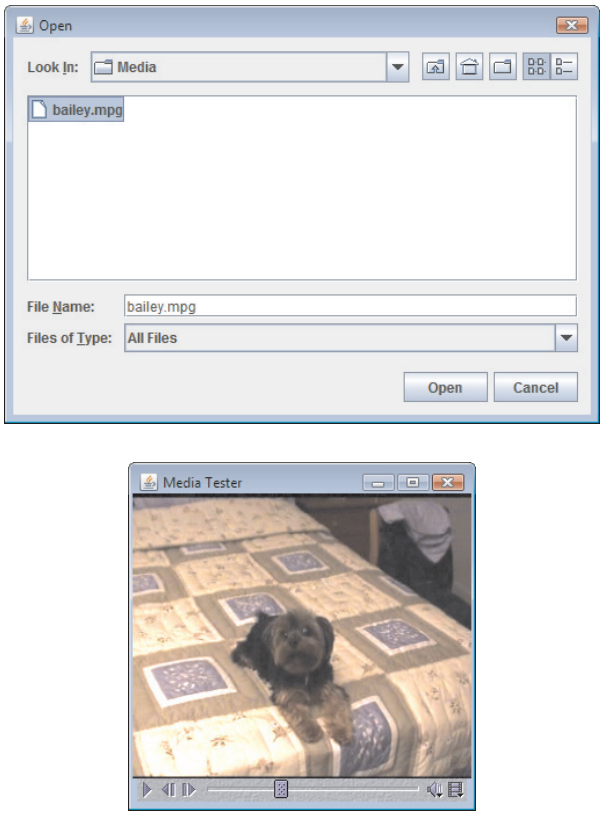


Figura 24.10 | Aplicativo de teste que cria um MediaPanel de um arquivo selecionado pelo usuário.

O método `main` (linhas 12–46) atribui um novo `JFileChooser` à variável local `fileChooser` (linha 15), mostra um diálogo de arquivo aberto (linha 18) e atribui o valor de retorno ao `result`. A linha 20 verifica `result` para determinar se o usuário escolheu um arquivo. Para criar um `Player` a fim de reproduzir o arquivo selecionado de mídia, você deve converter o objeto `File` retornado pelo `JFileChooser` em um objeto `URL`. O método **`toURI`** da classe `File` retorna um `URI` que aponta para o `File` no sistema. Então, invocamos o método **`toURL`** da classe `URI` para obter o arquivo `URL`. A instrução `try` (linhas 24–32) cria um `URL` para o arquivo selecionado e o atribui à `mediaURL`. A instrução `if` nas linhas 34–44 verifica se `mediaURL` não é `null` e cria os componentes GUI para reproduzir a mídia.

24.7 Conclusão

Neste capítulo, você aprendeu a fazer aplicativos mais interessantes incluindo som, imagens, imagens gráficas e vídeo. Introduzimos as capacidades multimídia do Java, incluindo a Java Media Framework API e a Java Sound API. Utilizou a classe `ImageIcon` para exibir e manipular imagens armazenadas em arquivos e aprendeu sobre os diferentes formatos de imagem suportados pelo Java. Usou o JNLP `FileOpenService` para permitir ao usuário de um aplicativo Java Web Start selecionar arquivos do sistema de arquivos local e utilizou fluxos para carregar o conteúdo desses arquivos para uso nos seus programas. Criou a animação exibindo uma série de imagens em uma ordem específica. Empregou mapas de imagem para tornar um aplicativo mais interativo. Em seguida, aprendeu a carregar clipes de áudio e a reproduzi-los uma vez ou em um loop contínuo. O capítulo concluiu com uma demonstração de carregar e reproduzir vídeo. No próximo capítulo, você continuará o estudo de conceitos GUI, baseando-se nas técnicas aprendidas no Capítulo 14.

24.8 Recursos da Web

`www.nasa.gov/multimedia/highlights/index.html`
A *NASA Multimedia Gallery* contém uma grande variedade de imagens, clipes de áudio e videoclipes que você pode descarregar e utilizar para testar seus programas de multimídia Java.

`commons.wikimedia.org/wiki/Main_Page`
O site *Wikimedia Commons* fornece acesso a milhões de arquivos de mídia.

`www.anbg.gov.au/anbg/index.html`
O site da Web *Australian National Botanic Gardens* fornece links para muitos sons de animais. Experimente, por exemplo, o link *Common Birds* na seção "Animals in the Gardens".

`www.thefreesite.com`
Esse site tem links para sons e clipes de arte gratuitos.

`www.soundcentral.com`
SoundCentral fornece clipes de áudio em formatos WAV, AU, AIFF e MIDI.

`www.animationfactory.com`
Animation Factory fornece milhares de animações GIF gratuitamente para uso pessoal.

`www.clipart.com`
Esse site é um serviço baseado em assinatura para imagens e sons.

`java.sun.com/developer/techDocs/hi/repository/`
O *Java look-and-feel Graphics Repository* fornece imagens projetadas para utilização em uma Swing GUI, incluindo imagens de botão de barra de ferramentas.

`www.freebyte.com/graphicprograms/`
Este guia contém links para vários programas de softwares de imagens gráficas gratuitos. O software pode ser utilizado para modificar imagens e desenhar imagens gráficas.

`graphicssoft.about.com/od/pixelbasedfreewin/`
Esse site fornece links para programas gratuitos de imagens gráficas projetados para utilização em máquinas Windows.

Referências de API multimídia do Java

`java.sun.com/javase/technologies/desktop/media/`
A página principal das Java Media APIs.

`java.sun.com/products/java-media/sound/`
A home page da *Java Sound API*. A API Java Sound fornece recursos para reproduzir e gravar áudio.

`java3d.dev.java.net/`
A home page da *Java 3D API*. Essa API pode ser utilizada para produzir imagens tridimensionais típicas dos videogames de hoje.

`java.sun.com/products/java-media/speech/`
A *Java Speech API* permite aos programas realizar síntese e reconhecimento de fala.

`freetts.sourceforge.net/docs/index.php`
FreeTTS é uma implementação da Java Speech API.

Resumo

Seção 24.2 Carregando, exibindo e dimensionando imagens

- Os construtores da classe `ImageIcon` podem receber argumentos de vários formatos diferentes, incluindo um array de bytes que contém os bytes de uma imagem, uma `Image` (pacote `java.awt`) já carregada na memória, ou uma `String` ou um URL representando a localização da imagem.
- O Java suporta vários formatos de imagem, incluindo Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) e Portable Network Graphics (PNG). Em geral, os nomes de arquivo desses tipos terminam em `.gif`, `.jpg` (ou `.jpeg`) e `.png`, respectivamente.
- Os programas Java Web Start podem acessar o sistema de arquivos local via JNLP APIs do pacote `javax.jnlp`. Você pode utilizar o `FileOpenService` do JNLP para solicitar acesso limitado ao sistema de arquivos local.
- O método `static lookup` da classe JNLP `ServiceManager` obtém uma referência ao `FileOpenService`. Como outros serviços são fornecidos pelo JNLP, esse método retorna uma referência `Object`, que você deve fazer uma coerção para o tipo apropriado.
- O método `openFileDialog` do método `FileOpenService` exibe um diálogo de seleção do arquivo. Isso faz o Java Web Start pedir ao usuário que aprove a solicitação de acesso de sistema de arquivos local do programa. Se o usuário der a permissão, o diálogo **Open** é exibido. O método `openFileDialog` tem dois parâmetros — uma `String` para sugerir um diretório a ser aberto e um array `String` de extensões de arquivo aceitáveis.

- Quando o usuário selecionar um arquivo de imagem e clicar no botão `Open` do diálogo, o método `openFileDialog` retornará um objeto `FileContents`, que por razões de segurança não fornece ao programa acesso ao local exato do arquivo no disco. Em vez disso, o programa pode obter um `InputStream` e ler os bytes do arquivo.
- O método `FileContents` `getLength` retorna o número de bytes no arquivo.
- Os métodos `Component` `getWidth` e `getHeight` retornam a largura e a altura de um `Component`, respectivamente.
- Os métodos `ImageIcon` `getIconWidth` e `getIconHeight` retornam a largura e a altura de uma imagem, respectivamente.
- A classe `Graphics` fornece métodos `drawImage` sobrecarregados, um do qual exibe uma versão redimensionada de um `Image`. O primeiro argumento é a `Image` a ser desenhada. O segundo e terceiro argumentos representam as coordenadas de canto superior esquerdo da `Image`. O quarto e quinto argumentos especificam a largura e a altura da `Image`, respectivamente. O último argumento é uma referência a um `ImageObserver` — uma interface implementada pela classe `Component`.
- É possível que um programa tente exibir uma imagem antes de ela ter sido completamente carregada (ou de o download ter sido concluído). Enquanto uma `Image` é carregada, o `ImageObserver` recebe notificações e atualiza a imagem na tela de acordo com a necessidade.
- Um applet arrastável pode ser arrastado para fora da janela de navegador mantendo a tecla *Alt* pressionada e arrastando o applet com o mouse. O applet então continuará executando, mesmo se o navegador for fechado. Clicar na caixa de fechamento do applet quando ele estiver executando fora do navegador faz com que o applet volte para a janela de navegador se ela ainda estiver aberta, ou termine, caso contrário.

Seção 24.3 Animação de uma série de imagens

- O método `openMultiFileDialog` do `FileOpenService` exibe um diálogo de seleção de arquivo que permite ao usuário selecionar múltiplos arquivos ao mesmo tempo. O método `FileOpenService` `openMultiFileDialog` aceita os mesmos argumentos que o método `openFileDialog`, mas retorna um array de objetos `FileContents` representando o conjunto de arquivos selecionados pelo usuário.
- Objetos `Timer` geram `ActionEvents` a intervalos fixos de milissegundos. O construtor `Timer` recebe um retardo em milissegundos e um `ActionListener`. O método `Timer` `start` inicia o `Timer`. O método `stop` indica que o `Timer` deve parar de gerar eventos. O método `restart` indica que o `Timer` deve começar a gerar eventos novamente.
- O método `ImageIcon` `paintIcon` exibe a imagem `ImageIcon`. O método requer quatro argumentos — uma referência para o `Component` em que a imagem será exibida, uma referência para o objeto `Graphics` utilizado para renderizar a imagem, a coordenada x do canto superior esquerdo da imagem e a coordenada y do canto superior esquerdo da imagem.

Seção 24.4 Mapas de imagem

- Um mapa de imagem é uma imagem que tem áreas ativas (*hot areas*) em que o usuário pode clicar para realizar uma tarefa, como carregar uma página Web diferente em um navegador.

Seção 24.5 Carregando e reproduzindo clipes de áudio

- O método `applet` `play` tem duas formas:

```
public void play( URL location, String soundFileName );
public void play( URL soundURL );
```

  Uma versão carrega o clipe de áudio armazenado no arquivo `soundFileName` da `location` e reproduz o som. A outra versão aceita um `URL` que contém a localização e o nome de arquivo do clipe de áudio.
- O método `Applet` `getDocumentBase` indica a localização do arquivo de HTML que carregou o applet. O método `getCodeBase` indica para um applet onde está localizado o arquivo `.class`.
- O dispositivo de som que reproduz clipes de áudio suporta vários formatos de arquivo de áudio, incluindo formato de arquivo Sun Audio (extensão `.au`), formato de arquivo Wave do Windows (extensão `.wav`), formato de arquivo AIFF da Macintosh (extensões `.aif` ou `.aiff`) e formato de arquivo Musical Instrument Digital Interface (MIDI) (extensões `.mid` ou `.rmi`). O Java Media Framework (JMF) suporta formatos adicionais.
- O método `applet` `getAudioClip` tem duas formas que aceitam os mesmos argumentos que o método `play`. O método `getAudioClip` retorna uma referência a um `AudioClip`. `AudioClips` têm três métodos — `play`, `loop` e `stop`. O método `play` reproduz o clipe de áudio uma vez. O método `loop` faz um loop contínuo do clipe de áudio. O método `stop` termina um clipe de áudio que atualmente está reproduzindo.

Seção 24.6 Reproduzindo vídeo e outros tipos de mídia com o Java Media Framework

- A Sun Microsystems, a Intel e a Silicon Graphics trabalharam em conjunto para produzir a Java Media Framework (JMF).
- O pacote `javax.media` e seus subpacotes contêm as classes Java Media Framework.
- A classe `Manager` declara os métodos para acessar os recursos do sistema a fim de reproduzir e manipular mídia.
- O método `toURI` da classe `File` retorna um `URI` que aponta para o `File` no sistema.

Terminologia

.aif, extensão de arquivo, 756, 758
.aiff, extensão de arquivo, 756, 758
AIFF, formato de arquivo do Macintosh (extensões .aif ou .aiff), 756
applet arrastável, 748
área ativa, 753
.au, extensão de arquivo, 756, 758
AudioClip, interface, 756
.avi, extensão de arquivo, 758
CannotRealizePlayerException, exceção, 759
.class, extensão de arquivo, 756
clipe de áudio, 756
createRealizedPlayer, método da classe Manager, 758
Dimension, classe, 752
drawImage, método da classe Graphics, 747
FileOpenService, interface, 743
Future Splash (.spl), formato de arquivo, 758
getAudioClip, método da classe Applet, 756
getCodeBase, método da classe Applet, 756
getControlPanelComponent, método da interface Player, 760
getDelay, método da classe Timer, 765
getHeight, método da classe Component, 747
getIconHeight, método da classe ImageIcon, 747
getIconWidth, método da classe ImageIcon, 747
getImage, método da classe ImageIcon, 747
getLength, método da interface FileContents, 746
getMinimumSize, método da classe Component, 752
getPreferredSize, método da classe Component, 752
getVisualComponent, método da interface Player, 760
getWidth, método da classe Component, 747
.gif, extensão de arquivo, 744
Graphics Interchange Format (GIF), 744
Image, classe, 744
ImageObserver, interface, 747
isRunning, método da classe Timer, 751
Java 3D API, 743
Java Advanced Imaging API, 743
Java Image I/O API, 743
Java Media Framework (JMF), API, 743
Java Sound API, 743
Java Speech API, 743
javax.media, pacote, 758
Joint Photographic Experts Group (JPEG), 744
.jpeg, extensão de arquivo, 744
.jpg, extensão de arquivo, 744
LIGHTWEIGHT_RENDERER, constante da classe Manager, 759
lookup, método da classe ServiceManager, 746
loop, método da interface AudioClip, 756
Macromedia Flash, filmes (.swf), 758
Manager, classe, 758
mapa de imagem, 753
Microsoft Audio/Video Interleave (.avi), formato de arquivo, 758
.mid, extensão de arquivo, 756, 758
.mov, extensão de arquivo, 758
.mp3, extensão de arquivo, 758
MPEG-1, vídeos (extensões .mpeg ou .mpg), 758
.mpeg, extensão de arquivo, 758
MPEG Layer 3 Audio (.mp3), formato de arquivo, 758
.mpg, extensão de arquivo, 758
multimídia, 743
Musical Instrument Digital Interface (MIDI), formato de arquivo (extensões .mid ou .rmi), 756
NoPlayerException, exceção, 759
openFileDialog, método da interface FileOpenService, 746
openMultiFileDialog, método da interface FileOpenService, 751
paintIcon, método da classe ImageIcon, 751
play, método da classe Applet, 756
play, método da interface AudioClip, 756
Player, interface, 758
.png, extensão de arquivo, 744
Portable Network Graphics (PNG), 744
QuickTime (.mov), formato de arquivo, 758
read, método da classe InputStream, 746
restart, método da classe Timer, 751
.rmi, extensão de arquivo, 756, 758
ServiceManager, classe, 746
setDelay, método da classe Timer, 765
setHint, método da classe Manager, 759
showStatus, método da classe Applet, 753
som, sistema, 756
.spl, extensão de arquivo, 758
start, método da classe Timer, 751
start, método da interface Player, 760
stop, método da classe Timer, 752
stop, método da interface AudioClip, 756
Sun Audio, formato de arquivo (extensão .au), 756
.swf, extensão de arquivo, 758
Timer, classe, 751
toURI, método da classe File, 761
toURL, método da classe URI, 761
.wav, extensão de arquivo, 756
Windows Wave, formato de arquivo (extensão .wav), 756

Exercícios de autorrevisão

24.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) O método Graphics _____ exibe uma imagem em um applet.
b) O Java fornece dois mecanismos para reproduzir sons em um applet — o método play de Applet e o método play da interface _____.
c) Um(a) _____ é uma imagem que tem áreas ativas em que o usuário pode clicar para realizar uma tarefa como carregar uma página Web.
d) O método _____ da classe ImageIcon exibe a imagem do ImageIcon.
e) O Java suporta vários formatos de imagem, incluindo _____, _____ e _____.
f) O método static lookup da classe JNLP _____ obtém uma referência ao FileOpenService.

24.2 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Um arquivo de áudio é marcado para a coleta de lixo depois de ele ser reproduzido por meio do método play da classe Applet.
b) A classe ImageIcon fornece construtores que permitem que um objeto ImageIcon seja inicializado somente com uma imagem do computador local.
c) O método play da classe AudioClip faz loop de um clipe de áudio continuamente.
d) A Java Image I/O API é utilizada para adicionar imagens gráficas 3D a um aplicativo Java.
e) O método de Applet getDocumentBase retorna, como um objeto da classe URL, a localização na Internet do arquivo HTML que invocou o applet.
f) Os métodos FileOpenService openFileDialog e openMultiFileDialog retornam um objeto FileContents e um array de objetos FileContents, respectivamente.

Respostas dos exercícios de autorrevisão

24.1 a) `drawImage`. b) `AudioClip`. c) mapa de imagem. d) `paintIcon`. e) Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), Portable Network Graphics (PNG). f) `ServiceManager`.

24.2 a) Verdadeira. b) Falsa. `ImageIcon` também pode carregar imagens da internet. c) Falsa. O método `play` da classe `AudioClip` reproduz um clipe de áudio uma vez. O método `loop` da classe `AudioClip` repete um clipe de áudio continuamente. d) Falsa. A Java 3D API é utilizada para criar e modificar imagens gráficas 3D. A Java Image I/O API é utilizada para ler e gerar a saída de imagens para arquivos. e) Verdadeira. f) Falsa.

Exercícios

24.3 Descreva como fazer uma animação "amigável ao navegador".

24.4 Descreva os métodos Java para reproduzir e manipular clipes de áudio.

24.5 Expliquem como os mapas de imagem são utilizados. Liste vários exemplos de sua utilização.

24.6 (***Apagando aleatoriamente uma imagem***) Suponha que uma imagem seja exibida em uma área retangular de tela. Uma maneira de apagar a imagem é simplesmente configurar cada pixel com a mesma cor imediatamente, mas o efeito visual será desagradável. Escreva um programa Java que exibe uma imagem e, então, a apaga utilizando a geração de números aleatórios para selecionar pixels individuais a serem apagados. Depois que a maior parte da imagem tiver sido apagada, apague todos os pixels restantes de uma vez. Você pode desenhar pixels individuais como uma linha que inicia e termina nas mesmas coordenadas. Você poderia tentar diversas variantes desse problema. Por exemplo, você talvez exiba linhas ou formas aleatoriamente para apagar regiões da tela.

24.7 (***Flasher de texto***) Crie um programa Java que faça um texto piscar repetidamente na tela. Faça isso alternando o texto com uma imagem simples de cor do fundo. Permita que o usuário controle a "velocidade de piscamento" e a cor ou padrão de fundo. Você precisará utilizar métodos **`getDelay`** e **`setDelay`** da classe `Timer`. Esses métodos são utilizados para recuperar e configurar o intervalo em milissegundos entre `ActionEvents`, respectivamente

24.8 (***Flasher de imagem***) Crie um programa Java que faça uma imagem piscar repetidamente na tela. Faça isso alternando a imagem com uma imagem simples de cor do fundo.

24.9 (***Relógio digital***) Implemente um programa que exibe um relógio digital na tela.

24.10 (***Chamando a atenção para uma imagem***) Se quiser enfatizar uma imagem, você pode colocar uma fileira de lâmpadas simuladas em torno dela. Pode fazer com que todas as lâmpadas acendam ao mesmo tempo ou ligar e fazê-las acender e apagar em sequência uma após a outra.

24.11 (***Ampliador de imagem***) Crie um programa que permita ampliar e reduzir uma imagem.

24.12 (***Modificação* de LoadImageAndScale**) Modifique o applet `LoadImageAndScale` (Figura 24.1) para fornecer um segundo botão que permite ao usuário escolher uma nova imagem. A rotina de tratamento de evento de botão deve utilizar o JNLP `FileOpenService` a fim de exibir um diálogo Open, para que o usuário possa selecionar uma nova imagem.

24.13 (***Visualizador de imagem***) Utilizando as técnicas de JNLP que aprendeu nas seções 24.2–24.3, crie um aplicativo visualizador de imagem que permite ao usuário selecionar um grupo de imagens para exibição. O aplicativo deve exibir uma `JList` contendo os nomes dos arquivos selecionados. Você pode obter o nome do arquivo representado por um objeto `FileContents` chamando seu método `getName`. Quando o usuário clicar no nome de uma imagem na `JList`, o aplicativo deve exibir a imagem na janela.

Seção especial: projetos de multimídia desafiadores

Os exercícios precedentes são voltados para o texto e projetados para testar seu entendimento de conceitos fundamentais de multimídia. Esta seção inclui uma coleção de projetos de multimídia avançados. Você deve achar esses problemas desafiadores, mas divertidos. Os problemas variam em dificuldade. Alguns requerem uma hora ou duas para escrever e implementar o programa. Outros são úteis para atribuições de laboratório que talvez requeiram duas ou três semanas de estudo e implementação. Alguns são projetos de conclusão de curso desafiadores. [*Nota para instrutores: Não* são fornecidas soluções para esses exercícios.]

24.14 (***Animação***) Crie um programa de animação Java de uso geral. Você deve permitir ao usuário especificar a sequência de quadros a ser exibida, a velocidade em que as imagens são exibidas, os áudios a ser reproduzidos enquanto a animação estiver executando e assim por diante.

24.15 (***Limericks***) Modifique o programa de escrever limericks que você escreveu no Exercício 16.6 para cantar os limericks que seu programa criou.

24.16 (***Transição aleatória entre imagens***) Esse exercício fornece um efeito visual muito elegante. Se você estiver exibindo uma imagem em uma determinada área na tela e quiser fazer uma transição para outra imagem na mesma área, armazene a nova imagem de tela em um buffer fora da tela e copie pixels aleatoriamente dela para a área de exibição, sobrepondo os pixels que já estão nessas localizações. Quando a maioria dos pixels for copiada, copie a inteira imagem nova para a área de exibição para certificar-se de que você está exibindo a nova imagem completa. Para implementar esse programa você pode precisar utilizar as classes `PixelGrabber` e `MemoryImageSource` (veja a documentação do Java API para descrições dessas classes). Você poderia tentar diversas variantes desse problema. Por exemplo, selecione todos os pixels em uma linha reta ou em uma forma escolhida aleatoriamente na nova imagem e sobreponha-os às posições correspondentes da antiga imagem.

24.17 **(*Áudio de fundo*)** Adicione áudio de fundo para um de seus aplicativos favoritos utilizando o método `loop` da classe `AudioClip` para reproduzir o som no fundo enquanto interage com seu aplicativo de maneira normal.

24.18 **(*Painel de caracteres rolantes*)** Crie um programa que rola caracteres da direita para a esquerda (ou da esquerda para a direita se isso for apropriado para seu idioma) ao longo de um painel de exibição. Como uma opção, exiba o texto em um loop contínuo, de modo que depois que o texto desaparecer em uma extremidade, ele reapareça na outra.

24.19 **(*Painel de imagens rolantes*)** Crie um programa que rola uma imagem ao longo de um painel.

24.20 **(*Relógio Analógico*)** Crie um programa que exibe um relógio analógico com ponteiros de hora, minutos e segundos que se movem apropriadamente conforme o tempo passa.

24.21 **(*Áudio dinâmico e caleidoscópio gráfico*)** Desenvolva um programa de caleidoscópio que exibe imagens gráficas refletidas para simular o popular brinquedo infantil. Incorpore efeitos de áudio que "reflitam" as imagens gráficas dinamicamente mutantes do seu programa.

24.22 **(*Gerador automático de quebra-cabeça*)** Crie um gerador e manipulador de quebra-cabeça. O usuário especifica uma imagem. O programa carrega e exibe a imagem, e depois a divide em formas selecionadas aleatoriamente e as embaralha. O usuário então utiliza o mouse para mover as peças para resolver o quebra-cabeça. Adicione sons de áudio de movimento e encaixe adequados. Você poderia controlar a posição de cada peça e então utilizar efeitos de áudio para ajudar o usuário a encaixar as peças nas posições corretas.

24.23 **(*Gerando e percorrendo o labirinto*)** Desenvolva um programa gerador de labirinto e de percurso do labirinto com base em multimídia utilizando os programas de labirinto escritos nos exercícios 18.20–18.22. Deixe o usuário personalizar o labirinto especificando o número de linhas e colunas e indicando o nível de dificuldade. Crie uma animação de um ratinho percorrendo o labirinto. Utilize áudio para dramatizar o movimento de seu personagem ratinho.

24.24 **(*Caça-níqueis*)** Desenvolva uma simulação multimídia de um caça-níqueis. Crie três rodas giratórias. Coloque símbolos e imagens de várias frutas em cada roda. Utilize a geração de números aleatórios para simular a giro e a parada de cada roda em um símbolo.

24.25 **(*Corrida de cavalos*)** Crie uma simulação de uma corrida de cavalos. Permita múltiplos competidores. Utilize áudios para um apresentador da corrida. Reproduza os áudios adequados para indicar o status correto de cada competidor por toda a corrida. Utilize áudios para anunciar os resultados finais. Você poderia tentar simular os tipos de jogo de corrida de cavalos que frequentemente ocorrem em parques de diversões. Os jogadores fazem turnos para usar o mouse e precisam realizar alguma manipulação baseada em suas habilidades com o mouse para avançar seus cavalos.

24.26 **(*Jogo de malha*)** Desenvolva uma simulação baseada em multimídia do jogo de malha. Utilize áudio e efeitos visuais apropriados.

24.27 **(*Jogo de bilhar*)** Crie uma simulação baseada em multimídia do jogo de bilhar. Cada jogador faz turnos para usar o mouse a fim de posicionar um taco de bilhar e acertá-lo contra uma bola no ângulo adequado para tentar fazer outras bolas caírem nos bolsos. Seu programa deve manter uma contagem.

24.28 **(*Artista*)** Projete um programa de arte que forneça a um artista uma grande variedade de recursos para desenhar, utilizar imagens e utilizar animações para criar uma exposição dinâmica de arte multimídia.

24.29 **(*Designer de fogos de artifício*)** Crie um programa em Java que alguém poderia utilizar para criar um show de fogos de artifício. Crie uma variedade de demonstrações de fogos de artifício. Então orquestre a queima dos fogos de artifício para obter um efeito máximo.

24.30 **(*Planejador de planta baixa*)** Desenvolva um programa que ajude alguém a organizar os móveis de uma casa. Adicione recursos que permitam que a pessoa alcance a melhor disposição possível.

24.31 **(*Palavras cruzadas*)** Palavras cruzadas estão entre os passatempos mais populares. Desenvolva um programa de palavras cruzadas baseado em multimídia. Seu programa deve permitir que o jogador insira e apague palavras facilmente. Associe seu programa a um grande dicionário computadorizado. Seu programa também deve ser capaz de sugerir as palavras com base nas letras já preenchidas. Forneça outro recurso que facilite o trabalho do entusiasta de palavras cruzadas.

24.32 **(*Jogo do 15*)** Escreva um programa com base em multimídia que permite que o usuário jogue o jogo do 15. O jogo é um tabuleiro de 4 por 4 com um total de 16 posições. Uma posição está vazia, as outras são ocupadas por 15 ladrilhos numerados de 1 a 15. O usuário pode mover qualquer ladrilho ao lado da posição atualmente vazia para aquela posição clicando no ladrilho. Seu programa deve criar o tabuleiro com os ladrilhos em ordem aleatória. O objetivo é organizar os ladrilhos na ordem sequencial linha por linha.

24.33 **(*Testador de precisão de reação/tempo de reação*)** Crie um programa que move uma forma criada aleatoriamente ao redor da tela. O usuário move o mouse para capturar e clicar na forma. A velocidade e o tamanho da forma podem ser variados. Mantenha estatísticas sobre quanto tempo geralmente o usuário leva para perceber uma forma de um determinado tamanho. O usuário provavelmente terá mais dificuldade para capturar mais rapidamente formas menores em movimento.

24.34 **(*Arquivo de calendário/lembretes*)** Utilizando tanto áudio como imagens, crie um arquivo de calendário e "lembretes" de uso geral. Por exemplo, o programa deve cantar "Feliz Aniversário" quando você utilizá-lo em seu aniversário. Faça com que o programa exiba imagens e reproduza áudios associados com eventos importantes. Além disso, faça-o lembrá-lo com antecedência desses eventos importantes. Seria interessante, por exemplo, fazer o programa fornecer um aviso com uma semana de antecedência de modo que você possa enviar um cartão de felicitações apropriado para essa pessoa especial.

24.35 **(*Rotação de imagens*)** Crie um programa que permita rotacionar uma imagem por vários graus (até um máximo de 360 graus). O programa deve permitir especificar que você quer girar a imagem continuamente. Esse programa deve permitir ajustar dinamicamente a velocidade de rotação.

24.36 (***Colorindo fotografias e imagens P&B***) Crie um programa que permita colorir uma fotografia em preto e branco. Forneça uma paleta de cores para selecionar cores. Seu programa deve permitir a aplicação de diferentes cores a regiões distintas da imagem.

24.37 (***Simulador Simpletron baseado em multimídia***) Modifique o simulador Simpletron que você desenvolveu nos exercícios dos capítulos anteriores (exercícios 7.35–7.37) para incluir recursos de multimídia. Adicione sons de computador para indicar que o Simpletron está executando instruções. Adicione um som de vidro quebrando quando um erro fatal ocorrer. Utilize luzes intermitentes para indicar as células de memória ou os registros que estão sendo manipulados atualmente. Utilize outras técnicas de multimídia conforme apropriado para tornar seu simulador Simpletron mais valioso para seus usuários como uma ferramenta educacional.

Fazendo a diferença

24.38 (***Projeto de acessibilidade: Síntese de fala***) Os computadores podem ajudar usuários cegos ou com problemas de visão fornecendo páginas Web que falam, e-mails e outros documentos que utilizam "mecanismos" de TTS (text-to-speech) ou de síntese de fala. Da mesma forma, para ajudar pessoas que têm dificuldade de interagir com um computador via mouse e teclado, os mecanismos de reconhecimento da fala permitem a computadores reconhecerem comandos falados. Com a síntese e o reconhecimento de fala, os usuários podem "falar" com computadores. Neste exercício, você irá pesquisar e explorar a síntese de fala com a Java Speech API (`java.sun.com/products/java-media/speech/`). Baixe e instale o sintetizador de fala FreeTTS de código-fonte aberto (freetts.sourceforge.net/docs/index.php). Explore a documentação do FreeTTS, depois implemente um aplicativo no qual o usuário pode inserir o texto em uma `JTextArea`. Quando o usuário clicar no `JButton` **Speak**, o programa deve utilizar o FreeTTS para falar o texto em voz alta.

24.39 (***Projeto de acessibilidade: Reconhecimento de fala***) Neste exercício, você irá pesquisar e explorar o reconhecimento da fala com a Java Speech API. Baixe o mecanismo de reconhecimento de fala Sphinx-4 de código-fonte aberto e instale-o (`cmusphinx.sourceforge.net/sphinx4/`). Escreva um programa que permite que um usuário fale com o computador. Utilize as capacidades de reconhecimento da fala de exibir o que o usuário diz em um `JTextArea`. Permita que o usuário salve o conteúdo do `JTextArea` em um arquivo no disco falando o comando "salvar".

24.40 (***Projeto: Simulador de robótica Simbad***) A robótica promete a grande tarefa de lidar com trabalhos perigosos para seres humanos, como mineração de carvão; exploração de minas e exploração das profundezas do oceano e do espaço. Simbad (`simbad.sourceforge.net`) é um simulador de robótica 3D de código-fonte aberto com base em Java. De acordo com a página Web do projeto, ele suporta simulações de um ou vários robôs; sensores de visão, distância e contato; e muito mais. Você pode baixar o Simbad de `simbad.sourceforge.net/index.php#download`. Você também precisará baixar e instalar o Java 3D — nesse site são fornecidas instruções para Mac OS X, Windows e Linux.

Assim que concluir o download do Simbad e a instalação do Java 3D, você pode experimentar o exemplo simples fornecido em `simbad.sourceforge.net/example1.php`. Depois de executar isso, leia o Simbad Programming Guide em `simbad.sourceforge.net/guide.php` e tente modificar o exemplo simples para realizar algumas tarefas diferentes. Se estiver interessado em explorar ainda mais a robótica, estude a documentação da API em `simbad.sourceforge.net/doc/` e crie seu próprio programa de simulação de robôs utilizando o Simbad. Por exemplo, crie a simulação de um aspirador de pó robótico que se move para a direção para a qual ele está voltado até encontrar um obstáculo e, então, escolhe outra direção aleatoriamente.

25

Se um ator entra pela porta, você não tem nada. Mas se ele entrar pela janela, você terá algo especial.
— **Billy Wilder**

...a força dos eventos desperta talentos adormecidos.
— **Edward Hoagland**

Você e eu veríamos fotografias mais interessantes se eles parassem de se preocupar e, em vez disso, aplicassem senso comum ao problema do registro da aparência e comportamento de suas próprias eras.
— **Jessie Tarbox Beals**

Componentes GUI: Parte 2

Objetivos

Neste capítulo, você aprenderá:

- A criar e manipular controles deslizantes, menus, menus pop-up e janelas.
- A alterar programaticamente a aparência e comportamento de uma GUI, utilizando a aparência e comportamento plugável do Swing.
- A criar uma interface de múltiplos documentos com `JDesktopPane` e `JInternalFrame`.
- A utilizar gerenciadores adicionais de layout.

Sumário

25.1 Introdução
25.2 JSlider
25.3 Windows: notas adicionais
25.4 Utilizando menus com frames
25.5 JPopupMenu
25.6 Aparência e comportamento plugável
25.7 JDesktopPane e JInternalFrame
25.8 JTabbedPane
25.9 Gerenciadores de layout: BoxLayout e GridBagLayout
25.10 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

25.1 Introdução

Neste capítulo, continuamos nosso estudo de GUIs. Discutimos componentes e gerenciadores de layout adicionais e projetamos a base para construir mais GUIs complexas.

Começamos nossa discussão com controles deslizantes que permitem selecionar um intervalo de valores de números inteiros. Em seguida, discutimos alguns detalhes adicionais das janelas. Você aprenderá a utilizar menus que permitem ao usuário executar efetivamente tarefas no programa. A aparência e comportamento de uma GUI Swing pode ser uniforme em todas as plataformas em que um programa Java executa ou a GUI pode ser personalizada utilizando a **aparência e comportamento plugável** (***Pluggable Look-And-Feel* — PLAF**) do Swing. Fornecemos um exemplo que ilustra como alterar entre a aparência e comportamento de metal padrão do Swing (que parece e se comporta da mesma maneira em diferentes plataformas), a nova aparência e comportamento Nimbus (introduzida no Capítulo 14), uma aparência e comportamento que simula o **Motif** (uma aparência e comportamento popular do UNIX) e uma que simula a aparência e comportamento do Windows da Microsoft.

Muitos aplicativos de hoje em dia utilizam uma interface de múltiplos documentos (*Multiple Document Interface* — MDI) — uma janela principal (frequentemente chamada janela-pai) contendo outras janelas (frequentemente chamadas janelas-filhas) para gerenciar vários documentos abertos em paralelo. Por exemplo, muitos programas de correio eletrônico permitem abrir várias janelas de correio eletrônico ao mesmo tempo de modo que você possa compor ou ler múltiplas mensagens. Demonstramos as classes do Swing para criar interface de múltiplos documentos. O capítulo termina com uma série de exemplos que discute gerenciadores adicionais de layout para organizar interfaces gráficas com o usuário.

O Swing é um assunto extenso e complexo. Há muito mais componentes GUI e capacidades do que o espaço neste livro permite apresentar. Muito outros componentes GUI Swing são introduzidos nos demais capítulos deste livro quando necessários.

25.2 JSlider

O **`JSlider`** permite a um usuário selecionar a partir de um intervalo de valores inteiros. A classe `JSlider` herda de `JComponent`. A Figura 25.1 mostra um `JSlider` horizontal com **marcas de medida** (*tick marks*) e um **marcador** (*thumb*) que permitem a um usuário selecionar um valor. Os `JSliders` podem ser personalizados para exibir marcas de medida principais, marcas de medida secundárias e rótulos para as marcas de medida. Eles também suportam **marcas de aderência**, que fazem a miniatura, quando posicionada entre duas marcas, aderir à marca mais próxima.

Figura 25.1 | Um componente `JSlider` com uma orientação horizontal.

A maioria dos componentes GUI Swing suporta interações do usuário pelo mouse e pelo teclado. Por exemplo, se um `JSlider` tem o foco (isto é, ele é o componente GUI atualmente selecionado na interface com o usuário), a tecla de seta para a esquerda e a tecla da seta para a direita fazem com que o marcador do `JSlider` diminua ou aumente por 1, respectivamente. A tecla da seta para baixo e a tecla da seta para cima também fazem com que a caixa de rolagem do `JSlider` diminua ou aumente por 1 tique, respectivamente. A tecla *PgDn* (*page down*) e a tecla *PgUp* (*page up*) fazem com que o marcador do `JSlider` diminua ou aumente por **incrementos de bloco** de um décimo do intervalo de valores, respectivamente. A tecla *Home* move a caixa de rolagem para o valor mínimo do `JSlider`, e a *tecla End* move a caixa de rolagem para o valor máximo do `JSlider`.

Os `JSliders` têm uma orientação horizontal ou uma orientação vertical. Para um `JSlider` horizontal, o valor mínimo está na extrema esquerda do `JSlider` e o valor máximo está na extrema direita. Para um `JSlider` vertical, o valor mínimo está na parte inferior

e o valor máximo na parte superior. As posições de valor mínimo e máximo em um `JSlider` podem ser alternadas chamando o método `JSlider` **`setInverted`** com argumento `boolean` `true`. A posição relativa da caixa de rolagem indica o valor atual do `JSlider`.

O programa nas figuras 25.2–25.4 permite ao usuário dimensionar um círculo desenhado em uma subclasse do `JPanel` chamada `OvalPanel` (Figura 25.2). O usuário especifica o diâmetro do círculo com um `JSlider` horizontal. A classe `OvalPanel` sabe desenhar um círculo em si mesma, utilizando sua própria variável de instância `diameter` para determinar o diâmetro do círculo — o `diameter` é utilizado como a largura e a altura do quadro delimitador em que o círculo será exibido. O valor `diameter` é configurado quando o usuário interage com o `JSlider`. O handler de evento chama o método `setDiameter` na classe `OvalPanel` para configurar o `diameter` e chama `repaint` para desenhar o novo círculo. A chamada a `repaint` resulta em uma chamada ao método `paintComponent` de `OvalPanel`.

```
1  // Figura 25.2: OvalPanel.java
2  // Uma classe personalizada de JPanel.
3  import java.awt.Graphics;
4  import java.awt.Dimension;
5  import javax.swing.JPanel;
6
7  public class OvalPanel extends JPanel
8  {
9     private int diameter = 10; // diâmetro padrão de 10
10
11    // desenha uma oval do diâmetro especificado
12    public void paintComponent( Graphics g )
13    {
14       super.paintComponent( g );
15
16       g.fillOval( 10, 10, diameter, diameter ); // desenha um círculo
17    } // fim do método paintComponent
18
19    // valida e configura o diâmetro e então repinta
20    public void setDiameter( int newDiameter )
21    {
22       // se diâmetro inválido, assume o padrão de 10
23       diameter = ( newDiameter >= 0 ? newDiameter : 10 );
24       repaint(); // repinta o painel
25    } // fim do método setDiameter
26
27    // utilizado pelo gerenciador de layout para determinar o tamanho preferido
28    public Dimension getPreferredSize()
29    {
30       return new Dimension( 200, 200 );
31    } // fim do método getPreferredSize
32
33    // utilizado pelo gerenciador de layout para determinar o tamanho mínimo
34    public Dimension getMinimumSize()
35    {
36       return getPreferredSize();
37    } // fim do método getMinimumSize
38 } // fim da classe OvalPanel
```

Figura 25.2 | Subclasse `JPanel` para desenhar círculos de um diâmetro especificado.

```
1  // Figura 25.3: SliderFrame.java
2  // Utilizando JSliders para dimensionar uma oval.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JSlider;
7  import javax.swing.SwingConstants;
8  import javax.swing.event.ChangeListener;
9  import javax.swing.event.ChangeEvent;
10
11 public class SliderFrame extends JFrame
12 {
13    private JSlider diameterJSlider; // controle deslizante para selecionar o diâmetro
```

```
14       private OvalPanel myPanel; // painel para desenhar um círculo
15
16       // construtor sem argumento
17       public SliderFrame()
18       {
19          super( "Slider Demo" );
20
21          myPanel = new OvalPanel(); // cria o painel para desenhar um círculo
22          myPanel.setBackground( Color.YELLOW ); // configura o fundo como amarelo
23
24          // configura o JSlider para controlar o valor do diâmetro
25          diameterJSlider =
26             new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
27          diameterJSlider.setMajorTickSpacing( 10 ); // cria uma marca de medida a cada 10
28          diameterJSlider.setPaintTicks( true ); // pinta as marcas de medida no controle deslizante
29
30          // registra o ouvinte de evento do JSlider
31          diameterJSlider.addChangeListener(
32
33             new ChangeListener() // classe interna anônima
34             {
35                // trata da alteração de valor do controle deslizante
36                public void stateChanged( ChangeEvent e )
37                {
38                   myPanel.setDiameter( diameterJSlider.getValue() );
39                } // fim do método stateChanged
40             } // fim da classe interna anônima
41          ); // fim da chamada a addChangeListener
42
43          add( diameterJSlider, BorderLayout.SOUTH ); // adiciona um controle deslizante ao quadro
44          add( myPanel, BorderLayout.CENTER ); // adiciona painel ao frame
45       } // fim do construtor de SliderFrame
46    } // fim da classe SliderFrame
```

Figura 25.3 | Valor do `JSlider` utilizado para determinar o diâmetro de um círculo.

```
1    // Figura 25.4: SliderDemo.java
2    // testando SliderFrame.
3    import javax.swing.JFrame;
4
5    public class SliderDemo
6    {
7       public static void main( String[] args )
8       {
9          SliderFrame sliderFrame = new SliderFrame();
10         sliderFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11         sliderFrame.setSize( 220, 270 ); // configura o tamanho do frame
12         sliderFrame.setVisible( true ); // exibe o frame
13      } // fim de main
14   } // fim da classe SliderDemo
```

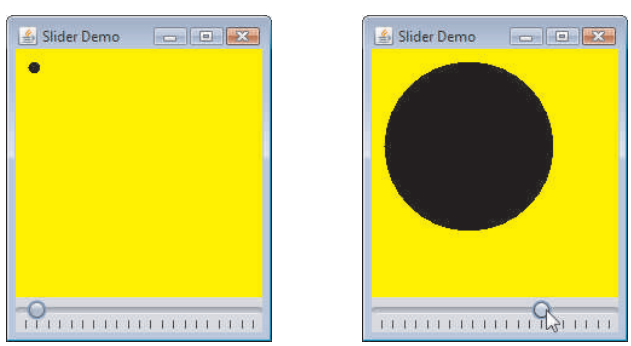


Figura 25.4 | Classe de teste para `SliderFrame`.

A classe `OvalPanel` (Figura 25.2) contém um método `paintComponent` (linhas 12–17) que desenha uma oval preenchida (nesse exemplo, um círculo), um método `setDiameter` (linhas 20–25) que altera o `diameter` do círculo e `repaints` a `OvalPanel`, um método `getPreferredSize` (linhas 28–31) que retorna a largura e altura preferidas de uma `OvalPanel` e um método `getMinimumSize` (linhas 34–37) que retorna a largura e altura mínimas de uma `OvalPanel`. A Seção 24.3 introduziu `getPreferredSize` e `getMinimumSize`, que são utilizados por alguns gerenciadores de layout para determinar o tamanho de um componente.

A classe `SliderFrame` (Figura 25.3) cria o `JSlider` que controla o diâmetro do círculo. O construtor da classe `SliderFrame` (linhas 17–45) cria o objeto `OvalPanel myPanel` (linha 21) e configura sua cor de fundo (linha 22). As linhas 25–26 criam o objeto `JSlider diameterSlider` para controlar o diâmetro do círculo desenhado no `OvalPanel`. O construtor `JSlider` recebe quatro argumentos. O primeiro argumento especifica a orientação do `diameterSlider`, que é `HORIZONTAL` (uma constante na interface `SwingConstants`). O segundo e o terceiro argumentos indicam os valores inteiros mínimo e máximo no intervalo de valores para este `JSlider`. O último argumento indica que o valor inicial do `JSlider` (isto é, onde a caixa de rolagem é exibida) deve ser `10`.

As linhas 27–28 personalizam a aparência do `JSlider`. O método **`setMajorTickSpacing`** indica que cada marca de medida principal representa 10 valores no intervalo de valores suportados pelo `JSlider`. O método **`setPaintTicks`** com um argumento `true` indica que as marcas de medida devem ser exibidas (elas não são exibidas por padrão). Para outros métodos que são utilizados para personalizar a aparência de um `JSLIDER`, consulte a documentação on-line do `JSlider` (`java.sun.com/javase/6/docs/api/javax/swing/JSlider.html`).

Os `JSliders` geram **`ChangeEvents`** (pacote `javax.swing.event`) em resposta a interações de usuário. Um objeto de uma classe que implementa a interface **`ChangeListener`** (pacote `javax.swing.event`) e declara o método **`stateChanged`** pode responder a `ChangeEvents`. As linhas 31–41 registram um `ChangeListener` para tratar eventos do `diameterSlider`. Quando o método `stateChanged` (linhas 36–39) é chamado em resposta a uma interação do usuário, a linha 38 chama o método `setDiameter` de `myPanel` e passa o valor atual do `JSlider` como um argumento. O método `JSlider` **`getValue`** retorna a posição atual do marcador.

25.3 Windows: notas adicionais

Um `JFrame` é uma **janela** com uma **barra de título** e uma **borda**. A classe `JFrame` é uma subclasse de `Frame` (pacote `java.awt`), que é uma subclasse de `Window` (pacote `java.awt`). Assim, `JFrame` é um dos componentes Swing GUI de maior peso. Ao exibir uma janela em um programa Java, a janela é fornecida pelo conjunto de ferramentas de janelas da plataforma local e, portanto, ela irá parecer qualquer outra janela exibida nessa plataforma. Quando um aplicativo Java executa em um Macintosh e exibe uma janela, a barra de título da janela e as bordas serão semelhantes àquelas de outros aplicativos do Macintosh. Quando um aplicativo Java é executado em um sistema Microsoft Windows e exibe uma janela, a barra de título e bordas da janela parecerão aquelas de outros aplicativos Microsoft Windows. E quando um aplicativo Java executar em uma plataforma UNIX e exibir uma janela, a barra de título da janela e as bordas serão semelhantes aos outros aplicativos UNIX dessa plataforma.

Por padrão, quando o usuário fecha uma janela `JFrame`, ela se torna oculta (isto é, ela desaparece da tela), mas você pode controlar isso com método `JFrame` **`setDefaultCloseOperation`**. Interface **`WindowConstants`** (pacote `javax.swing`), que a classe `JFrame` implementa, declara três constantes — `DISPOSE_ON_CLOSE`, `DO_NOTHING_ON_CLOSE` e `HIDE_ON_CLOSE` (o padrão) — para o uso com esse método. Algumas plataformas permitem que apenas um número limitado de janelas seja exibido na tela. Portanto, uma janela é um recurso valioso que deve ser devolvido ao sistema quando não for mais necessária. A classe `Window` (uma superclasse indireta de `JFrame`) declara o método **`dispose`** para esse propósito. Quando uma `janela` não é mais necessária em um aplicativo, você deve descartá-la explicitamente. Isso pode ser feito chamando o método `dispose` de `Window` ou chamando o método `setDefaultCloseOperation` com o argumento `WindowConstants.DISPOSE_ON_CLOSE`. Terminar um aplicativo também retornará os recursos da janela ao sistema. Utilizar `DO_NOTHING_ON_CLOSE` indica que o programa determinará o que fazer quando o usuário tenta fechar a janela. Por exemplo, o programa talvez queira perguntar se é preciso salvar as modificações de um arquivo antes de fechar uma janela.

Dica de desempenho 25.1

Uma janela é um recurso caro de sistema. Retorne-a ao sistema chamando seu método `dispose` quando a janela não mais é necessária.

Por padrão, uma janela não é exibida na tela até que o programa invoque o método `setVisible` da janela (herdado da classe `java.awt.Component`) com um argumento `true`. O tamanho de uma janela deve ser configurado com uma chamada ao método `setSize` (herdado da classe `java.awt.Component`). A posição de uma janela quando aparece na tela é especificada com o método **`setLocation`** (herdado da classe `java.awt.Component`).

Erro comum de programação 25.1

Esquecer de chamar o método `setVisible` em uma janela é um erro de lógica de tempo de execução — a janela não é exibida.

Erro comum de programação 25.2

Esquecer de chamar o método `setSize` em uma janela é um erro de lógica em tempo de execução — somente a barra de título aparece.

Quando o usuário manipula a janela, essa ação gera **eventos de janela**. Os ouvintes de evento são registrados para eventos de janela com o método `Window` **`addWindowListener`**. A interface **`WindowListener`** fornece sete métodos de tratamento de eventos de janela — **`windowActivated`** (chamado quando o usuário torna uma janela ativa), **`windowClosed`** (chamado depois de a janela ser fechada), **`windowClosing`** (chamado quando o usuário inicia o fechamento da janela), **`windowDeactivated`** (chamado quando o usuário torna outra janela a janela ativa), **`windowDeiconified`** (chamado quando o usuário restaura uma janela que está sendo minimizada), **`windowIconified`** (chamado quando o usuário minimiza uma janela) e **`windowOpened`** (chamado quando um programa exibe uma janela na tela pela primeira vez).

25.4 Utilizando menus com frames

Os **menus** são uma parte integrante das GUIs. Eles permitem que o usuário realize ações sem poluir desnecessariamente uma GUI com componentes extras. Em GUIs Swing, os menus podem ser anexados somente aos objetos das classes que fornecem o método **`setJMenuBar`**. Duas dessas classes são `JFrame` e `JApplet`. As classes utilizadas para declarar menus são `JMenuBar`, `JMenu`, `JMenuItem`, `JCheckBoxMenuItem` e classe `JRadioButtonMenuItem`.

Observações sobre a aparência e o comportamento 25.1

Os menus simplificam as GUIs porque os componentes podem ser ocultos dentro deles. Esses componentes serão visíveis somente quando o usuário procurá-los selecionando no menu.

Resumo dos vários componentes relacionados a menu

A classe **`JMenuBar`** (uma subclasse de `JComponent`) contém os métodos necessários para gerenciar uma **barra de menus**, que é um contêiner de menus. A classe **`JMenu`** (uma subclasse de `javax.swing.JMenuItem`) contém os métodos necessários para gerenciar menus. Os menus contêm itens de menu e são adicionados a barras de menus ou a outros menus, a exemplo dos submenus. Quando um menu é clicado, ele se expande para mostrar sua lista dos itens de menu.

A classe **`JMenuItem`** (uma subclasse de `javax.swing.AbstractButton`) contém os métodos necessários para gerenciar **itens de menu**. Um item de menu é um componente GUI dentro de um menu que, quando selecionado, resulta em um evento de ação. Um item de menu pode ser utilizado para iniciar uma ação ou ser um **submenu** que fornece mais itens de menu a partir dos quais o usuário pode selecionar. Os submenus são úteis para agrupar itens de menu relacionados em um menu.

A classe **`JCheckBoxMenuItem`** (uma subclasse de `javax.swing.JMenuItem`) contém os métodos necessários para gerenciar itens de menu que podem ser ativados ou desativados. Quando um `JCheckBoxMenuItem` é selecionado, uma marca de verificação aparece à esquerda do item de menu. Quando o `JCheckBoxMenuItem` é selecionado novamente, a marca é removida.

A classe **`JRadioButtonMenuItem`** (uma subclasse de `javax.swing.JMenuItem`) contém os métodos necessários para gerenciar itens de menu que podem ser ativados ou desativados, como os `JCheckBoxMenuItem`s. Quando múltiplos `JRadioButtonMenuItem`s são mantidos como parte de um `ButtonGroup`, apenas um item do grupo pode ser selecionado de cada vez. Quando um `JRadioButtonMenuItem` é selecionado, um círculo preenchido aparece à esquerda do item de menu. Quando outro `JRadioButtonMenuItem` é selecionado, o círculo preenchido do item de menu anteriormente selecionado é removido.

Utilizando menus em um aplicativo

As figuras 25.5–25.6 demonstram vários itens de menu e como especificar caracteres especiais chamados **mnemônicos** que podem fornecer acesso rápido a um menu ou item de menu a partir do teclado. Os mnemônicos podem ser utilizados com todas as subclasses de `javax.swing.AbstractButton`. A classe `MenuFrame` (Figura 25.5) cria a GUI e trata os eventos de item de menu. A maior parte do código nesse aplicativo aparece no construtor da classe (linhas 34–151).

```
1  // Figura 25.5: MenuFrame.java
2  // Demonstrando menus.
3  import java.awt.Color;
4  import java.awt.Font;
5  import java.awt.BorderLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ItemListener;
9  import java.awt.event.ItemEvent;
```

```
10 import javax.swing.JFrame;
11 import javax.swing.JRadioButtonMenuItem;
12 import javax.swing.JCheckBoxMenuItem;
13 import javax.swing.JOptionPane;
14 import javax.swing.JLabel;
15 import javax.swing.SwingConstants;
16 import javax.swing.ButtonGroup;
17 import javax.swing.JMenu;
18 import javax.swing.JMenuItem;
19 import javax.swing.JMenuBar;
20
21 public class MenuFrame extends JFrame
22 {
23    private final Color[] colorValues =
24       { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
25    private JRadioButtonMenuItem[] colorItems; // itens do menu Color
26    private JRadioButtonMenuItem[] fonts; // itens do menu Font
27    private JCheckBoxMenuItem[] styleItems; // itens do menu Font Style
28    private JLabel displayJLabel; // exibe texto de exemplo
29    private ButtonGroup fontButtonGroup; // gerencia itens do menu Font
30    private ButtonGroup colorButtonGroup; // gerencia itens do menu Color
31    private int style; // utilizado para criar estilos de fontes
32
33    // construtor sem argumento para configurar a GUI
34    public MenuFrame()
35    {
36       super( "Using JMenus" );
37
38       JMenu fileMenu = new JMenu( "File" ); // cria o menu File
39       fileMenu.setMnemonic( 'F' ); // configura o mnemônico como F
40
41       // cria item de menu About...
42       JMenuItem aboutItem = new JMenuItem( "About..." );
43       aboutItem.setMnemonic( 'A' ); // configura o mnemônico com A
44       fileMenu.add( aboutItem ); // adiciona o item about ao menu File
45       aboutItem.addActionListener(
46
47          new ActionListener() // classe interna anônima
48          {
49             // exibe um diálogo de mensagem quando o usuário seleciona About...
50             public void actionPerformed( ActionEvent event )
51             {
52                JOptionPane.showMessageDialog( MenuFrame.this,
53                   "This is an example\nof using menus",
54                   "About", JOptionPane.PLAIN_MESSAGE );
55             } // fim do método actionPerformed
56          } // fim da classe interna anônima
57       ); // fim da chamada para addActionListener
58
59       JMenuItem exitItem = new JMenuItem( "Exit" ); // cria o item exit
60       exitItem.setMnemonic( 'x' ); // configura o mnemônico como x
61       fileMenu.add( exitItem ); // adiciona o item exit ao menu File
62       exitItem.addActionListener(
63
64          new ActionListener() // classe interna anônima
65          {
66             // termina o aplicativo quando o usuário clica exitItem
67             public void actionPerformed( ActionEvent event )
68             {
69                System.exit( 0 ); // encerra o aplicativo
70             } // fim do método actionPerformed
71          } // fim da classe interna anônima
72       ); // fim da chamada para addActionListener
73
74       JMenuBar bar = new JMenuBar(); // cria a barra de menus
75       setJMenuBar( bar ); // adiciona uma barra de menus ao aplicativo
76       bar.add( fileMenu ); // adiciona o menu File à barra de menus
77
```

```
78          JMenu formatMenu = new JMenu( "Format" ); // cria o menu Format
79          formatMenu.setMnemonic( 'r' ); // configura o mnemônico como r
80
81          // array listando cores de string
82          String[] colors = { "Black", "Blue", "Red", "Green" };
83
84          JMenu colorMenu = new JMenu( "Color" ); // cria o menu Color
85          colorMenu.setMnemonic( 'C' ); // configura o mnemônico como C
86
87          // cria itens de menu de botão de opção para cores
88          colorItems = new JRadioButtonMenuItem[ colors.length ];
89          colorButtonGroup = new ButtonGroup(); // gerencia cores
90          ItemHandler itemHandler = new ItemHandler(); // handler para cores
91
92          // cria itens do menu Color com botões de opção
93          for ( int count = 0; count < colors.length; count++ )
94          {
95             colorItems[ count ] =
96                new JRadioButtonMenuItem( colors[ count ] ); // cria o item
97             colorMenu.add( colorItems[ count ] ); // adiciona o item ao menu Color
98             colorButtonGroup.add( colorItems[ count ] ); // adiciona ao grupo
99             colorItems[ count ].addActionListener( itemHandler );
100         } // for final
101
102         colorItems[ 0 ].setSelected( true ); // seleciona o primeiro item Color
103
104         formatMenu.add( colorMenu ); // adiciona o menu Color ao menu Format
105         formatMenu.addSeparator(); // adiciona um separador no menu
106
107         // array listando nomes de fonte
108         String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
109         JMenu fontMenu = new JMenu( "Font" ); // cria a fonte do menu
110         fontMenu.setMnemonic( 'n' ); // configura o mnemônico como n
111
112         // cria itens do menu radio button para nomes de fonte
113         fonts = new JRadioButtonMenuItem[ fontNames.length ];
114         fontButtonGroup = new ButtonGroup(); // gerencia os nomes das fontes
115
116         // criar itens do menu Font com botões de opção
117         for ( int count = 0; count < fonts.length; count++ )
118         {
119            fonts[ count ] = new JRadioButtonMenuItem( fontNames[ count ] );
120            fontMenu.add( fonts[ count ] ); // adiciona fonte ao menu Font
121            fontButtonGroup.add( fonts[ count ] ); // adiciona ao grupo de botões
122            fonts[ count ].addActionListener( itemHandler ); // adiciona handler
123         } // for final
124
125         fonts[ 0 ].setSelected( true ); // seleciona o primeiro item do menu Font
126         fontMenu.addSeparator(); // adiciona uma barra separadora ao menu Font
127
128         String[] styleNames = { "Bold", "Italic" }; // nomes dos estilos
129         styleItems = new JCheckBoxMenuItem[ styleNames.length ];
130         StyleHandler styleHandler = new StyleHandler(); // handler de estilo
131
132         // criar itens do menu Style com caixas de seleção
133         for ( int count = 0; count < styleNames.length; count++ )
134         {
135            styleItems[ count ] =
136               new JCheckBoxMenuItem( styleNames[ count ] ); // para estilo
137            fontMenu.add( styleItems[ count ] ); // adiciona ao menu Font
138            styleItems[ count ].addItemListener( styleHandler ); // handler
139         } // for final
140
141         formatMenu.add( fontMenu ); // adiciona o menu Font ao menu Format
142         bar.add( formatMenu ); // adiciona o menu Format à barra de menus
143
144         // configura o rótulo para exibir texto
145         displayJLabel = new JLabel( "Sample Text", SwingConstants.CENTER );
```

```
146          displayJLabel.setForeground( colorValues[ 0 ] );
147          displayJLabel.setFont( new Font( "Serif", Font.PLAIN, 72 ) );
148
149          getContentPane().setBackground( Color.CYAN ); // configura o fundo
150          add( displayJLabel, BorderLayout.CENTER ); // adiciona displayJLabel
151       } // fim do construtor de MenuFrame
152
153       // classe interna para tratar eventos de ação dos itens de menu
154       private class ItemHandler implements ActionListener
155       {
156          // processa seleções de cor e fonte
157          public void actionPerformed( ActionEvent event )
158          {
159             // processa a seleção de cor
160             for ( int count = 0; count < colorItems.length; count++ )
161             {
162                if ( colorItems[ count ].isSelected() )
163                {
164                   displayJLabel.setForeground( colorValues[ count ] );
165                   break;
166                } // fim do if
167             } // for final
168
169             // processa a seleção de fonte
170             for ( int count = 0; count < fonts.length; count++ )
171             {
172                if ( event.getSource() == fonts[ count ] )
173                {
174                   displayJLabel.setFont(
175                      new Font( fonts[ count ].getText(), style, 72 ) );
176                } // fim do if
177             } // for final
178
179             repaint(); // redesenha o aplicativo
180          } // fim do método actionPerformed
181       } // fim da classe ItemHandler
182
183       // classe interna para tratar eventos de itens dos itens de menu da caixa de verificação
184       private class StyleHandler implements ItemListener
185       {
186          // processa seleções de estilo da fonte
187          public void itemStateChanged( ItemEvent e )
188          {
189             String name = displayJLabel.getFont().getName(); // Font atual
190             Font font; // nova fonte baseada nas seleções de usuário
191
192             // determina quais itens são verificados e cria Font
193             if ( styleItems[ 0 ].isSelected() &&
194                  styleItems[ 1 ].isSelected() )
195                font = new Font( name, Font.BOLD + Font.ITALIC, 72 );
196             else if ( styleItems[ 0 ].isSelected() )
197                font = new Font( name, Font.BOLD, 72 );
198             else if ( styleItems[ 1 ].isSelected() )
199                font = new Font( name, Font.ITALIC, 72 );
200             else
201                font = new Font( name, Font.PLAIN, 72 );
202
203             displayJLabel.setFont( font );
204             repaint(); // redesenha o aplicativo
205          } // fim do método itemStateChanged
206       } // fim da classe StyleHandler
207    } // fim da classe MenuFrame
```

Figura 25.5 | JMenus e mnemônicos.

```
1  // Figura 25.6: MenuTest.java
2  // Testando MenuFrame.
3  import javax.swing.JFrame;
4
5  public class MenuTest
6  {
7     public static void main( String[] args )
8     {
9        MenuFrame menuFrame = new MenuFrame(); // criar MenuFrame
10       menuFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       menuFrame.setSize( 500, 200 ); // configura o tamanho do frame
12       menuFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe MenuTest
```

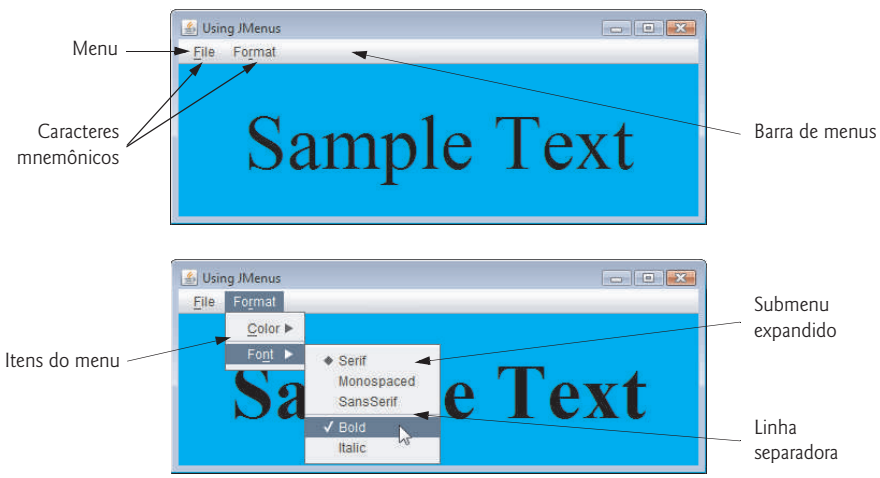


Figura 25.6 | Classe de teste para `MenuFrame`.

Configurando o menu File

As linhas 38–76 configuram o menu **File** e o anexam à barra de menus. O menu **File** contém um item de menu **About...** que exibe um diálogo de mensagem quando o item de menu é selecionado e um item de menu **Exit** que pode ser selecionado para terminar o aplicativo.

A linha 38 cria um `JMenu` e passa para o construtor a string `"File"` como o nome do menu. A linha 39 utiliza o método `JMenu` **`setMnemonic`** (herdado da classe `AbstractButton`) para indicar que `F` é o mnemônico desse menu. Pressionar a tecla *Alt* e a letra *F* abre o menu da mesma maneira que clicar o nome de menu com o mouse abriria. Na GUI, o caractere de mnemônico no nome do menu é exibido com um sublinhado. (Ver capturas de tela na Figura 25.6.)

Observações sobre a aparência e o comportamento 25.2

Os mnemônicos fornecem acesso rápido a comandos de menu e comandos de botão pelo teclado.

Observações sobre a aparência e o comportamento 25.3

Diferentes mnemônicos devem ser utilizados para cada botão ou item de menu. Normalmente, a primeira letra do rótulo no item de menu ou botão é utilizada como o mnemônico. Se diversos botões ou itens de menu iniciam com a mesma letra, escolha a próxima letra mais significativa do nome (por exemplo, x *comumente é escolhido para um botão* Exit *ou um item do menu). Mnemônicos não fazem distinção entre maiúsculas e minúsculas.*

As linhas 42–43 criam `JMenuItem` `aboutItem` com o texto "`About...`" e configuram seu mnemônico como a letra `A`. Esse item de menu é adicionado a `fileMenu` na linha 44 com o método **`add`** JMenu. Para acessar o item de menu **About...** pelo teclado, pressione a tecla *Alt* e a letra *F* para abrir o menu **File**, então pressione *A* para selecionar o item de menu **About...**. As linhas 47–56 criam um `ActionListener` para processar o evento de ação de `aboutItem`. As linhas 52–54 exibem uma caixa de diálogo de mensagem. Na maioria das utilizações anteriores de `showMessageDialog`, o primeiro argumento era `null`. O propósito do primeiro argumento é especificar a **janela-pai** que ajuda a determinar onde a caixa de diálogo será exibida. Se a janela-pai for especificada como `null`, a caixa de diálogo aparecerá no centro da tela. Caso contrário, ela aparece centralizada na janela-pai especificada. Neste exemplo, o programa especifica a janela-pai com `Menu-`

`Frame.this` — a referência `this` do objeto `MenuFrame`. Ao utilizar a referência `this` em uma classe interna, especificar `this` sozinha faz referência ao objeto da classe interna. Para representar objetos da classe externa com a referência `this`, qualifique `this` com o nome da classe externa e um ponto (`.`).

Em geral, caixas de diálogo são modais. Uma caixa de diálogo modal não permite qualquer outra janela do aplicativo ser acessada até que a caixa de diálogo seja fechada. Os diálogos exibidos com a classe `JOptionPane` são diálogos modais. A classe **`JDialog`** pode ser utilizada para criar seus próprios diálogos modais ou não modais.

As linhas 59–72 criam o item de menu `exitItem`, configuram seus mnemônicos como `x`, adicionam a `fileMenu` e registram um `ActionListener` que termina o programa quando o usuário seleciona `exit-Item`.

As linhas 74–76 criam o `JMenuBar`, o anexam à janela com o método setJMenuBar JFrame e usam o método **`add`** JMenuBar para anexar o `fileMenu` ao `JMenuBar`.

Erro comum de programação 25.3

Esquecer-se de configurar a barra de menus com o método `JFrame` `setJMenuBar` impede que a barra de menus seja exibida no `JFrame`.

Observações sobre a aparência e o comportamento 25.4

Os menus aparecem da esquerda para a direita na ordem em que eles são adicionados a um `JMenuBar`.

Configurando o menu Format

As linhas 78–79 criam o menu `formatMenu` e configuram seu mnemônico como `r`. (`F` não é utilizado porque esse é o mnemônico do menu **File**).

As linhas 84–85 criam o menu `colorMenu` (que será um submenu no menu **Format**) e configuram seu mnemônico como `C`. A linha 88 cria o array `JRadioButtonMenuItem` `colorItems` que fará referência aos itens do menu em `colorMenu`. A linha 89 cria o `ButtonGroup` `colorButtonGroup`, que irá assegurar que somente um dos itens de menu no submenu **Color** seja selecionado de cada vez. A linha 90 cria uma instância da classe interna `ItemHandler` (declarada nas linhas 154–181) que responde a seleções nos submenus **Color** e **Font** (discutidos mais adiante). A instrução `for` nas linhas 93–100 cria cada `JRadioButtonMenuItem` no array `colorItems`, adiciona cada item de menu a `colorMenu` e a `colorButtonGroup` e registra o `ActionListener` para cada item de menu.

A linha 102 invoca o método `AbstractButton` **`setSelected`** para selecionar o primeiro elemento no array `colorItems`. A linha 104 adiciona `colorMenu` como um submenu de `formatMenu`. A linha 105 invoca o método `JMenu` **`addSeparator`** para adicionar uma linha **separadora** horizontal ao menu.

Observações sobre a aparência e o comportamento 25.5

Um submenu é criado adicionando um menu como um item de menu a outro menu. Quando o mouse é posicionado sobre um submenu (ou o mnemônico do submenu é pressionado), o submenu expande para mostrar seus itens de menu.

Observações sobre a aparência e o comportamento 25.6

Separadores podem ser adicionados a um menu para agrupar itens de menu logicamente.

Observações sobre a aparência e o comportamento 25.7

Qualquer componente GUI "leve" (isto é, um componente que é uma subclasse de `JComponent`) pode ser adicionado a um `JMenu` ou a um `JMenuBar`.

As linhas 108–126 criam o submenu **Font**, vários `JRadioButtonMenuItems` e selecionam o primeiro elemento do array `JRadioButtonMenuItem` `fonts`. A linha 129 cria um array `JCheckBoxMenuItem` a fim de representar os itens de menu para especificar os estilos negrito e itálico para as fontes. A linha 130 cria uma instância da classe interna `StyleHandler` (declarada nas linhas 184–206) para responder aos eventos de `JCheckBoxMenuItem`. A estrutura `for` nas linhas 133–139 cria cada `JCheckBoxMenuItem`, adiciona cada item de menu ao `fontMenu` e registra o `ItemListener` para cada item de menu. A linha 141 adiciona `fontMenu` como um submenu de `formatMenu`. A linha 142 adiciona o `formatMenu` a `bar` (a barra de menus).

Criando o restante da GUI e definindo os handlers de evento

As linhas 145–147 criam um `JLabel` para o qual os itens do menu **Format** controlam a fonte, cor de fonte e estilo de fonte. A cor inicial do primeiro plano é configurada como o primeiro elemento do array `colorValues` (`Color.BLACK`) invocando o método **`setForeground`** **`JComponent`**. A fonte inicial é configurada como `Serif` com o estilo `PLAIN` e tamanho em pontos `72`. A linha 149 configura a cor de fundo do painel de conteúdo da janela como ciano (azul), e a linha 150 anexa o `JLabel` ao `CENTER` do painel de conteúdo `BorderLayout`.

O método `actionPerformed` da classe `ItemHandler` (linhas 157–180) utiliza duas instruções `for` para determinar qual item de menu de fonte ou de cor gerou o evento e configura a fonte ou a cor do `JLabel displayLabel`, respectivamente. A condição `if` na linha 162 utiliza o método `AbstractButton` **`isSelected`** para determinar o `JRadioButtonMenuItem` selecionado. A condição `if` na linha 172 invoca o objeto de evento do método `getSource` para obter uma referência ao `JRadioButtonMenuItem` que gerou o evento. A linha 175 invoca o método `AbstractButton getText` para obter o nome da fonte a partir do item de menu.

O método `StyleHandler itemStateChanged` (linhas 187–205) é chamado se o usuário selecionar um `JCheckBoxMenuItem` no `fontMenu`. As linhas 193–201 determinam quais `JCheckBoxMenuItem` estão selecionados e utilizam seu estado combinado para determinar o novo estilo de fonte.

25.5 JPopupMenu

Muitos dos aplicativos atuais de computador fornecem os chamados **menus pop-up sensíveis ao contexto**. No Swing, esses menus são criados com a classe **`JPopupMenu`** (uma subclasse de `JComponent`). Esses menus fornecem opções que são específicas do componente pelo qual o **evento de gatilho pop-up** foi gerado. Na maioria dos sistemas, o evento de acionamento de pop-up ocorre quando o usuário pressiona e libera o botão direito do mouse.

Observações sobre a aparência e o comportamento 25.8

O evento de acionamento do pop-up é específico da plataforma. Na maioria das plataformas que utilizam um mouse com múltiplos botões, o evento de acionamento do pop-up ocorre quando o usuário clica com o botão direito do mouse em um componente que suporta um menu pop-up.

O aplicativo nas figuras 25.7–25.8 cria um `JPopupMenu` que permite ao usuário selecionar uma de três cores e alterar a cor de fundo da janela. Quando o usuário clica com o botão direito do mouse sobre o fundo da janela `PopupFrame`, aparece um `JPopupMenu` que contêm cores. Se o usuário clicar em um `JRadioButtonMenuItem` de uma cor, o método `ItemHandler actionPerformed` altera a cor de fundo do painel de conteúdo da janela.

A linha 25 do construtor `PopupFrame` (Figura 25.7, linhas 21–69) cria uma instância da classe `ItemHandler` (declarada nas linhas 72–87) que processará os eventos dos itens a partir dos itens de menu do menu pop-up. A linha 29 cria o `JPopupMenu`. A instrução `for` (linhas 33–39) cria um objeto `JRadioButtonMenuItem` (linha 35), adiciona-o a `popupMenu` (linha 36), adiciona-o a `ButtonGroup colorGroup` (linha 37) para manter um `JRadioButtonMenuItem` selecionado de cada vez e registra seu `ActionListener` (linha 38). A linha 41 configura o fundo inicial como branco invocando o método `setBackground`.

```
1  // Figura 25.7: PopupFrame.java
2  // Demonstrando JPopupMenus.
3  import java.awt.Color;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JRadioButtonMenuItem;
10 import javax.swing.JPopupMenu;
11 import javax.swing.ButtonGroup;
12
13 public class PopupFrame extends JFrame
14 {
15    private JRadioButtonMenuItem[] items; // contém itens para cores
16    private final Color[] colorValues =
17       { Color.BLUE, Color.YELLOW, Color.RED }; // cores a serem utilizadas
18    private JPopupMenu popupMenu; // permite que o usuário selecione a cor
19
20    // construtor sem argumento configure a GUI
21    public PopupFrame()
22    {
23       super( "Using JPopupMenus" );
24
25       ItemHandler handler = new ItemHandler(); // handler para itens de menu
26       String[] colors = { "Blue", "Yellow", "Red" }; // array de cores
27
28       ButtonGroup colorGroup = new ButtonGroup(); // gerencia itens de cor
29       popupMenu = new JPopupMenu(); // cria menu pop-up
30       items = new JRadioButtonMenuItem[ colors.length ]; // aplica cores aos itens
31
```

```
32          // cria item de menu, adiciona-o ao menu pop-up, permite tratamento de eventos
33          for ( int count = 0; count < items.length; count++ )
34          {
35             items[ count ] = new JRadioButtonMenuItem( colors[ count ] );
36             popupMenu.add( items[ count ] ); // adiciona o item ao menu pop-up
37             colorGroup.add( items[ count ] ); // adiciona o item ao grupo de botões
38             items[ count ].addActionListener( handler ); // adiciona handler
39          } // for final
40
41          setBackground( Color.WHITE ); // configura o fundo como branco
42
43          // declara um MouseListener para a janela a fim de exibir o menu pop-up
44          addMouseListener(
45
46             new MouseAdapter() // classe interna anônima
47             {
48                // trata eventos de pressionamento do mouse
49                public void mousePressed( MouseEvent event )
50                {
51                   checkForTriggerEvent( event ); // verifica o acionamento
52                } // fim do método mousePressed
53
54                // trata eventos de liberação de botão do mouse
55                public void mouseReleased( MouseEvent event )
56                {
57                   checkForTriggerEvent( event ); // verifica o acionamento
58                } // fim do método mouseReleased
59
60                // determina se o evento deve acionar o menu de pop-up
61                private void checkForTriggerEvent( MouseEvent event )
62                {
63                   if ( event.isPopupTrigger() )
64                      popupMenu.show(
65                         event.getComponent(), event.getX(), event.getY() );
66                } // fim do método checkForTriggerEvent
67             } // fim da classe interna anônima
68          ); // fim da chamada para addMouseListener
69       } // fim do construtor PopupFrame
70
71       // classe interna privada para tratar eventos de item de menu
72       private class ItemHandler implements ActionListener
73       {
74          // processa seleções de itens de menu
75          public void actionPerformed( ActionEvent event )
76          {
77             // determina qual item de menu foi selecionado
78             for ( int i = 0; i < items.length; i++ )
79             {
80                if ( event.getSource() == items[ i ] )
81                {
82                   getContentPane().setBackground( colorValues[ i ] );
83                   return;
84                } // fim do if
85             } // for final
86          } // fim do método actionPerformed
87       } // fim da classe interna privada ItemHandler
88    } // fim da classe PopupFrame
```

Figura 25.7 | JPopupMenu para selecionar cores.

```
1  // Figura 25.8: PopupTest.java
2  // Testando PopupFrame.
3  import javax.swing.JFrame;
4
5  public class PopupTest
6  {
7     public static void main( String[] args )
```

```
8       {
9          PopupFrame popupFrame = new PopupFrame(); // cria PopupFrame
10         popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11         popupFrame.setSize( 300, 200 ); // configura o tamanho do frame
12         popupFrame.setVisible( true ); // exibe o frame
13      } // fim de main
14   } // fim da classe PopupTest
```

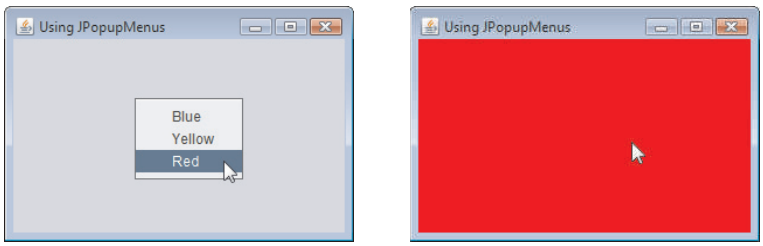


Figura 25.8 | Classe de teste para `PopupFrame`.

As linhas 44–68 registram um `MouseListener` para tratar os eventos de mouse da janela de aplicativo. Os métodos `mousePressed` (linhas 49–52) e `mouseReleased` (linhas 55–58) verificam o evento de acionamento do pop-up. Cada método chama o método utilitário privado `checkForTriggerEvent` (linhas 61–66) para determinar se o evento de disparo de pop-up ocorreu. Se ocorreu, o método `MouseEvent` **`isPopupTrigger`** retorna `true` e o método `JPopupMenu` **`show`** exibe o `JPopupMenu`. O primeiro argumento para o método `show` especifica o **componente de origem**, cuja posição ajuda a determinar onde o `JPopupMenu` aparecerá na tela. Os dois últimos argumentos são as coordenadas *x* e *y* (medidas do canto superior esquerdo do componente de origem) em que o `JPopupMenu` deve aparecer.

Observações sobre a aparência e o comportamento 25.9

Exibir um `JPopupMenu` para o evento de acionamento de pop-up de múltiplos componentes GUI requer o registro de handlers de eventos de mouse para cada um desses componentes GUI.

Quando o usuário seleciona um item no menu pop-up, o método `actionPerformed` da classe `ItemHandler` (linhas 75–86) determina qual `JRadioButtonMenuItem` o usuário selecionou e configura a cor de fundo do painel de conteúdo da janela.

25.6 Aparência e comportamento plugável

Um programa que utiliza componentes AWT GUI do Java (pacote `java.awt`) assume a aparência e o comportamento da plataforma em que o programa executa. Um aplicativo Java em execução em um Mac OS X se parece com outros aplicativos Mac OS X. Um em execução no Microsoft Windows se parece com outros aplicativos Windows. Um em execução em uma plataforma UNIX se parece com outros aplicativos nessa plataforma UNIX. Às vezes, isso é desejável porque permite que usuários do aplicativo em cada plataforma utilizem componentes GUI com os quais eles já estão familiarizados. Mas isso também introduz questões interessantes de portabilidade.

Dica de portabilidade 25.1

Os componentes GUI têm uma aparência distinta em diferentes plataformas e talvez requeiram quantidades diferentes de espaço para serem exibidos. Isso poderia alterar os layouts e alinhamentos da GUI.

Dica de portabilidade 25.2

Os componentes GUI em plataformas distintas têm funcionalidade padrão diferente (por exemplo, algumas plataformas permitem que um botão com o foco seja "pressionado" com a barra de espaço e outras não).

Os componentes GUI leves do Swing eliminam muitas dessas questões fornecendo funcionalidades uniformes entre diferentes plataformas e definindo uma aparência e funcionamento uniformes entre várias plataformas. A partir do Java SE 6 Update 10, essa é a aparência e funcionamento do Nimbo que discutimos na Seção 14.2. As versões anteriores do Java utilizavam a **aparência e funcionamento metal**, que ainda é o padrão. O Swing também fornece a flexibilidade para personalizar a aparência e comportamento com um estilo Microsoft Windows (somente em sistemas Windows), um estilo Motif (UNIX) (em diferentes plataformas) ou um estilo Macintosh (somente em sistemas Mac).

As figuras 25.9–25.10 demonstram um modo de alterar a aparência e funcionamento de uma GUI do Swing. Ele cria vários componentes GUI, assim você pode ver a modificação na aparência e funcionamento ao mesmo tempo. As janelas de saída mostram as aparências e funcionamentos do Metal, Nimbus, CDE/Motif, Windows e Windows Classic disponíveis nos sistemas Windows. As aparências e funcionamentos instalados irão variar entre uma e outra plataforma.

```
1  // Figura 25.9: LookAndFeelFrame.java
2  // Alterando a aparência e o comportamento.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.UIManager;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11 import javax.swing.JButton;
12 import javax.swing.JLabel;
13 import javax.swing.JComboBox;
14 import javax.swing.JPanel;
15 import javax.swing.SwingConstants;
16 import javax.swing.SwingUtilities;
17
18 public class LookAndFeelFrame extends JFrame
19 {
20    private UIManager.LookAndFeelInfo[] looks; // aparências e comportamentos
21    private String[] lookNames; // nomes das aparências e comportamentos
22    private JRadioButton[] radio; // botões de opção para selecionar a aparência e o funcionamento
23    private ButtonGroup group; // grupo de botões de opção
24    private JButton button; // exibe a aparência do botão
25    private JLabel label; // exibe a aparência do rótulo
26    private JComboBox comboBox; // exibe a aparência da caixa de combinação
27
28    // configura a GUI
29    public LookAndFeelFrame()
30    {
31       super( "Look and Feel Demo" );
32
33       // obtém as informações sobre a aparência e o comportamento instalados
34       looks = UIManager.getInstalledLookAndFeels();
35       lookNames = new String[ looks.length ];
36
37       // obtém os nomes das aparências e funcionamentos instalados
38       for ( int i = 0; i < looks.length; i++ )
39          lookNames[ i ] = looks[ i ].getName();
40
41       JPanel northPanel = new JPanel(); // cria o painel North
42       northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
43
44       label = new JLabel( "This is a " + lookNames[0] + " look-and-feel",
45          SwingConstants.CENTER ); // cria o rótulo
46       northPanel.add( label ); // adiciona o rótulo ao painel
47
48       button = new JButton( "JButton" ); // cria o botão
49       northPanel.add( button ); // adiciona botão ao painel
50
51       comboBox = new JComboBox( lookNames ); // cria a caixa de combinação
52       northPanel.add( comboBox ); // adiciona a caixa de combinação ao painel
53
54       // cria um array para botões de opção
55       radio = new JRadioButton[ looks.length ];
56
57       JPanel southPanel = new JPanel(); // cria o painel South
58
59       // utiliza um GridLayout com 3 botões em cada linha
60       int rows = (int) Math.ceil( radio.length / 3.0 );
61       southPanel.setLayout( new GridLayout( rows, 3 ) );
62
63       group = new ButtonGroup(); // grupo de botões para aparências e comportamentos
64       ItemHandler handler = new ItemHandler(); // handler de aparência e comportamento
65
66       for ( int count = 0; count < radio.length; count++ )
67       {
68          radio[ count ] = new JRadioButton( lookNames[ count ] );
```

```
            radio[ count ].addItemListener( handler ); // adiciona handler
            group.add( radio[ count ] ); // adiciona o botão de opção ao grupo
            southPanel.add( radio[ count ] ); // adiciona botão de opções ao painel
         } // for final

         add( northPanel, BorderLayout.NORTH ); // adiciona o painel North
         add( southPanel, BorderLayout.SOUTH ); // adiciona o painel South

         radio[ 0 ].setSelected( true ); // configura a seleção padrão
      } // fim do construtor LookAndFeelFrame

      // utiliza UIManager para alterar a aparência e o comportamento da GUI
      private void changeTheLookAndFeel( int value )
      {
         try // muda a aparência e o comportamento
         {
            // configura a aparência e o comportamento para esse aplicativo
            UIManager.setLookAndFeel( looks[ value ].getClassName() );

            // atualiza os componentes nesse aplicativo
            SwingUtilities.updateComponentTreeUI( this );
         } // fim do try
         catch ( Exception exception )
         {
            exception.printStackTrace();
         } // fim do catch
      } // fim do método changeTheLookAndFeel

      // classe interna private para tratar eventos de botão de opção
      private class ItemHandler implements ItemListener
      {
         // processa a seleção de aparência e comportamento feita pelo usuário
         public void itemStateChanged( ItemEvent event )
         {
            for ( int count = 0; count < radio.length; count++ )
            {
               if ( radio[ count ].isSelected() )
               {
                  label.setText( String.format(
                     "This is a %s look-and-feel", lookNames[ count ] ) );
                  comboBox.setSelectedIndex( count ); // configura o índice da caixa de combinação
                  changeTheLookAndFeel( count ); // muda a aparência e o comportamento
               } // fim do if
            } // for final
         } // fim do método itemStateChanged
      } // fim da classe interna privada ItemHandler
   } // fim da classe LookAndFeelFrame
```

Figura 25.9 | Aparência e comportamento de uma GUI baseada no Swing.

```
// Figura 25.10: LookAndFeelDemo.java
// Alterando a aparência e o comportamento.
import javax.swing.JFrame;

public class LookAndFeelDemo
{
   public static void main( String[] args )
   {
      LookAndFeelFrame lookAndFeelFrame = new LookAndFeelFrame();
      lookAndFeelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      lookAndFeelFrame.setSize( 400, 220 ); // configura o tamanho do frame
      lookAndFeelFrame.setVisible( true ); // exibe o frame
   } // fim de main
} // fim da classe LookAndFeelDemo
```

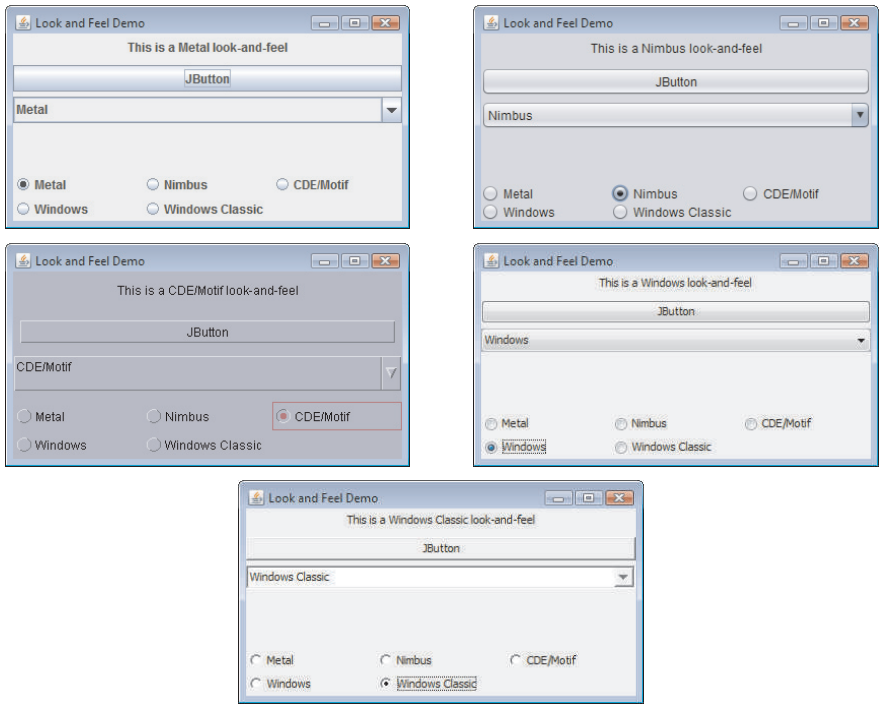


Figura 25.10 | Classe de teste para `LookAndFeelFrame`.

Cobrimos os componentes GUI e conceitos que tratam o evento nesse exemplo anteriormente, portanto aqui focalizamos o mecanismo para alterar a aparência e o funcionamento. A classe **`UIManager`** (pacote `javax.swing`) contém uma classe aninhada **`LookAndFeelInfo`** (uma classe `public static`) que mantém as informações sobre uma aparência e um comportamento. A linha 20 declara um array do tipo `UIManager.LookAndFeelInfo` (observe a sintaxe utilizada para identificar a classe interna `static LookAndFeelInfo`). A linha 34 utiliza o método **`getInstalledLookAndFeels`** de `UIManager` `static` para obter o array dos objetos `UIManager.LookAndFeelInfo` que descrevem cada aparência e comportamento disponível no seu sistema.

Dica de desempenho 25.2

Cada aparência e comportamento é representada por uma classe Java. O método `UIManager` `getInstalledLookAndFeels` não carrega cada classe. Em vez disso, fornece os nomes das classes de aparência e comportamento disponíveis de modo que uma escolha possa ser feita (presumivelmente uma vez na inicialização do programa). Isso reduz o overhead de ter de carregar todas as classes de aparência e comportamento mesmo se o programa não utilizar algumas delas.

Nosso método utilitário `changeTheLookAndFeel` (linhas 81–95) é chamado pelo handler de evento para os `JRadioButtons` na parte inferior da interface com o usuário. O handler de evento (declarado na classe interna `private ItemHandler` nas linhas 98–114) passa um inteiro que representa o elemento do array `looks` que deve ser utilizado para alterar a aparência e comportamento. A linha 86 invoca o método `static` **`setLookAndFeel`** de `UIManager` para alterar a aparência e comportamento. O método **`getClassName`** da classe `UIManager.LookAndFeelInfo` determina o nome da classe de aparência e comportamento que corresponde ao objeto `UIManager.LookAndFeelInfo`. Se a classe da aparência e comportamento ainda não estiver carregada, ela será carregada como parte da chamada a `setLookAndFeel`. A linha 89 invoca o método `static` **`updateComponentTreeUI`** da classe **`SwingUtilities`** (pacote `javax.swing`) para alterar a aparência e comportamento de cada componente GUI anexado ao seu argumento (instância `this` da nossa classe de aplicativo `LookAndFeelFrame`) para a nova aparência e comportamento.

25.7 JDesktopPane e JInternalFrame

Muitos aplicativos atuais utilizam uma **interface de múltiplos documentos** (**Multiple-Document Interface — MDI**) — uma janela principal (chamada **janela-pai**) contendo outras janelas (chamadas **janelas-filhas**), para gerenciar vários documentos abertos que estão sendo processados em paralelo. Por exemplo, muitos programas de correio eletrônico permitem ter várias janelas abertas ao mesmo tempo, assim você pode compor ou ler múltiplas mensagens simultaneamente. De maneira semelhante, vários processadores de texto permitem que o usuário abra múltiplos documentos em janelas separadas dentro de uma janela principal, tornando possível alternar entre elas sem fechar uma e abrir outra. O aplicativo nas figuras 25.11–25.12 demonstra as classes **`JDesktopPane`** e **`JInternalFrame`** do Swing para implementar interfaces de múltiplos documentos.

```
1  // Figura 25.11: DesktopFrame.java
2  // Demonstrando JDesktopPane
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.util.Random;
9  import javax.swing.JFrame;
10 import javax.swing.JDesktopPane;
11 import javax.swing.JMenuBar;
12 import javax.swing.JMenu;
13 import javax.swing.JMenuItem;
14 import javax.swing.JInternalFrame;
15 import javax.swing.JPanel;
16 import javax.swing.ImageIcon;
17
18 public class DesktopFrame extends JFrame
19 {
20    private JDesktopPane theDesktop;
21
22    // configura a GUI
23    public DesktopFrame()
24    {
25       super( "Using a JDesktopPane" );
26
27       JMenuBar bar = new JMenuBar(); // cria a barra de menus
28       JMenu addMenu = new JMenu( "Add" ); // cria o menu Add
29       JMenuItem newFrame = new JMenuItem( "Internal Frame" );
30
31       addMenu.add( newFrame ); // adiciona um novo item de quadro ao menu Add
32       bar.add( addMenu ); // adiciona o menu Add à barra de menus
33       setJMenuBar( bar ); // configura a barra de menus para esse aplicativo
34
35       theDesktop = new JDesktopPane(); // cria o painel de área de trabalho
36       add( theDesktop ); // adiciona painel de área de trabalho ao quadro
37
38       // configura o ouvinte para o item de menu newFrame
39       newFrame.addActionListener(
40
41          new ActionListener() // classe interna anônima
42          {
43             // exibe a nova janela interna
44             public void actionPerformed( ActionEvent event )
45             {
46                // cria o quadro interno
47                JInternalFrame frame = new JInternalFrame(
48                   "Internal Frame", true, true, true, true );
49
50                MyJPanel panel = new MyJPanel(); // cria um novo painel
51                frame.add( panel, BorderLayout.CENTER ); // adiciona o painel
52                frame.pack(); // configura o quadro interno de acordo com o tamanho do conteúdo
53
54                theDesktop.add( frame ); // anexa o quadro interno
55                frame.setVisible( true ); // mostra o quadro interno
56             } // fim do método actionPerformed
57          } // fim da classe interna anônima
58       ); // fim da chamada para addActionListener
59    } // fim do construtor DesktopFrame
60 } // fim da classe DesktopFrame
61
62 // classe para exibir um ImageIcon em um painel
63 class MyJPanel extends JPanel
64 {
65    private static Random generator = new Random();
66    private ImageIcon picture; // imagem a ser exibida
67    private final static String[] images = { "yellowflowers.png",
68       "purpleflowers.png", "redflowers.png", "redflowers2.png",
```

```
69          "lavenderflowers.png" };
70
71       // carrega a imagem
72       public MyJPanel()
73       {
74          int randomNumber = generator.nextInt( images.length );
75          picture = new ImageIcon( images[ randomNumber ] ); // configura o ícone
76       } // fim do construtor MyJPanel
77
78       // exibe imageIcon no painel
79       public void paintComponent( Graphics g )
80       {
81          super.paintComponent( g );
82          picture.paintIcon( this, g, 0, 0 ); // exibe o ícone
83       } // fim do método paintComponent
84
85       // retorna as dimensões da imagem
86       public Dimension getPreferredSize()
87       {
88          return new Dimension( picture.getIconWidth(),
89             picture.getIconHeight() );
90       } // fim do método getPreferredSize
91    } // fim da classe MyJPanel
```

Figura 25.11 | Interface de múltiplos documentos.

As linhas 12–33 definem um `JMenuBar`, um `JMenu` e um `JMenuItem`, para adicionar o `JMenuItem` a `JMenu`, adicionar o `JMenu` a `JMenuBar` e configurar o `JMenuBar` para a janela de aplicativo. Quando o usuário seleciona o `JMenuItem` `newFrame`, o aplicativo cria e exibe um novo objeto `JInternalFrame` contendo uma imagem.

A linha 35 atribui a variável `theDesktop` `JDesktopPane` (pacote `javax.swing`) a um novo objeto `JDesktopPane` que será utilizado para gerenciar as janelas-filhas de `JInternalFrame`. A linha 36 adiciona o `JDesktopPane` ao `JFrame`. Por padrão, o `JDesktopPane` é adicionado ao centro do painel de conteúdo `BorderLayout` e então o `JDesktopPane` se expande para preencher a janela inteira do aplicativo.

As linhas 39–58 registram um `ActionListener` para tratar o evento quando o usuário seleciona o item de menu `newFrame`. Quando o evento ocorre, o método `actionPerformed` (linhas 44–56) cria um objeto `JInternalFrame` nas linhas 47–48. O construtor `JInternalFrame` aqui utilizado recebe cinco argumentos — uma `String` para a barra de título da janela interna, um `boolean` indicando se o frame interno pode ser redimensionado pelo usuário, um `boolean` indicando se o frame interno pode ser fechado pelo usuário, um `boolean` indicando se o frame interno pode ser maximizado pelo usuário e um `boolean` indicando se o frame interno pode ser minimizado pelo usuário. Para cada um dos argumentos `boolean`, um valor `true` indica que a operação deve ser permitida (como é o caso aqui).

Como ocorre com `JFrames` e `JApplets`, um `JInternalFrame` tem um painel de conteúdo a que componentes GUI podem ser anexados. A linha 50 (Figura 25.11) cria uma instância da nossa classe `MyJPanel` (declarada nas linhas 63–91), que é adicionada ao `JInternalFrame` na linha 51.

```
1  // Figura 25.12: DesktopTest.java
2  // Demonstrando JDesktopPane.
3  import javax.swing.JFrame;
4
5  public class DesktopTest
6  {
7     public static void main( String[] args )
8     {
9        DesktopFrame desktopFrame = new DesktopFrame();
10       desktopFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       desktopFrame.setSize( 600, 480 ); // configura o tamanho do frame
12       desktopFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe DesktopTest
```

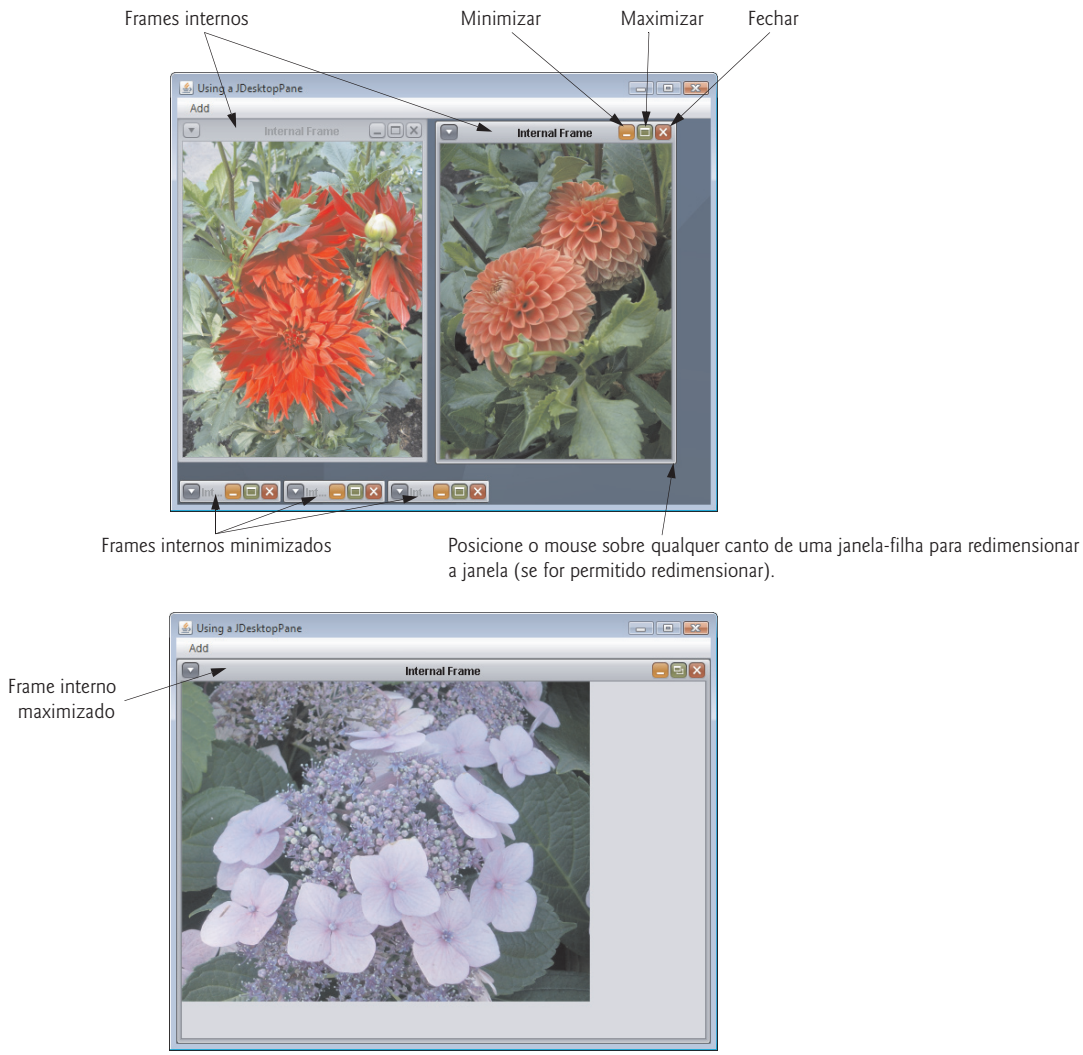


Figura 25.12 | Classe de teste para `DeskTopFrame`.

A linha 52 utiliza o método `JInternalFrame` **pack** para configurar o tamanho da janela-filha. O método `pack` utiliza os tamanhos preferidos dos componentes para determinar o tamanho da janela. A classe `MyJPanel` declara o método `getPreferredSize` (linhas 86–90) para especificar o tamanho preferido do painel para utilização pelo método `pack`. A linha 54 adiciona o `JInternalFrame` ao `JDesktopPane`, e a linha 55 exibe o `JInternalFrame`.

As classes `JInternalFrame` e `JDesktopPane` fornecem muitos métodos para gerenciar janelas-filhas. Consulte a documentação on-line da API de `JInternalFrame` e `JDesktopPane` para listas completas desses métodos:

```
java.sun.com/javase/6/docs/api/javax/swing/JInternalFrame.html
java.sun.com/javase/6/docs/api/javax/swing/JDesktopPane.html
```

25.8 JTabbedPane

Um **JTabbedPane** organiza componentes GUI em camadas, das quais somente uma é visível de cada vez. Os usuários acessam cada camada via uma guia — semelhante a pastas em um gabinete de arquivos. Quando o usuário clica em uma guia, a camada apropriada é exibida. As guias aparecem na parte superior por padrão, mas também podem ser posicionadas à esquerda, direita ou parte inferior do `JTabbedPane`. Qualquer componente pode ser posicionado em uma guia. Se o componente for um contêiner, como um painel, ele poderá utilizar qualquer gerenciador de layout para organizar vários componentes na guia. A classe `JTabbedPane` é uma subclasse de `JComponent`. O aplicativo nas figuras 25.13–25.14 cria um painel com guias com três guias. Cada guia exibe um dos `JPanel`s — `panel1`, `panel2` ou `panel3`.

```
1  // Figura 25.13: JTabbedPaneFrame.java
2  // Demonstrando o JTabbedPane.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JTabbedPane;
7  import javax.swing.JLabel;
8  import javax.swing.JPanel;
9  import javax.swing.JButton;
10 import javax.swing.SwingConstants;
11
12 public class JTabbedPaneFrame extends JFrame
13 {
14    // configura a GUI
15    public JTabbedPaneFrame()
16    {
17       super( "JTabbedPane Demo " );
18
19       JTabbedPane tabbedPane = new JTabbedPane(); // cria o JTabbedPane
20
21       // configura o pane11 e o adiciona ao JTabbedPane
22       JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
23       JPanel panel1 = new JPanel(); // cria o primeiro painel
24       panel1.add( label1 ); // adiciona o rótulo ao painel
25       tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );
26
27       // configura o panel2 e o adiciona a JTabbedPane
28       JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
29       JPanel panel2 = new JPanel(); // cria o segundo painel
30       panel2.setBackground( Color.YELLOW ); // configura o fundo como amarelo
31       panel2.add( label2 ); // adiciona o rótulo ao painel
32       tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );
33
34       // configura o panel3 e o adiciona a JTabbedPane
35       JLabel label3 = new JLabel( "panel three" );
36       JPanel panel3 = new JPanel(); // cria o terceiro painel
37       panel3.setLayout( new BorderLayout() ); // utilize o borderlayout
38       panel3.add( new JButton( "North" ), BorderLayout.NORTH );
39       panel3.add( new JButton( "West" ), BorderLayout.WEST );
40       panel3.add( new JButton( "East" ), BorderLayout.EAST );
41       panel3.add( new JButton( "South" ), BorderLayout.SOUTH );
42       panel3.add( label3, BorderLayout.CENTER );
43       tabbedPane.addTab( "Tab Three", null, panel3, "Third Panel" );
44
45       add( tabbedPane ); // adiciona o JTabbedPane ao quadro
46    } // fim do construtor JTabbedPaneFrame
47 } // fim da classe JTabbedPaneFrame
```

Figura 25.13 | `JTabbedPane` utilizado para organizar componentes GUI.

```
1  // Figura 25.14: JTabbedPaneDemo.java
2  // Demonstrando o JTabbedPane.
3  import javax.swing.JFrame;
4
5  public class JTabbedPaneDemo
6  {
7     public static void main( String[] args )
8     {
9        JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
10       tabbedPaneFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       tabbedPaneFrame.setSize( 250, 200 ); // configura o tamanho do frame
12       tabbedPaneFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe JTabbedPaneDemo
```

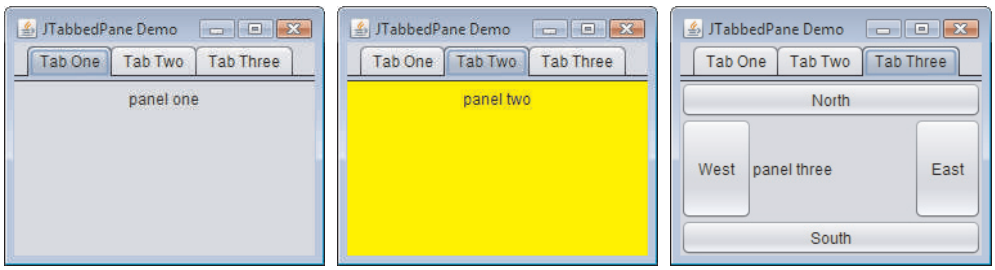


Figura 25.14 | Classe de teste para `JTabbedPaneFrame`.

O construtor (linhas 15–46) constrói a GUI. A linha 19 cria um `JTabbedPane` vazio com as configurações padrão, isto é, guias ao longo da parte superior. Se as guias não se ajustarem em uma linha, elas serão empacotadas a fim de formar linhas adicionais das guias. Em seguida, o construtor cria os `JPanels` `panel1`, `panel2` e `panel3` e seus componentes GUI. À medida que configuramos cada painel, nós o adicionamos ao `tabbedPane` utilizando o método `JTabbedPane` **`addTab`** com quatro argumentos. O primeiro argumento é uma `String` que especifica o título da guia. O segundo argumento é uma referência `Icon` que especifica um ícone a ser exibido na guia. Se o `Icon` for uma referência `null`, nenhuma imagem será exibida. O terceiro argumento é uma referência `Component` que representa o componente GUI a ser exibido quando o usuário clica na guia. O último argumento é uma `String` que especifica a dica de ferramenta para a guia. Por exemplo, a linha 25 adiciona o `JPanel` `panel1` ao `tabbedPane` com o título `"Tab One"` e a dica de ferramenta `"First Panel"`. Os `JPanels` `panel2` e `panel3` são adicionados ao `tabbedPane` nas linhas 32 e 43. Para visualizar uma guia, clique nela com o mouse ou utilize as teclas de seta para alternar as guias.

25.9 Gerenciadores de layout: `BoxLayout` e `GridBagLayout`

No Capítulo 14, introduzimos três gerenciadores de layout — `FlowLayout`, `BorderLayout` e `GridLayout`. Esta seção apresenta dois gerenciadores adicionais de layout (resumidos na Figura 25.15). Discutimos esses gerenciadores de layout nos exemplos que se seguem.

Gerenciador de layout	Descrição
`BoxLayout`	Um gerenciador de layout que permite que os componentes GUI sejam organizados da esquerda para a direita ou de cima para baixo em um contêiner. A classe `Box` declara um contêiner com `BoxLayout` como seu gerenciador padrão de layout e fornece métodos `static` para criar um `Box` com um `BoxLayout` horizontal ou vertical.
`GridBagLayout`	Um gerenciador de layout semelhante a `GridLayout`, mas os componentes podem variar de tamanho e podem ser adicionados em qualquer ordem.

Figura 25.15 | Gerenciadores de layout adicionais.

Gerenciador de layout `BoxLayout`

O gerenciador de layout `BoxLayout` (no pacote `javax.swing`) organiza os componentes GUI horizontalmente ao longo do eixo *x* ou verticalmente ao longo do eixo *y* de um contêiner. O aplicativo nas figuras 25.16–25.17 demonstra o `BoxLayout` e a classe contêiner `Box`, que utiliza o `BoxLayout` como seu gerenciador padrão de layout.

```
1  // Figura 25.16: BoxLayoutFrame.java
2  // Demonstrando BoxLayout.
3  import java.awt.Dimension;
4  import javax.swing.JFrame;
5  import javax.swing.Box;
6  import javax.swing.JButton;
7  import javax.swing.BoxLayout;
8  import javax.swing.JPanel;
9  import javax.swing.JTabbedPane;
10
11 public class BoxLayoutFrame extends JFrame
12 {
```

```
13       // configura a GUI
14       public BoxLayoutFrame()
15       {
16          super( "Demonstrating BoxLayout" );
17
18          // cria contêineres Box com BoxLayout
19          Box horizontal1 = Box.createHorizontalBox();
20          Box vertical1 = Box.createVerticalBox();
21          Box horizontal2 = Box.createHorizontalBox();
22          Box vertical2 = Box.createVerticalBox();
23
24          final int SIZE = 3; // número de botões em cada Box
25
26          // adiciona botões a Box horizontal1
27          for ( int count = 0; count < SIZE; count++ )
28             horizontal1.add( new JButton( "Button " + count ) );
29
30          // cria um suporte e adiciona botões a Box vertical1
31          for ( int count = 0; count < SIZE; count++ )
32          {
33             vertical1.add( Box.createVerticalStrut( 25 ) );
34             vertical1.add( new JButton( "Button " + count ) );
35          } // for final
36
37          // cria a cola horizontal e adiciona botões a Box horizontal2
38          for ( int count = 0; count < SIZE; count++ )
39          {
40             horizontal2.add( Box.createHorizontalGlue() );
41             horizontal2.add( new JButton( "Button " + count ) );
42          } // for final
43
44          // cria uma área rígida e adiciona botões a Box vertical2
45          for ( int count = 0; count < SIZE; count++ )
46          {
47             vertical2.add( Box.createRigidArea( new Dimension( 12, 8 ) ) );
48             vertical2.add( new JButton( "Button " + count ) );
49          } // for final
50
51          // cria cola vertical e adiciona botões ao painel
52          JPanel panel = new JPanel();
53          panel.setLayout(new BoxLayout( panel, BoxLayout.Y_AXIS ) );
54
55          for ( int count = 0; count < SIZE; count++ )
56          {
57             panel.add( Box.createGlue() );
58             panel.add( new JButton( "Button " + count ) );
59          } // for final
60
61          // cria um JTabbedPane
62          JTabbedPane tabs = new JTabbedPane(
63             JTabbedPane.TOP, JTabbedPane.SCROLL_TAB_LAYOUT );
64
65          // coloca cada contêiner no painel com guias
66          tabs.addTab( "Horizontal Box", horizontal1 );
67          tabs.addTab( "Vertical Box with Struts", vertical1 );
68          tabs.addTab( "Horizontal Box with Glue", horizontal2 );
69          tabs.addTab( "Vertical Box with Rigid Areas", vertical2 );
70          tabs.addTab( "Vertical Box with Glue", panel );
71
72          add( tabs ); // coloca o painel com guias no quadro
73       } // fim do construtor BoxLayoutFrame
74    } // fim da classe BoxLayoutFrame
```

Figura 25.16 | Gerenciador de layout `BoxLayout`.

```
1  // Figura 25.17: BoxLayoutDemo.java
2  // Demonstrando BoxLayout.
3  import javax.swing.JFrame;
4
5  public class BoxLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        BoxLayoutFrame boxLayoutFrame = new BoxLayoutFrame();
10       boxLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       boxLayoutFrame.setSize( 400, 220 ); // configura o tamanho do frame
12       boxLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe BoxLayoutDemo
```

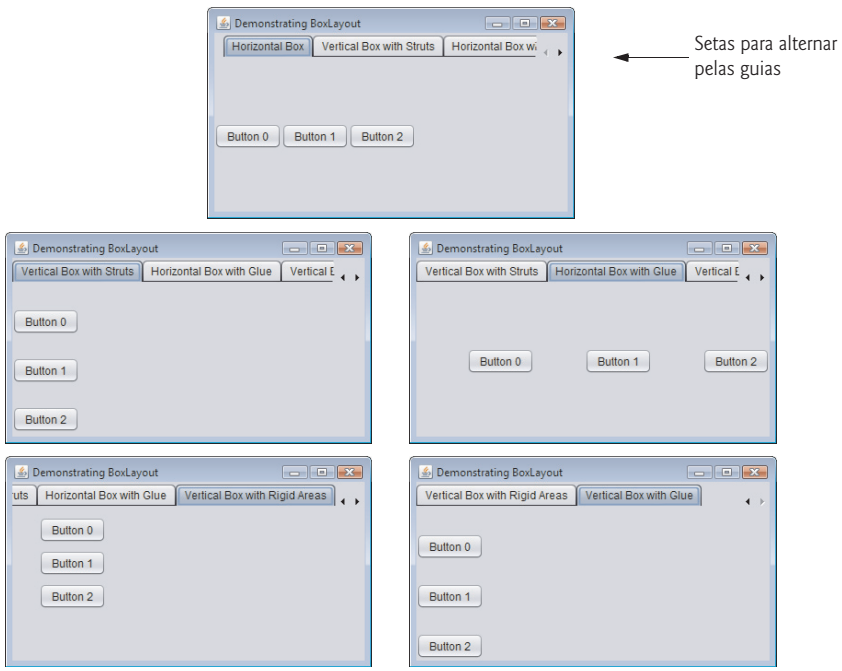


Figura 25.17 | Classe de teste para `BoxLayoutFrame`.

As linhas 19–22 criam contêineres `Box`. As referências `horizontal1` e `horizontal2` são inicializadas com método `static Box createHorizontalBox`, que retorna um contêiner `Box` com um `BoxLayout` horizontal em que os componentes GUI são organizados da esquerda para a direita. As variáveis `vertical1` e `vertical2` são inicializadas com o método `static Box` **`createVerticalBox`**, que retorna referências aos contêineres `Box` com um `BoxLayout` vertical, no qual os componentes GUI são organizados de cima para baixo.

O loop nas linhas 27–28 adiciona três `JButtons` a `horizontal1`. A estrutura `for` nas linhas 31–35 adiciona três `JButtons` ao `vertical1`. Antes de adicionar cada botão, a linha 33 adiciona um **suporte vertical** ao contêiner com o método `static Box` **`createVerticalStrut`**. Uma estrutura vertical é um componente GUI invisível que tem uma altura fixa em pixels e é utilizado para garantir uma quantidade fixa de espaço entre os componentes GUI. O argumento `int` para o método `createVerticalStrut` determina a altura da estrutura em pixels. Quando o contêiner é redimensionado, a distância entre os componentes GUI separados por struts não muda. A classe `Box` também declara o método **`createHorizontalStrut`** para `BoxLayouts` horizontais.

A estrutura `for` nas linhas 38–42 adiciona três `JButtons` ao `horizontal2`. Antes de adicionar cada botão, a linha 40 adiciona a **cola horizontal** ao contêiner com o método `static Box` **`createHorizontalGlue`**. A cola horizontal é um componente GUI invisível que pode ser utilizado entre componentes GUI de tamanho fixo para ocupar espaço adicional. Normalmente, o espaço extra aparece à direita do último componente GUI horizontal ou abaixo do último vertical em um `BoxLayout`. A cola permite que o espaço extra seja colocado entre componentes GUI. Quando o contêiner é redimensionado, componentes separados por componentes de cola permanecem no mesmo tamanho, mas a cola se expande ou se contrai para ocupar o espaço entre eles. A classe `Box` também declara o método **`createVerticalGlue`** para `BoxLayouts` verticais.

A estrutura `for` nas linhas 45–49 adiciona três `JButtons` ao `vertical2`. Antes de cada botão ser adicionado, a linha 47 adiciona uma **área rígida** ao contêiner com o método `static Box` **`createRigidArea`**. Uma área rígida é um componente GUI invisível que sempre tem a largura e altura fixas em pixels. O argumento para o método `createRigidArea` é um objeto `Dimension` que especifica a largura e altura da área.

As linhas 52–53 criam um objeto `JPanel` e configuram seu layout como um `BoxLayout` da maneira convencional, utilizando o método `Container` `setLayout`. O construtor `BoxLayout` recebe uma referência ao contêiner cujo layout ele controla, e uma constante indicando se o layout é horizontal (**`BoxLayout.X_AXIS`**) ou vertical (**`BoxLayout.Y_AXIS`**).

A estrutura `for` nas linhas 55–59 adiciona três `JButtons` ao `panel`. Antes de adicionar cada botão, a linha 57 adiciona um componente de cola ao contêiner com o método `static` `Box` **`createGlue`**. Esse componente expande ou contrai com base no tamanho da classe `Box`.

As linhas 62–63 criam um `JTabbedPane` para exibir os cinco contêineres nesse programa. O argumento **`JTabbedPane.TOP`** enviado ao construtor indica que as guias devem aparecer na parte superior do `JTabbedPane`. O argumento **`JTabbedPane.SCROLL_TAB_LAYOUT`** especifica que as guias devem ser posicionadas em uma nova linha se houver muitas para que se ajustem em uma linha.

Os contêineres `Box` e o `JPanel` são anexados ao `JTabbedPane` nas linhas 66–70. Tente executar o aplicativo. Quando a janela aparecer, redimensione a janela para ver como os componentes cola, estrutura e área rígida afetam o layout em cada guia.

Gerenciador de layout `GridBagLayout`

Um dos gerenciadores de layout predefinidos mais poderosos é **`GridBagLayout`** (no pacote `java.awt`). Esse layout é semelhante ao `GridLayout` pelo fato de que ele organiza os componentes em uma grade. Entretanto, `GridBagLayout` é mais flexível. Os componentes podem variar em tamanho (isto é, eles podem ocupar múltiplas linhas e colunas) e podem ser adicionados em qualquer ordem.

O primeiro passo na utilização de `GridBagLayout` é determinar a aparência da GUI. Para este passo você precisa somente de um pedaço de papel. Desenhe a GUI e, então, desenhe uma grade sobre ela, dividindo os componentes nas linhas e colunas. Os números iniciais de linhas e colunas devem ser 0, de modo que o gerenciador de layout `GridBagLayout` possa utilizar os números de linha e coluna para posicionar adequadamente os componentes na grade. A Figura 25.18 demonstra como desenhar linhas e colunas sobre uma GUI.

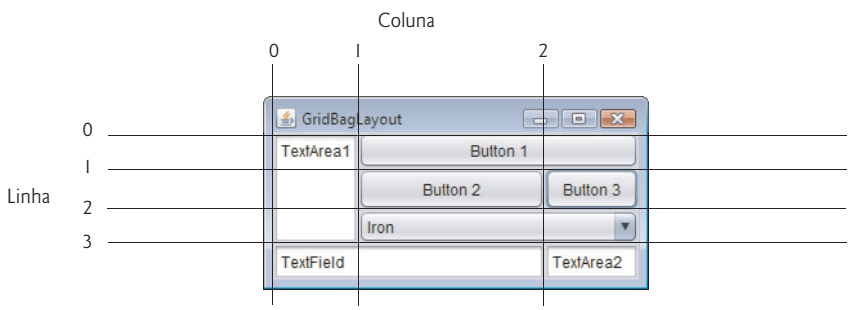


Figura 25.18 | Projetando uma GUI que utilizará `GridBagLayout`.

Um objeto **`GridBagConstraints`** descreve como um componente é posicionado em um `GridBagLayout`. Vários campos `GridBagConstraints` estão resumidos na Figura 25.19.

Campo	Descrição
`anchor`	Especifica a posição relativa (`NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST`, `CENTER`) do componente em uma área que ele não preenche.
`fill`	Redimensiona o componente na direção especificada (`NONE`, `HORIZONTAL`, `VERTICAL`, `BOTH`) quando a área de exibição for maior que o componente.
`gridx`	A coluna em que o componente será colocado.
`gridy`	A linha em que o componente será colocado.
`gridwidth`	O número de colunas que componente ocupa.
`gridheight`	O número de linhas que o componente ocupa.
`weightx`	A quantidade de espaço extra a alocar horizontalmente. O componente na grade pode tornar-se mais largo se houver espaço extra disponível.
`weighty`	A quantidade de espaço extra a alocar verticalmente. O componente na grade pode tornar-se mais alto se houver espaço extra disponível.

Figura 25.19 | Campos `GridBagConstraints`.

O campo `GridBagConstraints` **anchor** especifica a posição relativa do componente em uma área que ele não preenche. Atribui-se à variável `anchor` uma das seguintes constantes `GridBagConstraints`: **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST** ou **CENTER**. O valor padrão é `CENTER`.

O campo `GridBagConstraints` `fill` define como o componente aumenta se a área em que pode ser exibido for maior que o componente. Atribui-se à variável `fill` uma das seguintes constantes `GridBagConstraints`: **NONE**, **VERTICAL**, **HORIZONTAL** ou **BOTH**. O valor padrão é `NONE`, o que indica que o componente não crescerá em nenhuma direção. `VERTICAL` indica que ele crescerá verticalmente. `HORIZONTAL` indica que ele crescerá horizontalmente. `BOTH` indica que ele crescerá em ambas as direções.

As variáveis **gridx** e **gridy** especificam onde o canto superior esquerdo do componente é posicionado na grade. A variável `gridx` corresponde à coluna, e a variável `gridy` corresponde à linha. Na Figura 25.18, o `JComboBox` (exibindo `"Iron"`) tem um valor `gridx` de 1 e um valor `gridy` de 2.

A variável **gridwidth** especifica o número de colunas que um componente ocupa. O `JComboBox` ocupa duas colunas. A variável **gridheight** especifica o número de linhas que um componente ocupa. O `JTextArea` no lado esquerdo da Figura 25.18 ocupa três linhas.

A variável **weightx** especifica como distribuir o espaço horizontal extra para componentes na grade em um `GridBagLayout` quando o contêiner é redimensionado. Um valor zero indica que o componente na grade não aumenta horizontalmente por conta própria. Entretanto, se o componente distribui uma coluna contendo um componente com valor `weightx` diferente de zero, o componente com o valor `weightx` de zero crescerá horizontalmente na mesma proporção que o(s) outro(s) componente(s) dessa coluna. Isso ocorre porque cada componente deve ser mantido na mesma linha e coluna em que foi originalmente posicionado.

A variável **weighty** especifica como distribuir espaço vertical extra para componentes na grade em um `GridBagLayout` quando o contêiner é redimensionado. Um valor zero indica que o componente na grade não aumenta verticalmente por conta própria. Entretanto, se o componente se distribui por uma linha contendo um componente com valor `weighty` diferente de zero, o componente com o valor `weighty` de zero cresce verticalmente na mesma proporção que o(s) outro(s) componente(s) na mesma linha.

Na Figura 25.18, os efeitos de `weighty` e `weightx` não podem ser vistos facilmente até que o contêiner seja redimensionado e o espaço adicional torne-se disponível. Os componentes com valores de peso maiores ocupam mais do espaço adicional que aqueles com valores de peso menores.

Os componentes devem receber valores de peso positivos diferentes de zero — caso contrário, eles "se amontoam" no meio do contêiner. A Figura 25.20 mostra a GUI da Figura 25.18 com todos os pesos configurados como zero.

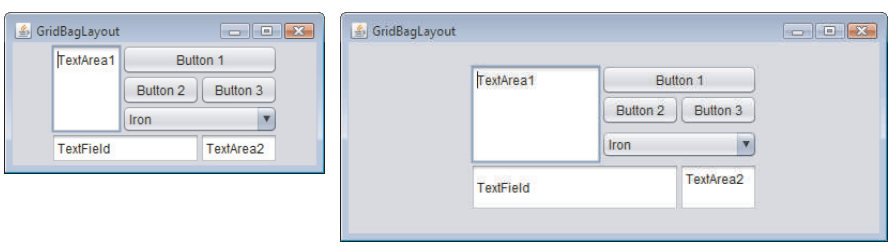


Figura 25.20 | `GridBagLayout` com os pesos configurados como zero.

O aplicativo nas figuras 25.21–25.22 utiliza o gerenciador de layout `GridBagLayout` para organizar os componentes da GUI na Figura 25.18. O aplicativo não faz nada, exceto demonstrar como utilizar `GridBagLayout`.

```
1  // Figura 25.21: GridbagFrame.java
2  // Demonstrando GridBagLayout
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JTextField;
9  import javax.swing.JButton;
10 import javax.swing.JComboBox;
11
12 public class GridBagFrame extends JFrame
13 {
14    private GridBagLayout layout; // layout desse quadro
15    private GridBagConstraints constraints; // restrições desse layout
16
17    // configura a GUI
```

```
18      public GridBagFrame()
19      {
20         super( "GridBagLayout" );
21         layout = new GridBagLayout();
22         setLayout( layout ); // configura o layout de frame
23         constraints = new GridBagConstraints(); // instancia restrições
24
25         // cria componentes GUI
26         JTextArea textArea1 = new JTextArea( "TextArea1", 5, 10 );
27         JTextArea textArea2 = new JTextArea( "TextArea2", 2, 2 );
28
29         String[] names = { "Iron", "Steel", "Brass" };
30         JComboBox comboBox = new JComboBox( names );
31
32         JTextField textField = new JTextField( "TextField" );
33         JButton button1 = new JButton( "Button 1" );
34         JButton button2 = new JButton( "Button 2" );
35         JButton button3 = new JButton( "Button 3" );
36
37         // weightx e weighty para textArea1 são 0: o padrão
38         // anchor para todos os componentes CENTER: o padrão
39         constraints.fill = GridBagConstraints.BOTH;
40         addComponent( textArea1, 0, 0, 1, 3 );
41
42         // weightx e weighty para button1 são 0: o padrão
43         constraints.fill = GridBagConstraints.HORIZONTAL;
44         addComponent( button1, 0, 1, 2, 1 );
45
46         // weightx e weighty para comboBox são 0: o padrão
47         // fill é HORIZONTAL
48         addComponent( comboBox, 2, 1, 2, 1 );
49
50         // button2
51         constraints.weightx = 1000;  // pode crescer na largura
52         constraints.weighty = 1;     // pode crescer na altura
53         constraints.fill = GridBagConstraints.BOTH;
54         addComponent( button2, 1, 1, 1, 1 );
55
56         // preenchimento é BOTH para button3
57         constraints.weightx = 0;
58         constraints.weighty = 0;
59         addComponent( button3, 1, 2, 1, 1 );
60
61         // weightx e weighty para textField são 0, preenchimento é BOTH
62         addComponent( textField, 3, 0, 2, 1 );
63
64         // weightx e weighty para textArea2 são 0, preenchimento é BOTH
65         addComponent( textArea2, 3, 2, 1, 1 );
66      } // fim do construtor GridBagFrame
67
68      // método para configurar restrições em
69      private void addComponent( Component component,
70         int row, int column, int width, int height )
71      {
72         constraints.gridx = column; // configura gridx
73         constraints.gridy = row; // configura gridy
74         constraints.gridwidth = width; // configura gridwidth
75         constraints.gridheight = height; // configure gridheight
76         layout.setConstraints( component, constraints ); // configura constraints
77         add( component ); // adiciona componente
78      } // fim do método addComponent
79   } // fim da classe GridBagFrame
```

Figura 25.21 | Gerenciador de layout `GridBagLayout`.

```
1  // Figura 25.22: GridBagDemo.java
2  // Demonstrando GridBagLayout
3  import javax.swing.JFrame;
4
5  public class GridBagDemo
6  {
7     public static void main( String[] args )
8     {
9        GridBagFrame gridBagFrame = new GridBagFrame();
10       gridBagFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       gridBagFrame.setSize( 300, 150 ); // configura o tamanho do frame
12       gridBagFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe GridBagDemo
```

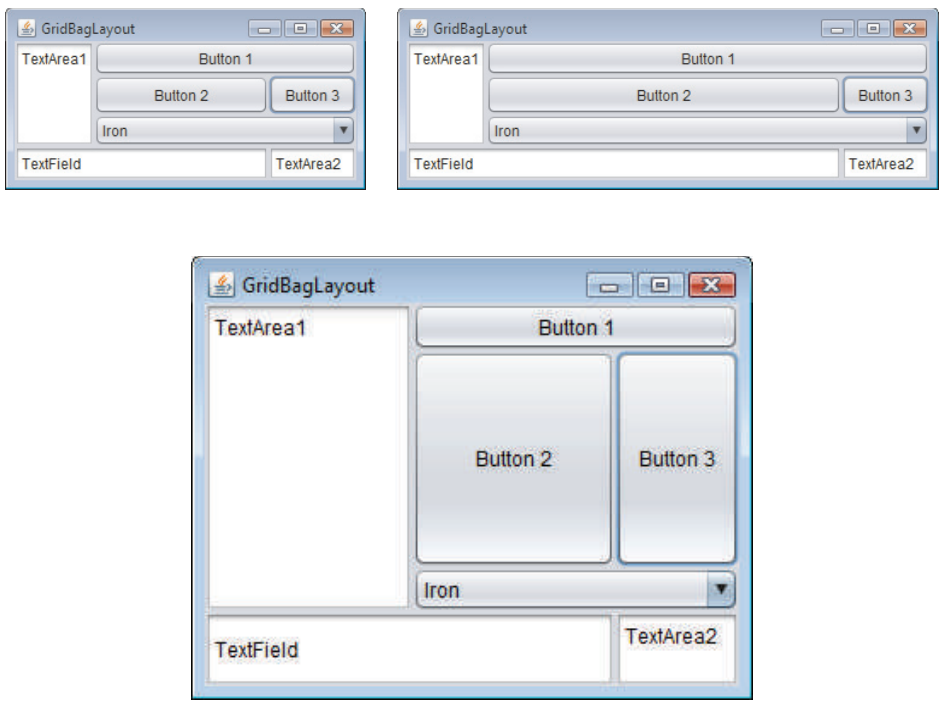


Figura 25.22 | Classe de teste para `GridBagFrame`.

A GUI contém três `JButtons`, duas `JTextAreas`, um `JComboBox` e um `JText-Field`. O gerenciador de layout é `GridBagLayout`. As linhas 21–22 criam o objeto `GridBagLayout` e configuram o gerenciador de layout para o `JFrame` como `layout`. A linha 23 cria o objeto `GridBagConstraints` utilizado para determinar a localização e tamanho de cada componente na grade. As linhas 26–35 criam cada componente GUI que será adicionado ao painel de conteúdo.

As linhas 39–40 configuram `JTextArea textArea1` e o adicionam ao painel de conteúdo. Os valores para `weightx` e `weighty` não são especificados em `constraints`, portanto cada um tem o valor zero por padrão. Portanto, a `JTextArea` não redimensionará a si própria mesmo se espaço estiver disponível. Entretanto, ela se distribui por múltiplas linhas, então o tamanho vertical está sujeito aos valores `weighty` dos `JButtons button2` e `button3`. Quando um botão é redimensionado verticalmente com base no seu valor `weighty`, a `JTextArea` também é redimensionada.

A linha 39 configura a variável `fill` em `constraints` como `GridBagConstraints.BOTH`, fazendo com que a `JTextArea` sempre preencha toda sua área alocada na grade. Um valor `anchor` não é especificado em `constraints`, assim o padrão `CENTER` é utilizado. Não utilizamos a variável `anchor` nesse aplicativo, portanto todos os componentes utilizarão o padrão. A linha 40 chama nosso método utilitário `addComponent` (declarado nas linhas 69–78). O objeto `JTextArea`, a linha, a coluna, o número de colunas a distribuir e o número de linhas a distribuir são passados como argumentos.

`JButton button1` é o próximo componente adicionado (linhas 43–44). Por padrão, os valores de `weightx` e `weighty` ainda são zero. A variável `fill` é configurada como `HORIZONTAL` — o componente sempre ocupará sua área na direção horizontal. A direção vertical não é ocupada. Como o valor `weighty` é zero, o botão se tornará mais alto somente se outro componente na mesma linha tiver um valor `weighty` diferente de zero. `JButton button1` está localizado na linha 0, coluna 1. Uma linha e duas colunas são ocupadas.

`JComboBox comboBox` é o próximo componente adicionado (linha 48). Por padrão, os valores `weightx` e `weighty` são zero, e a variável `fill` é configurada como `HORIZONTAL`. O botão `JComboBox` crescerá somente na direção horizontal. Observe que as variáveis `weightx`, `weighty` e `fill` retêm os valores configurados em `constraints` até eles serem alterados. O botão `JComboBox` é colocado na linha 2, coluna 1. Uma linha e duas colunas são ocupadas.

`JButton button2` é o próximo componente adicionado (linhas 51–54). É dado um valor de `weightx` de `1000` e um valor de `weighty` de `1`. A área ocupada pelo botão é capaz de crescer nas direções horizontal e vertical. A variável `fill` é configurada como `BOTH`, que especifica que o botão sempre ocupará a área inteira. Quando a janela é redimensionada, `button2` crescerá. O botão é colocado na linha 1, coluna 1. Uma linha e uma coluna são ocupadas.

`JButton button3` é adicionado a seguir (linhas 57–59). O valor `weightx` e o valor `weighty` são ambos configurados como zero e o valor de `fill` é `BOTH`. `JButton button3` crescerá se a janela for redimensionada; ele é afetado pelos valores de peso de `button2`. Observe que o valor de `weightx` para `button2` é muito maior do que para `button3`. Quando o redimensionamento ocorrer, `button2` ocupará uma porcentagem maior do novo espaço. O botão é colocado na linha 1, coluna 2. Uma linha e uma coluna são ocupadas.

Tanto o `JTextField textField` (linha 62) como a `JTextArea textArea2` (linha 65) têm um valor `weightx` de 0 e um valor `weighty` de 0. O valor de `fill` é `BOTH`. O `JTextField` é posicionado na linha 3, coluna 0 e a `JTextArea` na linha 3, coluna 2. O `JTextField` ocupa uma linha e duas colunas, a `JTextArea`, uma linha e uma coluna.

Os parâmetros do método `addComponent` são um `component` de referência `Component` e números inteiros `row`, `column`, `width` e `height`. As linhas 72–73 configuram as variáveis `GridBagConstraints gridx` e `gridy`. A variável `gridx` é atribuída à coluna em que o `Component` será posicionado e o valor `gridy` é atribuído à linha em que `Component` será posicionado. As linhas 74–75 configuram as variáveis `GridBagConstraints gridwidth` e `gridheight`. A variável `gridwidth` especifica o número de colunas que o `Component` estenderá na grade e a variável `gridheight` especifica o número de linhas que o `Component` estenderá na grade. A linha 76 configura as `GridBagConstraints` para um componente no `GridBagLayout`. O método **`setConstraints`** da classe `GridBagLayout` aceita um argumento `Component` e um argumento `GridBagConstraints`. A linha 77 adiciona o componente ao `JFrame`.

Quando executar esse aplicativo, tente redimensionar a janela para ver como as limitações para cada componente GUI afetam sua posição e tamanho na janela.

Constantes `GridBagConstraints` RELATIVE e REMAINDER

Em vez de `gridx` e `gridy`, uma variação de `GridBagLayout` utiliza as constantes `GridBagConstraints` **`RELATIVE`** e **`REMAINDER`**. `RELATIVE` especifica que o penúltimo componente em uma linha particular deve ser posicionado à direita do componente anterior na linha. `REMAINDER` especifica que um componente é o último componente em uma coluna. Qualquer componente que não seja penúltimo ou o último componente em uma linha deve especificar os valores para as variáveis `GridbagConstraints` `gridwidth` e `gridheight`. O aplicativo nas figuras 25.23–25.24 organiza os componentes no `GridBagLayout`, utilizando essas constantes.

```
1  // Figura 25.23: GridBagFrame2.java
2  // Demonstrando as constantes GridBagLayout.
3  import java.awt.GridBagLayout;
4  import java.awt.GridBagConstraints;
5  import java.awt.Component;
6  import javax.swing.JFrame;
7  import javax.swing.JComboBox;
8  import javax.swing.JTextField;
9  import javax.swing.JList;
10 import javax.swing.JButton;
11
12 public class GridBagFrame2 extends JFrame
13 {
14    private GridBagLayout layout; // layout desse quadro
15    private GridBagConstraints constraints; // restrições desse layout
16
17    // configura a GUI
18    public GridBagFrame2()
19    {
20       super( "GridBagLayout" );
21       layout = new GridBagLayout();
22       setLayout( layout ); // configura o layout de frame
23       constraints = new GridBagConstraints(); // instancia restrições
24
25       // cria componentes GUI
26       String[] metals = { "Copper", "Aluminum", "Silver" };
```

```
27          JComboBox comboBox = new JComboBox( metals );
28
29          JTextField textField = new JTextField( "TextField" );
30
31          String[] fonts = { "Serif", "Monospaced" };
32          JList list = new JList( fonts );
33
34          String[] names = { "zero", "one", "two", "three", "four" };
35          JButton[] buttons = new JButton[ names.length ];
36
37          for ( int count = 0; count < buttons.length; count++ )
38             buttons[ count ] = new JButton( names[ count ] );
39
40          // define restrições dos componentes GUI para textField
41          constraints.weightx = 1;
42          constraints.weighty = 1;
43          constraints.fill = GridBagConstraints.BOTH;
44          constraints.gridwidth = GridBagConstraints.REMAINDER;
45          addComponent( textField );
46
47          // buttons[0] -- weightx e weighty são 1: fill é BOTH
48          constraints.gridwidth = 1;
49          addComponent( buttons[ 0 ] );
50
51          // buttons[1] -- weightx e weighty são 1: fill é BOTH
52          constraints.gridwidth = GridBagConstraints.RELATIVE;
53          addComponent( buttons[ 1 ] );
54
55          // buttons[2] -- weightx e weighty são 1: fill é BOTH
56          constraints.gridwidth = GridBagConstraints.REMAINDER;
57          addComponent( buttons[ 2 ] );
58
59          // comboBox -- weightx é 1: fill é BOTH
60          constraints.weighty = 0;
61          constraints.gridwidth = GridBagConstraints.REMAINDER;
62          addComponent( comboBox );
63
64          // buttons[3] -- weightx é 1: fill é BOTH
65          constraints.weighty = 1;
66          constraints.gridwidth = GridBagConstraints.REMAINDER;
67          addComponent( buttons[ 3 ] );
68
69          // buttons[4] -- weightx e weighty são 1: fill é BOTH
70          constraints.gridwidth = GridBagConstraints.RELATIVE;
71          addComponent( buttons[ 4 ] );
72
73          // list -- weightx e weighty são 1: fill é BOTH
74          constraints.gridwidth = GridBagConstraints.REMAINDER;
75          addComponent( list );
76       } // fim do construtor GridBagFrame2
77
78       // adiciona um componente ao contêiner
79       private void addComponent( Component component )
80       {
81          layout.setConstraints( component, constraints );
82          add( component ); // adiciona componente
83       } // fim do método addComponent
84    } // fim da classe GridBagFrame2
```

Figura 25.23 | Constantes `GridBagConstraints` `RELATIVE` e `REMAINDER`.

```
1  // Figura 25.24: GridBagDemo2.java
2  // Demonstrando as constantes GridBagLayout.
3  import javax.swing.JFrame;
4
5  public class GridBagDemo2
6  {
```

```
7       public static void main( String[] args )
8       {
9          GridBagFrame2 gridBagFrame = new GridBagFrame2();
10         gridBagFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11         gridBagFrame.setSize( 300, 200 ); // configura o tamanho do frame
12         gridBagFrame.setVisible( true ); // exibe o frame
13      } // fim de main
14   } // fim da classe GridBagDemo2
```

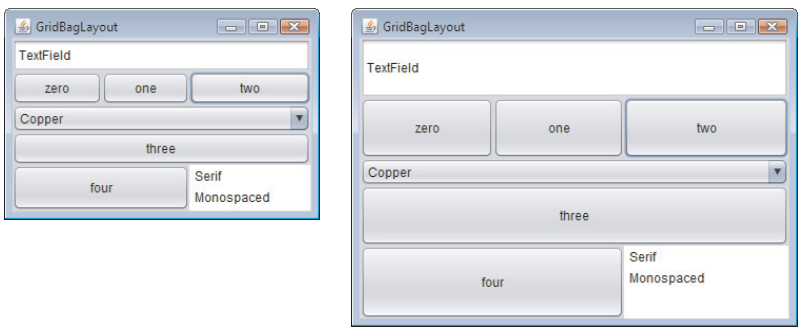


Figura 25.24 | Classe de teste para `GridBagDemo2`.

As linhas 21–22 criam um `GridBagLayout` e o utilizam para configurar o gerenciador de layout do `JFrame`. Os componentes que são inseridos no `GridBagLayout` são criados nas linhas 27–38 — eles são um `JComboBox`, um `JTextField`, uma `JList` e cinco `JButtons`.

O `JTextField` é adicionado primeiro (linhas 41–45). Os valores `weightx` e `weighty` são configurados como 1. A variável `fill` é configurada como `BOTH`. A linha 44 especifica que o `JTextField` é o último componente na linha. O `JTextField` é adicionado ao painel de conteúdo com uma chamada ao nosso método utilitário `addComponent` (declarado nas linhas 79–83). O método `addComponent` aceita um argumento `Component` e utiliza o método `GridBagLayout` `setConstraints` para configurar as limitações para o `Component`. O método `add` anexa o componente ao painel de conteúdo.

`JButton buttons[ 0 ]` (linhas 48–49) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. Como `buttons[ 0 ]` não é um dos dois últimos componentes na linha, ele recebeu um `gridwidth` de 1 e assim ocupará uma coluna. O `JButton` é adicionado ao painel de conteúdo com uma chamada para o método utilitário `addComponent`.

`JButton buttons[ 1 ]` (linhas 52–53) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. A linha 52 especifica que o `JButton` deve ser posicionado em relação ao componente anterior. O `Button` é adicionado ao `JFrame` com uma chamada a `addComponent`.

`JButton buttons[ 2 ]` (linhas 56–57) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. O `JButton` é o último componente da linha, então `REMAINDER` é utilizado. O `JButton` é adicionado ao painel de conteúdo com uma chamada a `addComponent`.

O `JComboBox` (linhas 60–62) tem um `weightx` de 1 e um `weighty` de 0. O `JComboBox` não crescerá verticalmente. O `JComboBox` é o único componente na linha, então `REMAINDER` é utilizado. O `JComboBox` é adicionado ao painel de conteúdo com uma chamada a `addComponent`.

`JButton buttons[ 3 ]` (linhas 65–67) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. O `JButton` é o único componente na linha, então `REMAINDER` é utilizado. O `JButton` é adicionado ao painel de conteúdo com uma chamada a `addComponent`.

`JButton buttons[ 4 ]` (linhas 70–71) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. Esse `JButton` é o único componente na linha, então `RELATIVE` é utilizado. O `JButton` é adicionado ao painel de conteúdo com uma chamada a `addComponent`.

A `JList` (linhas 74–75) tem valores `weightx` e `weighty` de 1. A variável `fill` é `BOTH`. O `JList` é adicionado ao painel de conteúdo com uma chamada a `addComponent`.

25.10 Conclusão

Este capítulo completa nossa introdução às GUIs. Nele, discutimos tópicos sobre GUI adicionais, como menus, controles deslizantes, menus pop-up, interfaces de múltiplos documentos, painéis com guias e aparência e funcionamento plugável do Java. Todos esses componentes podem ser adicionados a aplicativos existentes para torná-los mais fáceis de utilizar e entender. Também apresentamos gerenciadores de layout adicionais para organizar e dimensionar componentes GUI. No próximo capítulo, você aprenderá sobre o multithreading, uma capacidade poderosa que permite aos aplicativos utilizar threads para realizar múltiplas tarefas de uma vez.

Resumo

Seção 25.2 JSlider

- Os `JSliders` permitem ao usuário selecionar a partir de um intervalo de valores inteiros. `JSliders` podem exibir marcas de medida principais, marcas de medida secundárias menores e rótulos para as marcas de medida. Eles também suportam aderência às marcas, na qual posicionar o marcador entre duas marcas de medida faz o marcador aderir à marca de medida mais próxima.
- Se um `JSlider` tiver o foco, as teclas de seta esquerdas e direitas fazem com que marcador do `JSlider`- diminuir ou aumentar por 1. As teclas de seta que apontam para cima e para baixo também fazem com que o marcador do `JSlider` diminua ou aumente por 1, respectivamente. A *tecla PgDn* (page down) e a *tecla PgUp* (page up) fazem com que o marcador diminua ou aumente por incrementos de bloco de um décimo do intervalo de valores, respectivamente. A *tecla Home* move o marcador para o valor mínimo e a *tecla End* move-o para o valor máximo.
- Os `JSliders` têm orientação horizontal ou vertical. Para um `JSlider` horizontal, o valor mínimo está na extrema esquerda e o valor máximo, na extrema direita. Para um `JSlider` vertical, o valor mínimo está na parte inferior extrema e o valor máximo, na parte superior extrema. A posição da caixa de rolagem indica o valor atual do `JSlider`. O método `getValue` da classe `JSlider` retorna a posição atual da caixa de rolagem.
- O método `JSlider` `setMajorTickSpacing` configura o espaçamento para marcas de tique em um `JSlider`. O método `setPaintTicks` com um argumento `true` indica que as marcas de medida devem ser exibidas.
- Os `JSliders` geram `ChangeEvents` quando o usuário interage com um `JSlider`. Um `ChangeListener` declara o método `stateChanged` que pode responder a `ChangeEvents`.

Seção 25.3 Windows: notas adicionais

- Cada janela gera eventos de janela quando o usuário a manipula. A interface `WindowListener` fornece sete métodos de tratamento de evento de janela — `windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified` e `windowOpened`.

Seção 25.4 Utilizando menus com frames

- Menus são uma parte integral das GUIs que permitem aos usuários realizar ações sem atravancar desnecessariamente uma GUI com componentes extras. Na GUI do Swing, os menus só podem ser anexados a objetos das classes com o método `setJMenuBar` (por exemplo, `JFrame` e `JApplet`).
- As classes utilizadas para construir menus são `JMenuBar`, `JMenuItem`, `JMenu`, `JCheckBoxMenuItem` e `JRadioButtonMenuItem`.
- Um `JMenuBar` é um contêiner para menus. Um `JMenuItem` aparece em um menu que, quando selecionado, faz uma ação ser executada. Um `JMenu` contém itens de menu e pode ser adicionado a um `JMenuBar` ou a outros `JMenus` como submenus.
- Quando um menu é clicado, ele se expande para mostrar sua lista de itens de menu. O método `JMenu` `addSeparator` adiciona uma linha separadora a um menu.
- Quando um `JCheckBoxMenuItem` é selecionado, uma marca de verificação aparece à esquerda do item de menu. Quando o `JCheckBoxMenuItem` é selecionado novamente, a marca é removida.
- Quando múltiplos `JRadioButtonMenuItems` são mantidos como parte de um `ButtonGroup`, somente um pode ser selecionado em um dado momento. Quando um deles é selecionado, um círculo preenchido aparece à sua esquerda. Quando outro `JRadioButtonMenuItem` é selecionado, o círculo preenchido à esquerda do item previamente selecionado é removido.
- O método `AbstractButton` `setMnemonic` especifica o mnemônico para um `AbstractButton`. Os caracteres mnemônicos normalmente são exibidos com um caractere de sublinhado.
- Uma caixa de diálogo modal não permite acesso a qualquer outra janela no aplicativo até que o diálogo seja fechado. Os diálogos exibidos com a classe `JOptionPane` são diálogos modais. A classe `JDialog` pode ser utilizada para criar seus próprios diálogos modais ou não modais.

Seção 25.5 JPopupMenu

- Menus pop-up sensíveis ao contexto são criados com a classe `JPopupMenu`. O evento de acionamento do pop-up normalmente ocorre quando o usuário pressiona e libera o botão direito do mouse. O método `MouseEvent` `isPopupTrigger` retorna `true` se o evento de acionamento do pop-up ocorreu.
- O método `JPopupMenu` `show` exibe um `JPopupMenu`. O primeiro argumento especifica o componente de origem, que ajuda a determinar onde o `JPopupMenu` aparecerá. Os dois últimos argumentos são as coordenadas no canto superior esquerdo do componente de origem, em que o `JPopupMenu` aparece.

Seção 25.6 Aparência e comportamento plugável

- A classe `UIManager.LookAndFeelInfo` mantém informações sobre uma aparência e um funcionamento.
- O método `UIManager` `static` `getInstalledLookAndFeels` retorna um array de objetos `UIManager.LookAndFeelInfo` que descreve as aparências e funcionamentos disponíveis.
- O método `static` `setLookAndFeel` da `UIManager` altera a aparência e o comportamento. O método `static` `SwingUtilities` `updateComponentTreeUI` altera a aparência e o comportamento de cada componente anexado ao seu argumento `Component` à nova aparência e comportamento.

Seção 25.7 `JDesktopPane` e `JInternalFrame`

- Muitos aplicativos atuais utilizam uma interface de múltiplos documentos (*Multiple Document Interface* — MDI) para gerenciar vários documentos abertos que são processados em paralelo. As classes `JDesktopPane` e `JInternalFrame` do Swing fornecem suporte para criar interfaces de múltiplos documentos.

Seção 25.8 `JTabbedPane`

- Um `JTabbedPane` organiza componentes GUI em camadas, das quais somente uma é visível de cada vez. Usuários acessam cada camada clicando na guia.

Seção 25.9 Gerenciadores de layout: `BoxLayout` e `GridBagLayout`

- `BoxLayout` organiza os componentes GUI da esquerda para a direita ou de cima para baixo em um contêiner.
- A classe `Box` representa um contêiner com `BoxLayout` como seu gerenciador padrão de layout e fornece métodos `static` para criar um `Box` com um `BoxLayout` horizontal ou vertical.
- `GridBagLayout` é semelhante a `GridLayout`, mas o tamanho de cada componente pode variar e os componentes podem ser adicionados em qualquer ordem.
- Um objeto `GridBagConstraints` especifica como um componente é inserido em um `GridBagLayout`. O método `GridBagLayout` `setConstraints` recebe um argumento `Component` e um argumento `GridBagConstraints` e configura as restrições do `Component`.

Terminologia

`add`, método da classe `JMenu`, 777
`add`, método da classe `JMenuBar`, 778
`addSeparator`, método da classe `JMenu`, 778
`addTab`, método da classe `JTabbedPane`, 789
`addWindowListener`, método da classe `Window`, 773
aderência às marcas para um `JSlider`, 769
`anchor`, campo da classe `GridBagConstraints`, 793
aparência e funcionamento plugável (*Pluggable Look-And-Feel* — PLAF), 769
área rígida de classe `Box`, 791
barra de menus, 773
barra de título de uma janela, 772
bloquear o incremento de um `JSlider`, 769
`BOTH`, constante da classe `GridBagConstraints`, 793
caixa de diálogo modal, 778
`CENTER`, constante da classe `GridBagConstraints`, 793
`ChangeEvent`, classe, 772
`ChangeListener`, interface, 772
cola horizontal, 791
componente de origem, 781
`createGlue`, método da classe `Box`, 792
`createHorizontalGlue`, método da classe `Box`, 791
`createHorizontalStrut`, método da classe `Box`, 791
`createRigidArea`, método da classe `Box`, 791
`createVerticalBox`, método da classe `Box`, 791
`createVerticalGlue`, método da classe `Box`, 791
`createVerticalStrut`, método da classe `Box`, 791
`dispose`, método da classe `Window`, 772
`EAST`, constante da classe `GridBagConstraints`, 793
estrutura vertical, 791
evento de gatilho pop-up, 779
evento de janela, 773
`getClassName`, método da classe `UIManager LookAndFeelInfo`, 784
`getInstalledLookAndFeels`, método da classe `UIManager`, 784
`getValue`, método da classe `JSlider`, 772
`GridBagConstraints`, classe, 792
`gridheight`, campo da classe `GridBagConstraints`, 793
`gridwidth`, campo da classe `GridBagConstraints`, 793
`gridx`, campo da classe `GridBagConstraints`, 793
`gridy`, campo da classe `GridBagConstraints`, 793
`HORIZONTAL`, constante da classe `GridBagConstraints`, 793
indicador de classe `JSlider`, 769
interface de múltiplos documentos (*Multiple Document Interface* — MDI), 784
`isPopupTrigger`, método da classe `MouseEvent`, 781
`isSelected`, método da classe `AbstractButton`, 777
item de menu, 773
janela, 772
janela, eventos, 773
janela-filha em uma interface de múltiplos documentos, 784
janela-pai em uma interface de múltiplos documentos, 784
janela-pai para uma caixa de diálogo, 777
`JDesktopPane`, classe, 784
`JDialog`, classe, 778
`JMenu`, classe, 773
`JMenuItem`, classe, 773
`JPopupMenu`, classe, 779
`JSlider`, classe, 769
`JTabbedPane`, classe, 787
`LookAndFeelInfo`, classe aninhada da classe `UIManager`, 784
marcas de medida em um `JSlider`, 769
margem de uma janela, 772
MDI (Multiple Document Interface), 784
menu pop-up sensível ao contexto, 779
metal, aparência, 781
miniatura de um `JSlider`, 769
mnemônico, 773
Motif-style (UNIX), aparência e funcionamento, 769
`NONE`, constante da classe `GridBagConstraints`, 793
`NORTH`, constante da classe `GridBagConstraints`, 793
`NORTHEAST`, constante da classe `GridBagConstraints`, 793
`NORTHWEST`, constante da classe `GridBagConstraints`, 793
`pack`, método da classe `Window`, 787
`RELATIVE`, constante da classe `GridBagConstraints`, 796
`REMAINDER`, constante da classe `GridBagConstraints`, 796
`SCROLL_TAB_LAYOUT`, constante da classe `JTabbedPane`, 792
`setConstraints`, método da classe `GridBagLayout`, 796
`setDefaultCloseOperation`, método da classe `JFrame`, 772
`setForeground`, método da classe `JComponent`, 778
`setInverted`, método da classe `JSlider`, 770
`setJMenuBar`, método da classe `JFrame`, 778
`setLocation`, método da classe `Component`, 772
`setLookAndFeel`, método da classe `UIManager`, 784
`setMajorTickSpacing`, método da classe `JSlider`, 772
`setMnemonic`, método da classe `AbstractButton`, 777
`setPaintTicks`, método da classe `JSlider`, 772
`setSelected`, método da classe `AbstractButton`, 778

show, método da classe JPopupMenu, 781
SOUTH, constante da classe GridBagConstraints, 793
SOUTHEAST, constante da classe GridBagConstraints, 793
SOUTHWEST, constante da classe GridBagConstraints, 793
stateChanged, método da interface ChangeListener, 772
submenu, 773
SwingUtilities, classe, 784
TOP, constante da classe JTabbedPane, 792
UIManager, classe, 784
updateComponentTreeUI, método da classe SwingUtilities, 784
VERTICAL, constante da classe GridBagConstraints, 793
weightx, campo da classe GridBagConstraints, 793
weighty, campo da classe GridBagConstraints, 793
WEST, constante da classe GridBagConstraints, 793
windowActivated, método da interface WindowListener, 773
windowClosed, método da interface WindowListener, 773
windowClosing, método da interface WindowListener, 773
WindowConstants, interface, 772
windowDeactivated, método da interface WindowListener, 773
windowDeiconified, método da interface WindowListener, 773
windowIconified, método da interface WindowListener, 773
WindowListener, interface, 773
windowOpened, método da interface WindowListener, 773
X_AXIS, constante da classe Box, 792
Y_AXIS, constante da classe Box, 792

Exercícios de autorrevisão

25.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) A classe _____ é utilizada para criar um objeto de menu.

b) O método _____ da `JMenu` classe coloca uma barra separadora em um menu.

c) Os eventos `JSlider` são tratados pelo método _____ da interface _____.

d) A variável de instância `GridBagConstraints` _____ é configurada como `CENTER` por padrão.

25.2 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Quando o programador cria um `JFrame`, no mínimo um menu deve ser criado e adicionado ao `JFrame`.

b) A variável `fill` pertence à classe `GridBagLayout`.

c) O desenho em um componente GUI é realizado com relação à coordenada (0, 0) do canto superior esquerdo do componente.

d) O layout padrão para um `Box` é `BoxLayout`.

25.3 Encontre o(s) erro(s) em cada um dos seguintes itens e explique como corrigir o(s) erro(s).

a)
```
JMenubar b;
```

b)
```
mySlider = JSlider( 1000, 222, 100, 450 );
```

c)
```
gbc.fill = GridBagConstraints.NORTHWEST;  // configura preenchimento
```

d)
```
// sobrescreve para pintar sobre um componente Swing personalizado
public void paintcomponent( Graphics g )
{
   g.drawString( "HELLO", 50, 50 );
} // fim do método paintComponent
```

e)
```
// cria e exibe um JFrame
JFrame f = new JFrame( "A Window" );
f.setVisible( true );
```

Respostas dos exercícios de autorrevisão

25.1 a) `JMenu`. b) `addSeparator`. c) `stateChanged`, `ChangeListener`. d) `anchor`.

25.2 a) Falsa. Um `JFrame` não requer nenhum menu.

b) Falsa. A variável `fill` pertence à classe `GridBagConstraints`.

c) Verdadeira.

d) Verdadeira.

25.3 a) `JMenubar` deve ser `JMenuBar`.

b) O primeiro argumento para o construtor deve ser um `SwingConstants.HORIZONTAL` ou `SwingConstants.VERTICAL` e a palavra-chave `new` deve ser utilizada depois do operador `=`. Além disso, o valor mínimo deve ser menor que o máximo e o valor inicial deve estar no intervalo.

c) A constante deve ser `BOTH`, `HORIZONTAL`, `VERTICAL` ou `NONE`.

d) `paintcomponent` deve ser `paintComponent` e o método deve chamar o `super.paintComponent( g )` como sua primeira instrução.

e) O método `setSize` do `JFrame` também deve ser chamado para estabelecer o tamanho da janela.

Exercícios

25.4 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um `JMenuItem` que é um `JMenu` é chamado _____.

b) O método _____ anexa um `JMenuBar` a um `JFrame`.

c) A classe contêiner _____ tem um `BoxLayout` padrão.

d) Um(a) _____ gerencia um conjunto de janelas-filhas declarado com a classe `JInternalFrame`.

25.5 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Os menus requerem um objeto `JMenuBar` então podem ser anexados a um `JFrame`.

b) `BoxLayout` é o gerenciador padrão de layout para um `JFrame`.

c) `JApplet`s podem conter menus.

25.6 Localize o(s) erro(s) em cada um dos seguintes itens. Explique como corrigir o erro (s).

a)
```
x.add( new JMenuItem( "Submenu Color" ) ); // cria submenu
```

b)
```
container.setLayout( new GridbagLayout() );
```

25.7 Escreva um programa que exiba um círculo de tamanho aleatório e calcule e exiba a área, raio, diâmetro e circunferência. Utilize as seguintes equações: *diâmetro* = 2 × *raio*, *área* = π × *raio*^{2}, *circunferência* = 2 × π × *raio*. Utilize a constante `Math.PI` para pi (π). Todos os desenhos devem ser feitos em uma subclasse de `JPanel` e os resultados dos cálculos devem ser exibidos em um `JtextArea` de leitura.

25.8 Aprimore o programa no Exercício 25.7 permitindo ao usuário alterar o raio com um `JSlider`. O programa deve funcionar em todos os raios no intervalo de 100 a 200. À medida que o raio muda, o diâmetro, a área e a circunferência devem ser atualizados e exibidos. O raio inicial deve ser 150. Utilize as equações do Exercício 25.7. Todos os desenhos devem ser feitos em uma subclasse de `JPanel` e os resultados dos cálculos devem ser exibidos em um `JtextArea` de leitura.

25.9 Explore os efeitos da variação dos valores `weightx` e `weighty` do programa na Figura 25.21. O que acontece quando um slot tem um peso não zero, mas não tem permissão de preencher a área inteira (isto é, o valor `fill` não é `BOTH`)?

25.10 Escreva um programa que utilize o método `paintComponent` para desenhar o valor atual de um `JSlider` em uma subclasse de `JPanel`. Além disso, forneça um `JTextField` em que um valor específico possa ser inserido. O `JTextField` deve exibir o valor atual do `JSlider` todas as vezes. Alterar o valor no `JTextField` também deve atualizar o `JSlider`. Um `JLabel` deve ser utilizado para identificar o `JTextField`. Os métodos `JSlider` `setValue` e `getValue` devem ser utilizados. [*Nota*: O método `setValue` é um método `public` que não retorna um valor e aceita um argumento do tipo inteiro, o valor `JSlider`, que determina a posição da caixa de rolagem.]

25.11 Declare uma subclasse de `JPanel` chamado `MyColorChooser` que forneça três objetos `JSlider` e três objetos `JTextField`. Cada `JSlider` representa os valores de 0 a 255 para as partes de azul, verde e vermelha de uma cor. Utilize esses valores como os argumentos para o construtor `Color` a fim de criar um novo objeto `Color`. Exiba o valor atual de cada `JSlider` no correspondente `JTextField`. Quando o usuário altera o valor do `JSlider`, o `JTextField` deve ser alterado correspondentemente. Utilize seu novo componente GUI como parte de um aplicativo que exiba o valor `Color` atual desenhando um retângulo preenchido.

25.12 Modifique a classe `MyColorChooser` do Exercício 25.11 para permitir ao usuário digitar um valor inteiro em um `JTextField` para configurar o valor de vermelho, verde ou azul. Quando o usuário pressionar *Enter* no `JTextField`, o `JSlider` correspondente deve ser configurado com o valor apropriado.

25.13 Modifique o aplicativo no Exercício 25.12 para desenhar a cor atual como um retângulo em uma instância de uma subclasse do `JPanel`, que fornece seu próprio método `paintComponent` para desenhar o retângulo e fornece os métodos *set* para configurar os valores de vermelho, verde e azul para a cor atual. Quando um método *set* é invocado, o painel de desenho deve automaticamente repintar `(repaint)` a si próprio.

25.14 Modifique o aplicativo no Exercício 25.13 para permitir que usuário arraste o mouse pelo painel de desenho (uma subclasse do `JPanel`) para desenhar uma forma na cor atual. Permita ao usuário escolher que forma desenhar.

25.15 Modifique o aplicativo no Exercício 25.14 para fornecer ao usuário a capacidade de terminar o aplicativo clicando na caixa de fechamento na janela que é exibida e selecionando `Exit` em um menu `File`. Utilize as técnicas mostradas na Figura 25.5.

25.16 (***Aplicativo de desenho completo***) Utilizando as técnicas desenvolvidas neste capítulo e no Capítulo 14, crie um aplicativo de desenho completo. O programa deve utilizar os componentes GUI dos capítulos 14 e 25 para permitir que o usuário selecione as características de forma, cor e preenchimento. Cada forma deve ser armazenada em um array de objetos `MyShape`, onde `MyShape` é a superclasse na sua hierarquia das classes de forma. Utilize um `JDesktopPane` e `JInternalFrame`s para permitir ao usuário criar múltiplos desenhos separados em janelas-filhas separadas. Crie a interface com o usuário como uma janela-filha separada contendo todo o componente GUI que permite ao usuário determinar as características da forma que será desenhada. O usuário então pode clicar em qualquer `JInternalFrame` para desenhar a forma.

A mais geral definição da beleza… Multiplicidade na Unidade.
— **Samuel Taylor Coleridge**

Não bloqueie o modo de pesquisa.
— **Charles Sanders Peirce**

Uma pessoa com um relógio sabe que horas são; uma pessoa com dois relógios nunca está segura.
— **Provérbio**

Aprenda a trabalhar e a esperar.
— **Henry Wadsworth Longfellow**

O mundo está mudando tão rápido hoje que um homem que diz que algo não pode ser feito, em geral, é interrompido por alguém fazendo esse algo.
— **Elbert Hubbard**

26

Multithreading

Objetivos

Neste capítulo, você aprenderá:

- O que são as threads e por que elas são úteis.
- Como as threads permitem gerenciar atividades concorrentes.
- O ciclo de vida de uma thread.
- As prioridades e agendamento de threads.
- A criar e executar `Runnable`s.
- A sincronização de threads.
- O que são relacionamentos produtor/consumidor e como são implementados com multithreading.
- A permitir que múltiplas threads atualizem componentes GUI Swing de maneira segura para threads.
- Sobre as interfaces `Callable` e `Future`, que podem ser utilizadas com threading para executar tarefas que retornam resultados.

Sumário

26.1 Introdução
26.2 Estados de thread: ciclo de vida de uma thread
26.3 Prioridades de thread e agendamento de thread
26.4 Criando e executando threads
- 26.4.1 `Runnables` e a classe `Thread`
- 26.4.2 Gerenciamento de threads com o framework `Executor`

26.5 Sincronização de thread
- 26.5.1 Compartilhamento de dados não sincronizados
- 26.5.2 Compartilhamento de dados sincronizados — tornando operações atômicas

26.6 Relacionamento entre produtor e consumidor sem sincronização
26.7 Relacionamento de produtor/consumidor: `ArrayBlockingQueue`
26.8 Relacionamento entre produtor e consumidor com sincronização
26.9 Relacionamento de produtor/consumidor: buffers limitados
26.10 Relacionamento de produtor/consumidor: as interfaces `Lock` e `Condition`
26.11 Multithreading com GUI
- 26.11.1 Realizando cálculos em uma thread worker
- 26.11.2 Processando resultados intermediários com `SwingWorker`

26.12 Interfaces `Callable` e `Future`
26.13 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas aos exercícios de autorrevisão | Exercícios |

26.1 Introdução

Seria interessante se pudéssemos fazer uma coisa por vez, e fazê-la bem, mas em geral isso é difícil. O corpo humano realiza uma grande variedade de operações **paralelamente** — ou, como diremos por todo este capítulo, **concorrentemente**. A respiração, a circulação sanguínea, a digestão, o pensamento e a locomoção, por exemplo, podem ocorrer simultaneamente. Todos os sentidos — visão, tato, olftato, paladar e audição — podem ser empregados ao mesmo tempo. Os computadores, também, realizam operações concorrentemente. É comum aos computadores pessoais compilar um programa, enviar um arquivo para uma impressora e receber mensagens de correio eletrônico em uma rede concorrentemente. Apenas os computadores que têm múltiplos processadores podem, de fato, executar múltiplas instruções concorrentemente. Os sistemas operacionais em computadores de um único processador criam a ilusão da execução concorrente alternando rapidamente entre atividades, mas nesses computadores apenas uma instrução pode executar de cada vez.

A maioria das linguagens de programação não permite especificar atividades concorrentes. Especialmente, as linguagens fornecem instruções de controle sequenciais que permitem especificar que apenas uma ação deve ser realizada por vez, com a execução avançando para a ação seguinte depois que a anterior tiver sido concluída. Historicamente, a concorrência foi implementada com os primitivos de sistemas operacionais disponíveis apenas para programadores de sistemas experientes.

A linguagem de programação Ada, desenvolvida pelo Departamento de Defesa dos Estados Unidos, tornou primitivos de concorrência amplamente disponíveis para as empresas contratadas do Departamento de Defesa que estavam construindo sistemas de comando e controle militar. Entretanto, a tecnologia Ada não foi amplamente utilizada nas universidades e na indústria.

Concorrência no Java

O Java disponibiliza a concorrência por meio da linguagem e das APIs. Você especifica que um aplicativo contém **threads**, ou **linhas de execução** separadas, nas quais cada uma tem sua própria pilha de chamadas de método e seu próprio contador de programa, permitindo a execução simultânea com outras threads ao compartilhar recursos no nível do aplicativo como a memória. Essa capacidade, chamada **multithreading**, *não* está disponível nas linguagens C e C++ básicas, mas já existem bibliotecas que fornecem essa funcionalidade.

Dica de desempenho 26.1

Um problema com aplicativos de uma única thread que pode levar a uma fraca responsividade é que as atividades longas e demoradas devem ser concluídas antes de as outras poderem iniciar. Em um aplicativo com múltiplas threads, as threads podem ser distribuídas por múltiplos processadores (se disponíveis) de modo que múltiplas tarefas sejam mesmo executadas concorrentemente e o aplicativo possa operar de modo mais eficiente. O multithreading também pode aumentar o desempenho em sistemas de um único processador que simulam a concorrência — quando uma thread não puder avançar (porque, por exemplo, está esperando o resultado de uma operação E/S), outra pode utilizar o processador.

Ao contrário das linguagens que não têm capacidades de multithreading integradas e, portanto, devem fazer chamadas não portáveis para primitivos de multithreading do sistema operacional, o Java inclui primitivos de multithreading como parte da própria linguagem e de suas bibliotecas. Isso facilita a manipulação de threads de maneira portável entre plataformas.

Usos da programação concorrente

Discutiremos muitas aplicações da **programação de processos concorrentes**. Por exemplo, ao fazer download de um arquivo grande (como uma imagem, um clipe de áudio ou um videoclipe) na internet, o usuário pode não querer esperar até o download do clipe inteiro para iniciar a reprodução. Para resolver esse problema, podemos colocar múltiplas threads para trabalhar — uma delas faz o download do clipe e a outra o reproduz. Essas atividades prosseguem concorrentemente. Para evitar a reprodução instável, iremos **sincronizar** as threads de modo que a thread do player não inicie até que haja uma quantidade suficiente do clipe na memória para manter a thread do player ocupado.

A Java Virtual Machine (JVM) também utiliza threads. Ela cria threads para executar programas e threads para realizar tarefas de limpeza, como a coleta de lixo.

A programação concorrente é difícil

Escrever programas de múltiplas threads pode ser difícil. Embora a mente humana possa realizar funções simultaneamente, as pessoas acham difícil alternar entre linhas de pensamento paralelas. Para ver por que programas de múltiplas threads podem ser difíceis de escrever e de entender, tente a seguinte experiência: abra três livros na página 1 e tente ler os livros concorrentemente. Leia algumas palavras do primeiro livro e depois do segundo, do terceiro e, então, faça um loop e leia as próximas poucas palavras a seguir do primeiro livro e assim por diante. Depois dessa experiência, você apreciará os desafios da tecnologia multithreading — alternar entre os livros, ler brevemente, lembrar-se de onde parou em cada livro, aproximar ainda mais o livro que está lendo para poder vê-lo melhor e afastar os que não está lendo — e, no meio de todo esse caos, tentar compreender o conteúdo dos livros!

Utilize APIs de concorrência predefinida sempre que possível

É difícil e propenso a erros programar aplicativos concorrentes. Se precisar utilizar a sincronização em um programa, você deve seguir algumas diretrizes simples. *Utilize classes existentes da Java API* (como a classe `ArrayBlockingQueue` que discutimos na Seção 26.7) *que gerenciam a sincronização para você*. As classes da Java API são escritas por experts, foram inteiramente testadas e depuradas, operam com eficiência e ajudam a evitar interrupções e armadilhas comuns. Se precisar de capacidades ainda mais complexas, utilize as interfaces `Lock` e `Condition` que são introduzidas na Seção 26.10.

As interfaces `Lock` e `Condition` só devem ser utilizadas por programadores avançados que conhecem as armadilhas comuns da programação concorrente. Explicamos esses tópicos neste capítulo por várias razões — eles fornecem uma base sólida para entender como os aplicativos concorrentes sincronizam o acesso à memória compartilhada; é importante entender os conceitos, mesmo que um aplicativo não utilize essas ferramentas explicitamente; e mostrando a complexidade envolvida no uso desses recursos de baixo nível, esperamos enfatizar a importância do uso das capacidades de concorrência predefinidas sempre que possível.

26.2 Estados de thread: ciclo de vida de uma thread

A qualquer dado momento, diz-se que uma thread está em um dos vários **estados de thread** — ilustrados no diagrama de estado UML na Figura 26.1. Vários dos termos no diagrama são definidos nas seções posteriores.

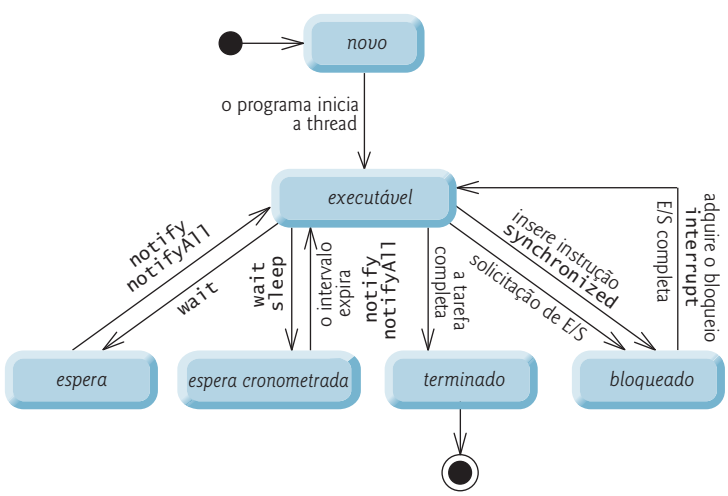


Figura 26.1 | Diagrama de estado de ciclo e vida da thread.

Estados novo *e* executável

Uma nova thread inicia seu ciclo de vida no estado ***novo***. Ela permanece nesse estado até que o programa inicia a thread, o que a coloca no estado ***executável***. Considera-se que uma thread no estado *executável* está executando sua tarefa.

Estado de espera

Às vezes a thread *executável* transita para o estado de ***espera*** enquanto espera outra thread realizar uma tarefa. Uma *thread de espera* transita de volta para o estado *executável* apenas quando outra thread a notifica para continuar executando.

Estado de espera sincronizada

Uma thread *executável* pode entrar no estado de ***espera sincronizada*** por um intervalo especificado de tempo. Ela transita para o estado *executável* quando esse intervalo de tempo expira ou quando o evento pelo qual ela está esperando ocorre. As threads de *espera sincronizada* não podem utilizar um processador, mesmo se houver um disponível. Uma thread *executável* pode transitar para o estado de *espera sincronizada* se fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa. Essa thread retorna ao estado *executável* quando é notificada por outra thread ou quando o intervalo sincronizado expira — o que ocorrer primeiro. Outra maneira de colocar uma thread no estado de *espera sincronizada* é colocar a thread *executável* para dormir. Uma **thread adormecida** permanece no estado *de espera sincronizada* por um período de tempo designado (chamado **intervalo de adormecimento**), depois do qual ela retorna ao estado *executável*. As threads dormem quando, por um breve período, não têm de realizar nenhuma tarefa. Por exemplo, um processador de texto pode conter uma thread que grave periodicamente backups (isto é, grava uma cópia) do documento atual no disco para fins de recuperação. Se a thread não dormisse entre os sucessivos backups, seria necessário um loop em que testaria continuamente se deve ou não gravar uma cópia do documento em disco. Esse loop consumiria tempo de processador sem realizar trabalho produtivo, reduzindo assim o desempenho de sistema. Nesse caso, é mais eficiente para a thread especificar um intervalo de adormecimento (igual ao período entre backups sucessivos) e entrar no estado de *espera sincronizada*. Essa thread é retornada ao estado *executável* quando seu intervalo de adormecimento expira, ponto em que ela grava uma cópia do documento no disco e entra novamente no estado de *espera sincronizada*.

Estado bloqueado

Uma thread executável transita para o estado ***bloqueado*** quando tenta realizar uma tarefa que não pode ser completada imediatamente e deve esperar temporariamente até que essa tarefa seja concluída. Por exemplo, quando uma thread emite uma solicitação de entrada/saída, o sistema operacional bloqueia a thread de executar até que essa solicitação de E/S se complete — nesse ponto, a thread *bloqueada* transita para o estado *executável* e, desse modo, pode retomar a execução. Uma thread *bloqueada* não pode utilizar um processador, mesmo se algum estiver disponível.

Estado terminado

Uma thread *executável* entra no estado ***terminado*** (às vezes chamado estado ***morto***) quando completa sua tarefa com sucesso ou, de outro modo, a termina (talvez por causa de um erro). No diagrama de estado UML da Figura 26.1, o estado *terminado* é seguido pelo estado final da UML (símbolo do alvo) para indicar o fim das transições de estado.

Visão do sistema operacional do estado executável

No nível do sistema operacional, o estado *executável* do Java geralmente inclui dois estados separados (Figura 26.2). O sistema operacional oculta esses estados da Java Virtual Machine (JVM), que vê apenas o estado *executável*. Quando uma thread entra pela primeira vez no estado *executável* a partir do estado *novo*, ela está no estado ***pronto***. Uma thread *pronta* entra no estado de ***execução*** (isto é, começa a executar) quando o sistema operacional a atribui a um processador — também conhecido como **despachar a thread**. Na maioria dos sistemas operacionais, cada thread recebe uma pequena quantidade de tempo de processador — chamada de **quantum** ou **fração de tempo** — com a qual realiza sua tarefa. Decidir o tamanho que o quantum deve ter é um tópico-chave em cursos de sistemas operacionais. Quando seu quantum expira, a thread retorna ao estado *pronto*, e o sistema operacional atribui

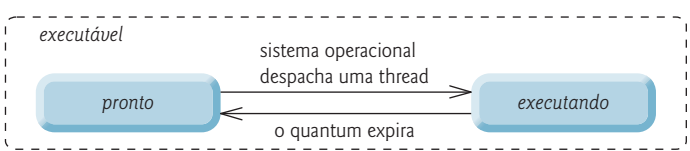


Figura 26.2 | Visualização interna do sistema operacional do estado *executável* do Java.

outra thread ao processador (ver a Seção 26.3). As transições entre os estado *pronto* e *executável* são tratadas isoladamente pelo sistema operacional. A JVM não "vê" as transições — ela simplesmente visualiza a thread como *executável* e a deixa para o sistema operacional fazer a transição da thread entre os estados *pronto* e *executável*. O processo que utiliza um sistema operacional para determinar qual thread despachar é conhecido como **agendamento de thread** e depende das prioridades de thread (discutidas na próxima seção).

26.3 Prioridades de thread e agendamento de thread

Toda thread do Java tem uma **prioridade de thread** que ajuda a determinar a ordem em que são agendadas. As prioridades do Java variam entre `MIN_PRIORITY` (uma constante de 1) e `MAX_PRIORITY` (uma constante de 10). Por padrão, toda thread recebe a prioridade `NORM_PRIORITY` (uma constante de 5). Cada nova thread herda a prioridade da thread que o cria. Informalmente, as threads de prioridade mais alta são mais importantes para um programa e devem ser alocadas em tempo de processador antes das threads de prioridade mais baixa. Entretanto, as prioridades de thread não podem garantir a ordem em que elas são executadas.

[*Nota*: As constantes `MAX_PRIORITY`, `MIN_PRIORITY` e `NORM_PRIORITY` são declaradas na classe **`Thread`**. Recomenda-se não criar e utilizar explicitamente objetos `Thread` para implementar a concorrência, mas, em vez disso, utilizar a interface `Executor` (que é descrita na Seção 26.4.2). A classe `Thread` contém alguns métodos `static` úteis, que discutimos mais adiante no capítulo.]

A maioria dos sistemas operacionais suporta o fracionamento de tempo, o que permite que threads de igual prioridade compartilhem um processador. Sem o fracionamento de tempo, cada thread em um conjunto de threads de igual prioridade executa até sua conclusão (a menos que ela deixe o estado *executável* e entre no estado de *espera* ou *espera sincronizada* ou, ainda, seja interrompida por uma thread de prioridade mais alta) antes de outras threads de igual prioridade terem uma chance de executar. Com o fracionamento de tempo, mesmo se a thread não tiver concluído a execução quando seu quantum expirar, o processador é tirado da thread e recebe a próxima thread de igual prioridade, se houver alguma disponível.

O **agendador de threads** (*thread scheduler*) de um sistema operacional determina que thread executa em seguida. Uma simples implementação do scheduler de thread mantém a thread de prioridade mais alta *executando* o tempo todo e, se houver mais de uma thread de prioridade mais alta, isso assegura que cada uma dessas threads executa por um quantum no estilo **rodízio**. A Figura 26.3 ilustra uma

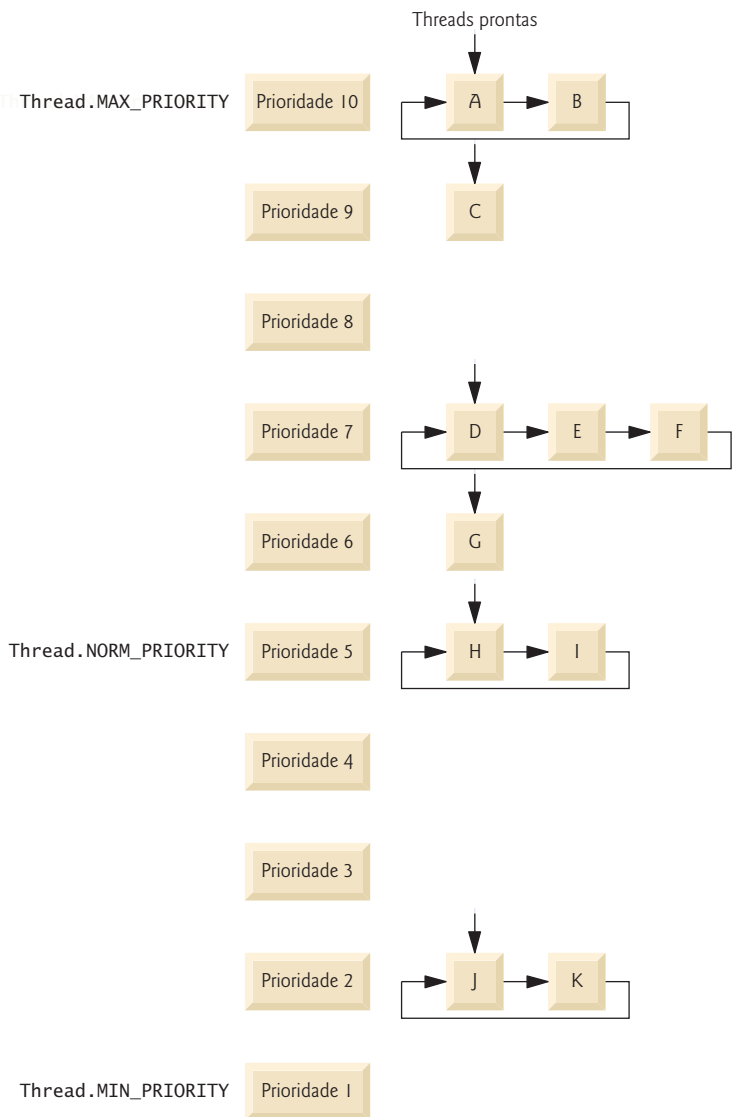


Figura 26.3 | Agendamento de prioridade de threads.

fila de prioridades de múltiplos níveis para as threads. Na figura, supondo um computador de um único processador, as threads A e B executam por um quantum no esquema de rodízio até que ambas as threads completem a execução. Isso significa que A obtém um quantum de tempo para executar. Então B obtém um quantum. Então A obtém outro quantum. Então B obtém outro quantum. Isso continua até que uma thread complete. O processador então dedica toda sua energia à thread que resta (a menos que outra thread de prioridade 10 torne-se pronta). Em seguida, a thread C executa até sua conclusão (assumindo que nenhuma thread de prioridade mais alta ou igual chegará). Cada uma das threads D, E e F executa um quantum no esquema rodízio até que todas completem a execução (novamente supondo que nenhuma thread de prioridade mais alta ou igual chegará). Esse processo continua até que todos os threads executem até sua conclusão.

Quando uma thread de prioridade mais alta entra no estado *pronto*, o sistema operacional geralmente faz preempção da thread atualmente *em execução* (uma operação conhecida como **agendamento preemptivo**). Dependendo do sistema operacional, as threads de prioridade mais alta poderiam adiar — possivelmente por um tempo indefinido — a execução de threads de prioridade mais baixa. Esse **adiamento indefinido** é às vezes referido, mais alegoricamente, como **inanição**.

O Java fornece utilitários de concorrência de nível superior para ocultar um pouco dessa complexidade e tornar os programas de múltiplas threads menos propensos a erros. As prioridades de thread são utilizadas nos bastidores para interagir com o sistema operacional, mas a maioria dos programadores que utiliza o multithreading do Java não se preocupará com a configuração e o ajuste de prioridades de thread.

Dica de portabilidade 26.1

O agendamento de threads é dependente de plataforma — o comportamento de um programa de múltiplas threads pode variar nas diferentes implementações do Java.

Dica de portabilidade 26.2

Ao projetar programas de múltiplas threads, considere as capacidades de threading de todas as plataformas nas quais os programas executarão. Utilizar prioridades diferentes das prioridades padrão tornará a comportamento dos seus programas dependentes da plataforma. Se seu objetivo for a portabilidade, não ajuste as prioridades de thread.

26.4 Criando e executando threads

A forma preferida de criar aplicativos Java de múltiplas threads é implementando a interface **`Runnable`** (do pacote `java.lang`). Um objeto `Runnable` representa uma "tarefa" que pode executar concorrentemente com outras tarefas. A interface `Runnable` declara o método **`run`** único, que contém o código que define a tarefa que um objeto `Runnable` deve realizar. Quando uma thread executando um `Runnable` é criada e iniciada, ela chama o método `run` do objeto `Runnable`, que executa na nova thread.

26.4.1 Runnables e a classe Thread

A classe `PrintTask` (Figura 26.4) implementa `Runnable` (linha 5), de modo que múltiplos `PrintTasks` possam executar concorrentemente. A variável `sleepTime` (linha 7) armazena um valor do tipo inteiro aleatório de 0 a 5 segundos criado no construtor `PrintTask` (linha 16). Cada thread executando uma `PrintTask` adormece pelo período de tempo especificado por `sleepTime` e, então, gera saída do nome de sua tarefa e de uma mensagem indicando que isso foi feito no estado adormecido.

```
1  // Figura 26.4: PrintTask.java
2  // Classe PrintTask dorme por um tempo aleatório de 0 a 5 segundos
3  import java.util.Random;
4
5  public class PrintTask implements Runnable
6  {
7     private final int sleepTime; // tempo de adormecimento aleatório para a thread
8     private final String taskName; // nome de tarefa
9     private final static Random generator = new Random();
10
11    public PrintTask( String name )
12    {
13       taskName = name; // configura o nome da tarefa
14
15       // seleciona tempo de adormecimento aleatório entre 0 e 5 segundos
16       sleepTime = generator.nextInt( 5000 ); // milissegundos
17    } // fim do construtor PrintTask
18
19    // método run contém o código que uma thread executará
20    public void run()
21    {
```

```
22          try // coloca a thread para dormir pelo período de tempo sleepTime
23          {
24             System.out.printf( "%s going to sleep for %d milliseconds.\n",
25                taskName, sleepTime );
26             Thread.sleep( sleepTime ); // coloca a thread para dormir
27          } // fim do try
28          catch ( InterruptedException exception )
29          {
30             System.out.printf( "%s %s\n", taskName,
31                "terminated prematurely due to interruption" );
32          } // fim do catch
33
34          // imprime o nome da tarefa
35          System.out.printf( "%s done sleeping\n", taskName );
36       } // fim do método run
37    } // fim da classe PrintTask
```

Figura 26.4 | A classe `PrintTask` dorme por um tempo aleatório de 0 a 5 segundos.

Uma `PrintTask` executa quando uma thread chama o método `run` da `PrintTask`. As linhas 24–25 exibem uma mensagem que indica o nome da tarefa em execução atualmente e que a tarefa vai dormir por `sleepTime` milissegundos. A linha 26 invoca o método `static` **`sleep`** da classe `Thread` para colocar a thread no estado *espera sincronizada* pelo período de tempo especificado. Nesse ponto, a thread perde o processador e o sistema permite que outra thread execute. Quando a thread acordar, ela entra novamente no estado *executável*. Quando a `PrintTask` for novamente atribuída a um processador, a linha 35 gera saída de uma mensagem que indica que a tarefa não está mais dormindo, depois o método `run` termina. Observe que `catch` nas linhas 28–32 é necessário porque o método `sleep` poderia lançar uma exceção verificada do tipo **`InterruptedException`** se o método **`interrupt`** da thread adormecida fosse chamado.

A Figura 26.5 cria objetos `Thread` para executar `PrintTasks`. As linhas 12–14 criam três threads, cada uma especificando uma nova `PrintTask` a executar. As linhas 19–21 chamam o método `Thread` `start` em cada uma das threads, colocando cada uma delas no estado *executável* e configurando uma chamada para o correspondente método `run` de `Runnable` para executar a tarefa. A linha 23 gera saída de uma mensagem indicando que todas as tarefas das threads foram iniciadas e que o método `main` terminou de executar.

```
1  // Figura 26.5: ThreadCreator.java
2  // Criando e iniciando três threads para executar Runnables.
3  import java.lang.Thread;
4
5  public class ThreadCreator
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "Creating threads" );
10
11       // cria cada thread com um novo runnable selecionado
12       Thread thread1 = new Thread( new PrintTask( “task1” ) );
13       Thread thread2 = new Thread( new PrintTask( “task2” ) );
14       Thread thread3 = new Thread( new PrintTask( “task3” ) );
15
16       System.out.println( "Threads created, starting tasks." );
17
18       // inicia threads e coloca no estado executável
19       thread1.start(); // invoca o método run de task1
20       thread2.start(); // invoca o método run de task2
21       thread3.start(); // invoca o método run de task3
22
23       System.out.println( "Tasks started, main ends.\n" );
24    } // fim de main
25 } // fim da classe ThreadCreator
```

```
Creating threads
Threads created, starting tasks
Tasks started, main ends

task3 going to sleep for 491 milliseconds
```

```
task2 going to sleep for 71 milliseconds
task1 going to sleep for 3549 milliseconds
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

```
Creating threads
Threads created, starting tasks
task1 going to sleep for 4666 milliseconds
task2 going to sleep for 48 milliseconds
task3 going to sleep for 3924 milliseconds
Tasks started, main ends

task2 done sleeping
task3 done sleeping
task1 done sleeping
```

Figura 26.5 | Criando e iniciando três threads para executar `Runnable`s.

O código em `main` executa na **thread principal**, uma thread criada pela JVM. O código no método `run` de `PrintTask` (linhas 20–36 da Figura 26.4) executa nas threads criadas nas linhas 12–14 da Figura 26.5. Quando o método `main` terminar, o próprio programa continuará executando porque ainda haverá threads que estão ativas (isto é, quaisquer das três threads que criamos que ainda não alcançaram o estado *terminado*). O programa não terminará até que sua última thread complete a execução.

As saídas de exemplo desse programa mostram o nome e tempo de adormecimento de cada tarefa quando a thread vai dormir. A thread com o tempo de adormecimento mais curto normalmente acorda primeiro, o que indica que ela não está mais dormindo e terminará. Na Seção 26.9, discutimos as questões de multithreading que poderiam impedir que a thread com o tempo mais curto de adormecimento acordasse primeiro. Na primeira saída, a thread principal termina antes de qualquer outra thread gerar saída de seus nomes e tempos de adormecimento. Isso mostra que a thread principal executa até a conclusão antes que qualquer uma das outras threads tenha uma chance de executar. Na segunda saída, todas as threads geram saída de seus nomes e períodos de adormecimento antes de a thread principal terminar. Isso mostra que outras threads executaram antes de a thread principal terminar. Esse é um exemplo do agendamento de rodízio discutido na Seção 26.3. Observe também que na primeira saída de exemplo, `task3` vai dormir primeiro e `task1` por último, embora iniciássemos a thread da `task1` primeiro. Isso ilustra o fato de que não podemos prever a ordem em que as threads serão agendadas, mesmo se soubermos a ordem em que foram criadas e iniciadas.

26.4.2 Gerenciamento de threads com o framework Executor

Embora seja possível criar threads explicitamente, como na Figura 26.5, recomenda-se utilizar a interface `Executor` para gerenciar a execução de objetos `Runnable` para você. Em geral, um objeto **`Executor`** cria e gerencia um grupo de threads chamado **pool de threads** para executar `Runnable`s. O uso de um `Executor` é muito mais vantajoso do que você mesmo criar threads. Os `Executors` podem reutilizar threads existentes para eliminar o overhead que cria uma nova thread para cada tarefa e podem aprimorar o desempenho otimizando o número de threads a fim de assegurar que o processador permaneça ocupado, sem criar tantas threads que esgotam os recursos do aplicativo.

A interface `Executor` declara um método único chamado **`execute`** que aceita um `Runnable` como um argumento. O `Executor` atribui cada `Runnable` passado para seu método `execute` a uma das threads disponíveis no pool de threads. Se não houver nenhuma thread disponível, o `Executor` cria uma nova thread ou espera uma thread tornar-se disponível e atribui a essa thread o `Runnable` que foi passado para o método `execute`.

A **interface `ExecutorService`** (do pacote `java.util.concurrent`) estende `Executor` e declara muitos outros métodos para gerenciar o ciclo de vida de um `Executor`. Um objeto que implementa a interface `ExecutorService` pode ser criado utilizando métodos `static` declarados na classe **`Executors`** (do pacote `java.util.concurrent`). Utilizamos a interface `ExecutorService` e um método da classe `Executors` no aplicativo a seguir, que executa três tarefas.

A Figura 26.6 utiliza um objeto `ExecutorService` para gerenciar threads que executam `PrintTasks`. O método `main` (linhas 8–29) cria e nomeia três objetos `PrintTask` (linhas 11–13). A linha 18 utiliza o método `Executors` **`newCachedThreadPool`** para obter um `ExecutorService` que cria novas threads conforme requerido pelo aplicativo. Essas threads são utilizadas por `threadExecutor` para executar os `Runnable`s.

```
1  // Figura 26.6: TaskExecutor.java
2  // Utilizando um ExecutorService para executar Runnables.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class TaskExecutor
7  {
8     public static void main( String[] args )
9     {
10       // cria e nomeia cada executável
11       PrintTask task1 = new PrintTask( "task1" );
12       PrintTask task2 = new PrintTask( "task2" );
13       PrintTask task3 = new PrintTask( "task3" );
14
15       System.out.println( "Starting Executor" );
16
17       // cria ExecutorService para gerenciar threads
18       ExecutorService threadExecutor = Executors.newCachedThreadPool();
19
20       // inicia threads e coloca no estado executável
21       threadExecutor.execute( task1 ); // inicia task1
22       threadExecutor.execute( task2 ); // inicia task2
23       threadExecutor.execute( task3 ); // inicia task3
24
25       // encerra threads trabalhadoras quando suas tarefas terminarem
26       threadExecutor.shutdown();
27
28       System.out.println( "Tasks started, main ends.\n" );
29    } // fim de main
30 } // fim da classe TaskExecutor
```

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
Starting Executor
task1 going to sleep for 1342 milliseconds
task2 going to sleep for 277 milliseconds
task3 going to sleep for 2737 milliseconds
Tasks started, main ends

task2 done sleeping
task1 done sleeping
task3 done sleeping
```

Figura 26.6 | Utilizando um `ExecutorService` para executar `Runnable`s.

As linhas 21–23 invocam o método `execute` do `ExecutorService`. Esse método executa o `Runnable` que lhe foi passado como um argumento (nesse caso um `PrintTask`) em algum momento no futuro. A tarefa especificada pode executar em uma das threads no pool de threads do `ExecutorService`, em uma nova thread criada para executá-la, ou na thread que chamou o método `execute`; o `ExecutorService` gerencia esses detalhes. O método `execute` retorna imediatamente de cada invocação — o programa não espera cada `PrintTask` terminar. A linha 26 chama o método `ExecutorService` **`shutdown`**, que notifica o `ExecutorService` para parar de aceitar novas tarefas, mas continua executando as tarefas que já foram enviadas. Uma vez que todos os `Runnable`s enviados anteriormente forem completados, `threadExecutor` termina. A linha 28 gera saída de uma mensagem que indica que as tarefas foram iniciadas e que a thread principal está terminando sua execução. As saídas de exemplo desse programa são semelhantes àquelas do programa anterior e demonstram novamente o não determinismo do agendamento de threads.

26.5 Sincronização de thread

Quando múltiplas threads compartilham um objeto e ele é modificado por uma ou várias delas, podem ocorrer resultados indeterminados (como veremos nos exemplos) a menos que o acesso ao objeto compartilhado seja gerenciado adequadamente. Se uma thread estiver no processo de atualização de um objeto compartilhado e outra thread também tentar atualizá-lo, não é claro a atualização de qual thread entra em vigor. Quando isso acontecer, não se pode confiar no comportamento do programa — às vezes, o programa produzirá resultados corretos e, outras vezes, não. Em qualquer caso, não haverá nenhuma indicação de que o objeto compartilhado foi manipulado incorretamente.

O problema pode ser resolvido fornecendo a somente uma thread, por vez, o código de *acesso exclusivo* que manipula o objeto compartilhado. Durante esse tempo, outras threads que desejarem manipular o objeto são mantidas na espera. Quando a thread com acesso exclusivo ao objeto terminar de manipulá-lo, uma das threads que foi mantida na espera tem a permissão de prosseguir. Esse processo, chamado **sincronização de threads**, coordena o acesso a dados compartilhados por múltiplas threads concorrentes. Sincronizando threads dessa maneira, você pode assegurar que cada thread que acessa um objeto compartilhado exclui todas as outras threads de fazer isso simultaneamente — isso é chamado **exclusão mútua**.

Monitores

Uma maneira comum de realizar a sincronização é utilizar os **monitores** predefinidos do Java. Todo objeto tem um monitor e um **bloqueio de monitor** (ou **bloqueio intrínseco**). O monitor assegura que o bloqueio de monitor do seu objeto é mantido por no máximo uma única thread em qualquer dado momento. Desse modo, monitores e bloqueios de monitor podem ser utilizados para forçar a exclusão mútua. Se uma operação exigir que a thread em execução mantenha um bloqueio enquanto a operação for realizada, uma thread deve adquirir o bloqueio antes de prosseguir com a operação. Outras threads tentando realizar uma operação que requer o mesmo bloqueio serão *bloqueadas* até que a primeira thread libere o bloqueio, ponto em que as threads *bloqueadas* podem tentar adquirir o bloqueio e prosseguir com a operação.

Para especificar que uma thread deve manter um bloqueio de monitor para executar um bloco do código, o código deve ser colocado em uma **instrução `synchronized`**. Diz-se que o código está **guardado** pelo bloqueio de monitor; uma thread deve **adquirir o bloqueio** para executar as instruções `synchronized`. O monitor permite que apenas uma thread por vez execute instruções dentro de blocos `synchronized` que bloqueiam o mesmo objeto, uma vez que somente uma thread por vez pode manter o bloqueio de monitor. As instruções `synchronized` são declaradas utilizando a **palavra-chave `synchronized`**:

```
synchronized ( objeto )
{
   instruções
} // fim da instrução synchronized
```

onde *objeto* é o objeto cujo bloqueio de monitor será adquirido; *objeto* é normalmente `this` se for o objeto no qual a instrução `synchronized` aparece. Se diversas instruções `synchronized` estiverem tentando executar em um objeto ao mesmo tempo, apenas uma delas poderá estar ativa no objeto por vez — todas as outras threads que tentarem entrar em uma instrução `synchronized` do mesmo objeto serão colocadas no estado *bloqueado*.

Quando uma instrução `synchronized` termina de executar, o bloqueio de monitor do objeto é lançado e uma das threads *bloqueadas* tentando inserir uma instrução `synchronized` pode ganhar permissão para adquirir o bloqueio para prosseguir. O Java também permite **métodos `synchronized`**. Antes de executar, um método `synchronized` não `static` deve adquirir o bloqueio no objeto que é utilizado para chamar o método. De modo semelhante, um método `synchronized static` deve adquirir o bloqueio na classe que é utilizada para chamar o método.

26.5.1 Compartilhamento de dados não sincronizados

Um exemplo aqui ilustrará os perigos de compartilhar um objeto entre threads sem sincronização apropriada. Nesse exemplo, dois `Runnable`s mantêm referências a um único array do tipo inteiro. Cada `Runnable` grava cinco valores ao array e, então, termina. Isso pode parecer inofensivo, mas pode resultar em erros se o array for manipulado sem sincronização.

Classe `SimpleArray`

Um objeto `SimpleArray` (Figura 26.7) será compartilhado por múltiplas threads. `SimpleArray` permitirá que essas threads coloquem valores `int` no `array` (declarados na linha 8). A linha 9 inicializa a variável `writeIndex`, que será utilizada para determinar o elemento de array que deve ser gravado em seguida. O construtor (linhas 13–16) cria um array de inteiros do tamanho desejado.

```
1  // Figura 26.7: SimpleArray.java
2  // A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads.
3  import java.util.Arrays;
4  import java.util.Random;
5
```

```
6  public class SimpleArray // ATENÇÃO: NÃO SEGURO PARA THREADS!
7  {
8     private final int[] array; // o array de inteiros compartilhado
9     private int writeIndex = 0; // o índice do próximo elemento a ser gravado
10    private final static Random generator = new Random();
11
12    // constrói um SimpleArray de um dado tamanho
13    public SimpleArray( int size )
14    {
15       array = new int[ size ];
16    } // fim do construtor
17
18    // adiciona um valor ao array compartilhado
19    public void add( int value )
20    {
21       int position = writeIndex; // armazena o índice de gravação
22
23       try
24       {
25          // coloca a thread para dormir por 0-499 milissegundos
26          Thread.sleep( generator.nextInt( 500 ) );
27       } // fim do try
28       catch ( InterruptedException ex )
29       {
30          ex.printStackTrace();
31       } // fim do catch
32
33       // coloca valor no elemento correto
34       array[ position ] = value;
35       System.out.printf( "%s wrote %2d to element %d.\n",
36          Thread.currentThread().getName(), value, position );
37
38       ++writeIndex; // incrementa índice do elemento a ser gravado depois
39       System.out.printf( "Next write index: %d\n", writeIndex );
40    } // fim do método add
41
42    // utilizado para gerar saída do conteúdo do array de inteiros compartilhado
43    public String toString()
44    {
45       return "\nContents of SimpleArray:\n" + Arrays.toString( array );
46    } // fim do método toString
47 } // fim da classe SimpleArray
```

Figura 26.7 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads.

O método `add` (linhas 19–40) permite que novos valores sejam inseridos no fim do array. A linha 21 armazena o valor `writeIndex` atual. A linha 26 coloca a thread que invoca `add` para dormir por um intervalo aleatório de 0 a 499 milissegundos. Isso é feito para tornar mais óbvios os problemas associados com o acesso não sincronizado a dados compartilhados. Depois que a thread não estiver mais dormindo, a linha 34 insere o valor passado para `add` no array do elemento especificado por `position`. As linhas 35–36 geram saída de uma mensagem que indica o nome da thread em execução, o valor que foi inserido no array e onde ele foi inserido. A expressão `Thread.currentThread.getName()` (linha 36) primeiro obtém uma referência a `Thread` em execução atualmente, então utiliza o método `getName` dessa `Thread` para obter seu nome. A linha 38 incrementa `writeIndex` para que a chamada seguinte para `add` insira um valor no próximo elemento do array. As linhas 43–46 sobrescrevem o método `toString` para criar uma representação `String` do conteúdo do array.

Classe ArrayWriter

A classe `ArrayWriter` (Figura 26.8) implementa a interface `Runnable` para definir uma tarefa para inserir valores em um objeto `SimpleArray`. O construtor (linhas 10–14) aceita dois argumentos — um número inteiro `value`, que é o primeiro valor que essa tarefa inserirá no objeto `SimpleArray`, e uma referência ao objeto `SimpleArray`. A linha 20 invoca o método `add` no objeto `SimpleArray`. A tarefa se completa depois que três números inteiros consecutivos que iniciam com `startValue` forem adicionados ao objeto `SimpleArray`.

```
1  // Figura 26.8: ArrayWriter.java
2  // Adiciona números inteiros a um array compartilhado com outros Runnables
3  import java.lang.Runnable;
4
```

```
5  public class ArrayWriter implements Runnable
6  {
7     private final SimpleArray sharedSimpleArray;
8     private final int startValue;
9
10    public ArrayWriter( int value, SimpleArray array )
11    {
12       startValue = value;
13       sharedSimpleArray = array;
14    } // fim do construtor
15
16    public void run()
17    {
18       for ( int i = startValue; i < startValue + 3; i++ )
19       {
20          sharedSimpleArray.add( i ); // adiciona um elemento ao array compartilhado
21       } // for final
22    } // fim do método run
23 } // fim da classe ArrayWriter
```

Figura 26.8 | Adiciona números inteiros a um array compartilhado com outros `Runnables`.

Classe SharedArrayTest

A classe `SharedArrayTest` (Figura 26.9) executa duas tarefas `ArrayWriter` que adicionam valores a um objeto `SimpleArray` simples. A linha 12 constrói um objeto `SimpleArray` de seis elementos. As linhas 15–16 criam duas novas tarefas `ArrayWriter`, uma que coloca os valores 1–3 no objeto `SimpleArray`, e outra que coloca os valores 11–13. As linhas 19–21 criam um `ExecutorService` e executam os dois `ArrayWriters`. A linha 23 invoca o método `shutDown` do `ExecutorService` para impedir que tarefas adicionais iniciem e para permitir que o aplicativo termine quando as tarefas em execução atualmente completarem a execução.

Lembre-se de que o método `ExecutorService shutdown` retorna imediatamente. Dessa maneira, qualquer código que apareça depois da chamada para o método `ExecutorService shutdown` na linha 23 continuará executando enquanto a thread `main` ainda estiver atribuída a um processador. Gostaríamos de gerar saída do objeto `SimpleArray` para mostrar os resultados *depois* de as threads completarem suas tarefas. Portanto, precisamos que o programa espere as threads completarem suas tarefas antes de `main` gerar saída do conteúdo do objeto `SimpleArray`. A interface `ExecutorService` fornece o método **`awaitTermination`** para esse propósito. Esse método retorna o controle para seu chamador quando todas as tarefas executando no `ExecutorService` se completarem ou quando o tempo-limite especificado expirar. Se todas as tarefas forem completadas antes de `awaitTermination` terminar, esse método retorna `true`; caso contrário, retorna `false`. Os dois argumentos para `awaitTermination` representam um valor de tempo-limite e uma unidade da medida especificada com uma constante da classe `TimeUnit` (nesse caso, `TimeUnit.MINUTES`). Nesse exemplo, se ambas as tarefas forem completadas antes de `awaitTermination` expirar, a linha 32 exibe o conteúdo do objeto `SimpleArray`. Caso contrário, as linhas 34–35 imprimem uma mensagem que indica que as tarefas não terminaram de executar antes de `awaitTermination`.

```
1  // Figura 26.9: SharedArrayTest.java
2  // Executa dois Runnables para adicionar elementos a um SimpleArray compartilhado.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedArrayTest
8  {
9     public static void main( String[] arg )
10    {
11       // constrói o objeto compartilhado
12       SimpleArray sharedSimpleArray = new SimpleArray( 6 );
13
14       // cria duas tarefas a serem gravadas no SimpleArray compartilhado
15       ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
16       ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );
17
18       // executa as tarefas com um ExecutorService
19       ExecutorService executor = Executors.newCachedThreadPool();
20       executor.execute( writer1 );
21       executor.execute( writer2 );
22
23       executor.shutdown();
```

```
24
25          try
26          {
27             // espera 1 minuto por ambos os gravadores terminarem a execução
28             boolean tasksEnded = executor.awaitTermination(
29                1, TimeUnit.MINUTES );
30
31             if ( tasksEnded )
32                System.out.println( sharedSimpleArray ); // imprime o conteúdo
33             else
34                System.out.println(
35                   "Timed out while waiting for tasks to finish." );
36          } // fim do try
37          catch ( InterruptedException ex )
38          {
39             System.out.println(
40                "Interrupted while waiting for tasks to finish." );
41          } // fim do catch
42       } // fim de main
43    } // fim da classe SharedArrayTest
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-2 wrote 11 to element 0.
Next write index: 4
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]
```

O primeiro `pool-1-thread-1` grava o valor `1` no elemento `0`. Depois, `pool-1-thread-2` grava o valor `11` no elemento `0`, sobrescrevendo assim o valor previamente armazenado.

Figura 26.9 | Executa dois `Runnable`s para inserir valores em um array compartilhado.

A saída na Figura 26.9 demonstra os problemas (destacados na saída) que podem ser causados pela falha em sincronizar o acesso a dados compartilhados. O valor `1` foi gravado no elemento `0` e, posteriormente, sobrescrito pelo valor `11`. Além disso, quando `writeIndex` foi incrementado por 3, nada foi gravado nesse elemento, conforme indicado por `0` nesse elemento do array impresso.

Lembre-se de que adicionamos chamadas para o método `Thread` `sleep` entre operações nos dados compartilhados para enfatizar a imprevisibilidade do agendamento de threads e aumentar a probabilidade de produzir a saída incorreta. É importante notar que mesmo se essas operações tivessem permissão de prosseguir em seu ritmo normal, você ainda poderia ver erros na saída do programa. Contudo, os processadores modernos podem tratar as operações simples do método `SimpleArray` `add` com tanta rapidez que talvez não fosse possível ver os erros causados pelas duas threads executando esse método simultaneamente, mesmo se você testasse o programa dezenas de vezes. Um dos desafios da programação de múltiplas threads é indicar erros com precisão — eles podem ocorrer tão raramente que um programa com bugs não produz resultados incorretos durante os testes, criando a ilusão de que o programa está correto.

26.5.2 Compartilhamento de dados sincronizados — tornando operações atômicas

Os erros de saída da Figura 26.9 podem ser atribuídos ao fato de que o objeto compartilhado, `SimpleArray`, não é **seguro para threads** — `SimpleArray` é suscetível a erros se acessado concorrentemente por múltiplas threads. O problema reside no método `add`, que armazena o valor de `writeIndex`, coloca um novo valor nesse elemento e, então, incrementa `writeIndex`. Esse método não apresentaria nenhum problema em programas de uma única thread. Contudo, se uma única thread obtiver o valor de `writeIndex`, não haverá garantia de que outra thread não poderá avançar e incrementar `writeIndex` antes de a primeira thread ter tido uma chance de colocar um valor no array. Se isso acontecer, a primeira thread estará gravando no array com base em um **valor obsoleto** de `writeIndex` — um valor que não é mais válido. Outra possibilidade é que uma thread pudesse obter o valor de `writeIndex` depois de outra thread adicionar um elemento ao array, mas antes de `writeIndex` ser incrementado. Nesse caso, também, a primeira thread gravaria no array com base em um valor inválido para `writeIndex`.

`SimpleArray` não é seguro para thread porque permite que qualquer número de threads leia e modifique dados compartilhados concorrentemente, o que pode causar erros. Para tornar `SimpleArray` seguro para threads, devemos assegurar que duas threads não podem acessá-lo ao mesmo tempo. Também devemos assegurar que, enquanto uma thread estiver no processo de armazenar `writeIndex`,

adicionar um valor ao array e incrementar `writeIndex`, nenhuma outra thread pode ler ou alterar o valor de `writeIndex` ou modificar o conteúdo do array em nenhum momento durante essas três operações. Em outras palavras, queremos que essas três operações — armazenar `writeIndex`, gravar no array, incrementar `writeIndex` — sejam uma **operação atômica**, que não pode ser dividida em suboperações menores. Podemos simular a atomicidade assegurando que apenas uma thread executa as três operações por vez. Qualquer outra thread que precisar realizar a operação deve esperar até a primeira thread ter terminado a operação `add` em sua totalidade.

A atomicidade pode ser alcançada utilizando a palavra-chave `synchronized`. Colocando nossas três suboperações em uma instrução `synchronized` ou método `synchronized`, permitimos que apenas uma thread por vez adquira o bloqueio e realize as operações. Quando essa thread tiver completado todas as operações no bloco `synchronized` e liberar o bloqueio, outra thread pode adquiri-lo e começar a executar as operações. Isso garante que uma thread que executa as operações verá os valores reais dos dados compartilhados e que esses valores não mudarão inesperadamente no meio das operações como resultado do fato de outra thread modificá-los.

Observação de engenharia de software 26.1

Coloque todos os acessos a dados mutáveis que podem ser compartilhados por múltiplas threads dentro de instruções `synchronized` ou métodos `synchronized` que sincronizam no mesmo bloqueio. Ao realizar múltiplas operações em dados compartilhados, mantenha o bloqueio pela totalidade da operação para assegurar que a operação seja efetivamente atômica.

Classe `SimpleArray` com sincronização

A Figura 26.10 exibe a classe `SimpleArray` com a sincronização apropriada. Observe que é ela idêntica à classe `SimpleArray` da Figura 26.7, exceto que `add` é agora um método `synchronized` (linha 20). Desse modo, apenas uma thread por vez pode executar esse método. Reutilizamos as classes `ArrayWriter` (Figura 26.8) e `SharedArrayTest` (Figura 26.9) do exemplo anterior.

```
1  // Figura 26.10: SimpleArray.java
2  // A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads
3  // threads with synchronization.
4  import java.util.Arrays;
5  import java.util.Random;
6
7  public class SimpleArray
8  {
9     private final int[] array; // o array de inteiros compartilhado
10    private int writeIndex = 0; // o índice do próximo elemento a ser gravado
11    private final static Random generator = new Random();
12
13    // constrói um SimpleArray de um dado tamanho
14    public SimpleArray( int size )
15    {
16       array = new int[ size ];
17    } // fim do construtor
18
19    // adiciona um valor ao array compartilhado
20    public synchronized void add( int value )
21    {
22       int position = writeIndex; // armazena o índice de gravação
23
24       try
25       {
26          // coloca a thread para dormir por 0-499 milissegundos
27          Thread.sleep( generator.nextInt( 500 ) );
28       } // fim do try
29       catch ( InterruptedException ex )
30       {
31          ex.printStackTrace();
32       } // fim do catch
33
34       // coloca valor no elemento correto
35       array[ position ] = value;
36       System.out.printf( "%s wrote %2d to element %d.\n",
37          Thread.currentThread().getName(), value, position );
38
39       ++writeIndex; // incrementa índice do elemento a ser gravado depois
40       System.out.printf( "Next write index: %d\n", writeIndex );
41    } // fim do método add
```

```
42
43       // utilizado para gerar saída do conteúdo do array de inteiros compartilhado
44       public String toString()
45       {
46          return "\nContents of SimpleArray:\n" + Arrays.toString( array );
47       } // fim do método toString
48    } // fim da classe SimpleArray
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
1 11 12 13 2 3
```

Figura 26.10 | A classe que gerencia um array de inteiros a ser compartilhado por múltiplas threads com a sincronização.

A linha 20 declara o método como `synchronized`, fazendo todas as operações nesse método comportarem-se como uma operação simples, atômica. A linha 22 realiza a primeira suboperação — armazenar o valor de `writeIndex`. A linha 35 define a segunda suboperação, gravar um elemento no elemento do índice `position`. A linha 39 incrementa `writeIndex`. Quando o método termina a execução na linha 41, a thread em execução libera o bloqueio `SimpleArray`, possibilitando que a outra thread comece a executar o método `add`.

No método `synchronized` `add`, imprimimos mensagens para o console indicando o progresso de threads enquanto elas executam esse método, além de realizar as operações reais necessárias para inserir um valor no array. Fazemos isso para que as mensagens sejam impressas na ordem correta, o que nos permite ver se o método está adequadamente sincronizado comparando essas saídas com as do exemplo anterior, não sincronizado. Continuamos a gerar saída de mensagens a partir de blocos `synchronized` nos exemplos posteriores por propósitos de demonstração, mas, em geral, a E/S *não deve* ser realizada em blocos `synchronized`, porque é importante minimizar o período de tempo que um objeto permanece "bloqueado". Além disso, a linha 27 nesse exemplo chama o método `Thread` `sleep` para enfatizar a imprevisibilidade do agendamento de thread. Você *nunca* deve chamar `sleep` enquanto mantém um bloqueio em um aplicativo real.

Dica de desempenho 26.2

Mantenha a duração de instruções `synchronized` o mais curta possível mantendo ao mesmo tempo a sincronização necessária. Isso minimiza o tempo de espera por threads bloqueadas. Evite realizar E/S, cálculos longos e operações que não exigem a sincronização enquanto mantiver um bloqueio.

Outra nota sobre segurança de thread: Dissemos que é necessário sincronizar o acesso a todos os dados que podem ser compartilhados por múltiplas threads. De fato, essa sincronização é necessária somente para **dados mutáveis**, ou dados que podem mudar durante seu tempo de vida. Se os dados compartilhados não mudarem em um programa de múltiplas threads, então não é possível para uma thread ver valores antigos ou incorretos em consequência da manipulação desses dados por outra thread.

Ao compartilhar dados imutáveis por threads, declare os campos de dados correspondentes `final` para indicar que os valores das variáveis não mudarão depois de eles serem inicializados. Isso impede a modificação acidental dos dados compartilhados posteriormente em um programa, o que poderia comprometer a segurança de thread. Rotular referências de objeto como `final` indica que a referência não mudará, mas não garante que o próprio objeto seja imutável — isso depende inteiramente das propriedades do objeto. Contudo, ainda é uma boa prática marcar referências que não mudarão como `final`, já que fazer isso força o construtor do objeto a ser atômico — o objeto será inteiramente construído com todos seus campos inicializados antes de o programa acessá-lo.

Boa prática de programação 26.1

Sempre declare como `final` campos de dados que você não espera que mudem. As variáveis primitivas que são declaradas como `final` podem ser seguramente compartilhadas pelas threads. Uma referência de objeto que é declarada como `final` assegura que o objeto que ela referencia será totalmente construído e inicializado antes de ser utilizado pelo programa, e impede que a referência aponte para outro objeto.

26.6 Relacionamento entre produtor e consumidor sem sincronização

Em um **relacionamento produtor/consumidor**, a parte **produtora** de um aplicativo gera dados e os armazena em um objeto compartilhado e a parte **consumidora** do aplicativo lê os dados do objeto compartilhado. O relacionamento produtor/consumidor separa a tarefa de identificar o trabalho a ser feito a partir das tarefas envolvidas, de fato, na realização do trabalho. Um exemplo de relacionamento produtor/consumidor comum é o **spooling de impressão**. Embora uma impressora não pudesse estar disponível quando você quer imprimir a partir de um aplicativo (isto é, o produtor), você ainda pode "completar" a tarefa de impressão, uma vez que os dados são temporariamente colocados em disco até a impressora tornar-se disponível. De modo semelhante, quando a impressora (isto é, um consumidor) estiver disponível, ela não precisa esperar até que um usuário atual queira imprimir. Os trabalhos de impressão em spool podem ser impressos assim que a impressora se disponibilizar. Outro exemplo de relacionamento produtor/consumidor é um aplicativo que copia dados em CDs colocando os dados em um buffer de tamanho fixo, o qual é esvaziado à medida que a unidade de CD-RW grava os dados no CD.

Em um relacionamento produtor/consumidor de múltiplas threads, uma **thread produtora** gera dados e os coloca em um objeto compartilhado chamado **buffer**. Uma **thread consumidora** lê dados do buffer. Esse relacionamento requer que a sincronização assegure que os valores sejam produzidos e consumidos corretamente. Todas as operações sobre dados mutáveis que são compartilhados pelas múltiplas threads (por exemplo, dados no buffer) devem ser guardadas com um bloqueio para impedir corrupção, como discutido na Seção 26.5. As operações nos dados de buffers compartilhados por uma thread consumidora e produtora são também **dependentes de estado** — as operações devem prosseguir somente se o buffer estiver no estado correto. Se o buffer estiver em um estado não cheio, o produtor pode produzir; se o buffer estiver em um estado não vazio, o consumidor pode consumir. Todas as operações que acessam o buffer devem utilizar a sincronização para assegurar que os dados são gravados no buffer ou lidos do buffer somente se o buffer estiver no estado adequado. Se o produtor que está tentando colocar os próximos dados no buffer determina que ele está cheio, a thread produtora deve esperar até haver espaço para gravar um novo valor. Se uma thread consumidora encontrar o buffer vazio ou determinar que os dados anteriores já foram lidos, a consumidora também deve esperar que novos dados sejam disponibilizados.

Considere como os erros de lógica podem surgir se não sincronizarmos o acesso entre múltiplas threads que manipulam dados compartilhados. Nosso próximo exemplo (Figura 26.11–Figura 26.15) implementa um relacionamento produtor/consumidor sem a sincronização adequada. Uma thread produtora grava os números 1 a 10 em um buffer compartilhado — uma posição de memória única compartilhada entre duas threads (uma variável `int` simples chamada `buffer` na linha 6 da Figura 26.14 nesse exemplo). A thread consumidora lê esses dados do buffer compartilhado e os exibe. A saída do programa mostra os valores que a produtora grava (produz) no buffer compartilhado e os valores que a consumidora lê (consome) a partir do buffer compartilhado.

Cada valor que a thread produtora gravar no buffer compartilhado deve ser consumido exatamente uma vez pela thread consumidora. Entretanto, as threads nesse exemplo erroneamente não estão sincronizadas. Portanto, os dados podem ser perdidos ou deturpados se a produtora colocar novos dados no buffer compartilhado antes de a consumidora ler os dados anteriores. Além disso, os dados podem ser duplicados incorretamente se a consumidora consumir os dados outra vez antes de a produtora produzir o próximo valor. Para mostrar essas possibilidades, a thread consumidora no exemplo a seguir mantém um total de todos os valores que ele lê. A thread produtora produz valores de 1 a 10. Se a consumidora ler cada valor produzido uma vez e apenas uma vez, o total será 55. Entretanto, se executar esse programa várias vezes, você verá que o total nem sempre é 55 (como mostrado nas saídas da Figura 26.15). Para enfatizar a questão, as threads produtoras e consumidoras no exemplo dormem por intervalos aleatórios de até três segundos entre a execução de suas tarefas. Portanto, não sabemos quando a thread produtora tentará gravar um novo valor ou quando a thread consumidora tentará ler um valor.

Implementando o relacionamento produtor/consumidor

O programa consiste na interface `Buffer` (Figura 26.11) e nas classes `Producer` (Figura 26.12), `Consumer` (Figura 26.13), `UnsynchronizedBuffer` (Figura 26.14) e `SharedBufferTest` (Figura 26.15). A interface `Buffer` (Figura 26.11) declara os métodos `set` (linha 6) e `get` (linha 9) que um `Buffer` (como `UnsynchronizedBuffer`) deve implementar para permitir que a thread `Producer` coloque um valor no `Buffer` e a thread `Consumer` recupere um valor do `Buffer`, respectivamente. Nos exemplos subsequentes, os métodos `set` e `get` chamarão os métodos que lançam `InterruptedExceptions`. Declaramos cada método com uma cláusula `throws` aqui para que não tenhamos de modificar essa interface para os exemplos posteriores.

```
1  // Figura 26.11: Buffer.java
2  // Interface Buffer especifica métodos chamados por Producer e Consumer.
3  public interface Buffer
4  {
5     // coloca o valor int no Buffer
6     public void set( int value ) throws InterruptedException;
7
8     // retorna o valor int a partir do Buffer
9     public int get() throws InterruptedException;
10 } // fim da interface Buffer
```

Figura 26.11 | A interface buffers especifica os métodos chamados por `Producer` e `Consumer`.

A classe `Producer` (Figura 26.12) implementa a interface `Runnable`, permitindo que ela seja executada como uma tarefa em uma thread separada. O construtor (linhas 11–14) inicializa a referência `Buffer sharedLocation` com um objeto criado em `main` (linha 14 da Figura 26.15) e passado para o construtor. Como veremos, esse é um objeto `UnsynchronizedBuffer` que implementa a interface `Buffer` sem sincronizar acesso ao objeto compartilhado. A thread `Producer` nesse programa executa as tarefas especificadas no método `run` (linhas 17–39). Cada iteração do loop (linhas 21–35) invoca o método `Thread` `sleep` (linha 25) para colocar a thread `Producer` no estado de *espera sincronizada* de um intervalo aleatório de tempo entre 0 e 3 segundos. Quando a thread acordar, a linha 26 passa o valor da variável de controle `count` para o método `set` do objeto `Buffer` para configurar o valor do buffer compartilhado. As linhas 27–28 mantêm um total de todos os valores produzidos até aqui e gera saída desse valor. Quando o loop é concluído, as linhas 37–38 exibem uma mensagem indicando que `Producer` concluiu a produção de dados e está terminando. Em seguida, o método `run` termina, o que indica que `Producer` completou sua tarefa. Observe que qualquer método chamado a partir do método `run` do `Runnable` (por exemplo, o método `Buffer` `set`) executa como parte da thread de execução dessa tarefa. Esse fato se torna importante na Seção 26.7 quando adicionamos a sincronização ao relacionamento produtor/consumidor.

```
1  // Figura 26.12: Producer.java
2  // Producer com um método run que insere os valores de 1 a 10 no buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // referência a objeto compartilhado
9
10    // construtor
11    public Producer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // fim do construtor Producer
15
16    // armazena os valores de 1 a 10 em sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
21       for ( int count = 1; count <= 10; count++ )
22       {
23          try // dorme de 0 a 3 segundos, então coloca valor em Buffer
24          {
25             Thread.sleep( generator.nextInt( 3000 ) ); // sono aleatório
26             sharedLocation.set( count ); // configura valor no buffer
27             sum += count; // incrementa soma de valores
28             System.out.printf( "\t%2d\n", sum );
29          } // fim do try
30          // se as linhas 25 ou 26 são interrompidas, imprimem o rastreamento de pilha
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // for final
36
37       System.out.println(
38          "Producer done producing\nTerminating Producer" );
39    } // fim do método run
40 } // fim da classe Producer
```

Figura 26.12 | `Producer` com o método `run` que insere os valores 1 a 10 no buffer.

A classe `Consumer` (Figura 26.13) também implementa a interface `Runnable`, permitindo que `Consumer` execute concorrentemente com `Producer`. As linhas 11–14 inicializam a referência `Buffer sharedLocation` com um objeto que implementa a interface `Buffer` (criada em `main`, Figura 26.15) e passada para o construtor como o parâmetro `shared`. Como veremos, esse é o mesmo objeto `UnsynchronizedBuffer` que é utilizado para inicializar o objeto `Producer` — portanto, as duas threads compartilham o mesmo objeto. A thread `Consumer` nesse programa realiza as tarefas especificadas no método `run` (linhas 17–39). As linhas 21–35 iteram 10 vezes. Cada iteração invoca o método `Thread` `sleep` (linha 26) para colocar a thread `Consumer` no estado de *espera sincronizada* por até 3 segundos. Em

seguida, a linha 27 utiliza o método `get` de `Buffer` para recuperar o valor no buffer compartilhado, então adiciona o valor à variável `sum`. A linha 28 exibe o total de todos os valores consumidos até agora. Quando o loop for concluído, as linhas 37–38 exibem uma linha que indica a soma dos valores consumidos. Então o método `run` termina, o que indica que `Consumer` completou sua tarefa. Depois que ambas as threads entram no estado *terminada*, o programa é encerrado.

```
1  // Figura 26.13: Consumer.java
2  // Consumer com um método run que itera, lendo 10 valores do buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // referência a objeto compartilhado
9
10    // construtor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // fim do construtor Consumer
15
16    // lê o valor do sharedLocation 10 vezes e soma os valores
17    public void run()
18    {
19       int sum = 0;
20
21       for ( int count = 1; count <= 10; count++ )
22       {
23          // dorme de 0 a 3 segundos, lê o valor do buffer e adiciona a soma
24          try
25          {
26             Thread.sleep( generator.nextInt( 3000 ) );
27             sum += sharedLocation.get();
28             System.out.printf( "\t\t\t%2d\n", sum );
29          } // fim do try
30          // se as linhas 26 ou 27 são interrompidas, imprimem o rastreamento de pilha
31          catch ( InterruptedException exception )
32          {
33             exception.printStackTrace();
34          } // fim do catch
35       } // for final
36
37       System.out.printf( "\n%s %d\n%s\n",
38          "Consumer read values totaling", sum, "Terminating Consumer" );
39    } // fim do método run
40 } // fim da classe Consumer
```

Figura 26.13 | `Consumer` com um método `run` que itera, lendo 10 valores de buffer.

[*Nota*: Utilizamos o método `sleep` no método `run` das classes `Producer` e `Consumer` para enfatizar o fato de que, em aplicativos com múltiplas threads, é imprevisível quando cada thread realizará sua tarefa e por quanto tempo ela realizará a tarefa quando tiver um processador. Normalmente, essas questões de agendamento de thread estão além do controle do desenvolvedor em Java. Nesse programa, as tarefas da nossa thread são bem simples — `Producer` grava os valores de `1` a `10` no buffer e `Consumer` lê 10 valores do buffer e adiciona cada valor à variável `sum`. Sem a chamada de método `sleep`, e se `Producer` executasse primeiro, considerando os processadores fenomenalmente rápidos de hoje, `Producer` possivelmente completaria sua tarefa antes de `Consumer` ter uma chance de executar. Se `Consumer` executasse primeiro, ele possivelmente consumiria dados da lixeira dez vezes, então terminaria antes que `Producer` pudesse produzir o primeiro valor real.]

A classe `UnsynchronizedBuffer` (Figura 26.14) implementa a interface `Buffer` (linha 4). Um objeto dessa classe é compartilhado entre `Producer` e `Consumer`. A linha 6 declara a variável de instância `buffer` e a inicializa com o valor `-1`. Esse valor é utilizado para demonstrar o caso em que `Consumer` tenta consumir um valor antes de `Producer` colocar alguma vez um valor em `buffer`. Os métodos `set` (linhas 9–13) e `get` (linhas 16–20) não sincronizam o acesso ao campo `buffer`. O método `set` simplesmente atribui seu argumento a `buffer` (linha 12) e o método `get` simplesmente retorna o valor de `buffer` (linha 19).

```
1  // Figura 26.14: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer mantém o número inteiro compartilhado que é acessado por
3  // uma thread produtora e uma thread consumidora via métodos set e get.
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // compartilhado pelas threads producer e consumer
7
8     // coloca o valor no buffer
9     public void set( int value ) throws InterruptedException
10    {
11       System.out.printf( "Producer writes\t%2d", value );
12       buffer = value;
13    } // fim do método set
14
15    // retorna valor do buffer
16    public int get() throws InterruptedException
17    {
18       System.out.printf( "Consumer reads\t%2d", buffer );
19       return buffer;
20    } // fim do método get
21 } // fim da classe UnsynchronizedBuffer
```

Figura 26.14 | `UnsynchronizedBuffer` mantém o inteiro compartilhado que é acessado por uma thread produtora e uma consumidora por meio dos métodos `set` e `get`.

Na classe `SharedBufferTest` (Figura 26.15), a linha 11 cria um `ExecutorService` para executar os `Runnables` `Producer` e `Consumer`. A linha 14 cria um objeto `UnsynchronizedBuffer` e o atribui à variável `Buffer` `sharedLocation`. Esse objeto armazena os dados que as threads `Producer` e `Consumer` compartilharão. As linhas 23–24 criam e executam `Producer` e `Consumer`. Observe que os construtores `Producer` e `Consumer` recebem o mesmo objeto `Buffer` (`sharedLocation`), assim, cada objeto é inicializado com uma referência ao mesmo `Buffer`. Essas linhas também carregam implicitamente as threads e chamam o método `run` de cada `Runnable`. Por fim, a linha 26 chama o método `shutdown` de modo que o aplicativo possa terminar quando as threads que estão executando `Producer` e `Consumer` completarem suas tarefas. Quando `main` termina (linha 27), a thread principal de execução entra no estado *terminada*.

```
1  // Figura 26.15: SharedBufferTest.java
2  // Aplicativo com duas threads que manipulam um buffer não sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // cria UnsynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
16       System.out.println(
17          "Action\t\tValue\tSum of Produced\tSum of Consumed" );
18       System.out.println(
19          "------\t\t-----\t---------------\t---------------\n" );
20
21       // executa Producer e Consumer, fornecendo-lhes acesso
22       // a sharedLocation
23       application.execute( new Producer( sharedLocation ) );
24       application.execute( new Consumer( sharedLocation ) );
25
26       application.shutdown(); // termina o aplicativo quando as tarefas terminam
27    } // fim de main
28 } // fim da classe SharedBufferTest
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------
Producer writes  1       1
Producer writes  2       3                              —— 1 é perdido
Producer writes  3       6                              —— 2 é perdido
Consumer reads   3                       3
Producer writes  4      10
Consumer reads   4                       7
Producer writes  5      15
Producer writes  6      21                              —— 5 é perdido
Producer writes  7      28                              —— 6 é perdido
Consumer reads   7                      14
Consumer reads   7                      21              —— 7 lido novamente
Producer writes  8      36
Consumer reads   8                      29
Consumer reads   8                      37              —— 8 é lido novamente
Producer writes  9      45
Producer writes 10      55                              —— 9 é perdido

Producer done producing
Terminating Producer
Consumer reads  10                      47
Consumer reads  10                      57              —— 10 lido novamente
Consumer reads  10                      67              —— 10 lido novamente
Consumer reads  10                      77              —— 10 lido novamente

Consumer read values totaling 77
Terminating Consumer
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------
Consumer reads  -1                      -1              —— lê -1 dados inválidos
Producer writes  1       1
Consumer reads   1                       0
Consumer reads   1                       1              —— 1 lido novamente
Consumer reads   1                       2              —— 1 lido novamente
Consumer reads   1                       3              —— 1 lido novamente
Consumer reads   1                       4              —— 1 lido novamente
Producer writes  2       3
Consumer reads   2                       6
Producer writes  3       6
Consumer reads   3                       9
Producer writes  4      10
Consumer reads   4                      13
Producer writes  5      15
Producer writes  6      21                              —— 5 é perdido
Consumer reads   6                      19

Consumer read values totaling 19
Terminating Consumer
Producer writes  7      28                              —— 7 nunca lido
Producer writes  8      36                              —— 8 nunca lido
Producer writes  9      45                              —— 9 nunca lido
Producer writes 10      55                              —— 10 nunca lido

Producer done producing
Terminating Producer
```

Figura 26.15 | Aplicativo com duas threads que manipulam um buffer não sincronizado.

Considerando a visão geral desse exemplo, lembre-se de que gostaríamos que `Producer` executasse primeiro e que cada valor produzido por `Producer` fosse consumido exatamente uma vez por `Consumer`. Entretanto, quando estudamos a primeira saída da Figura 26.15, vemos que `Producer` grava os valores `1`, `2` e `3` antes de `Consumer` ler seu primeiro valor (`3`). Portanto, os valores `1` e `2` são perdidos. Posteriormente, os valores `5`, `6` e `9` são perdidos, enquanto `7` e `8` são lidos duas vezes e `10` é lido quatro vezes. Então, a primeira saída produz um total incorreto de 77, em vez de um total correto de 55. Na segunda saída, `Consumer` lê o valor de `-1` antes de `Producer` ter gravado um valor. O `Consumer` lê o valor `1` cinco vezes antes de o `Producer` gravar o valor `2`. Nesse meio-tempo, os valores `5`, `7`, `8`, `9` e `10` são inteiramente perdidos — os quatro últimos porque `Consumer` termina antes `Producer`. Um total consumidor incorreto de 19 é exibido. (As linhas na saída em que `Producer` ou `Consumer` atuou fora de ordem são destacadas.)

Dica de prevenção de erro 26.1

O acesso a um objeto compartilhado pelas threads concorrentes deve ser controlado cuidadosamente ou um programa pode produzir resultados incorretos.

Para resolver os problemas de dados perdidos e duplicados, a Seção 26.7 apresenta um exemplo em que utilizamos uma `ArrayBlockingQueue` (do pacote `java.util.concurrent`) para sincronizar o acesso ao objeto compartilhado, garantindo que cada valor será processado uma única vez.

26.7 Relacionamento de produtor/consumidor: ArrayBlockingQueue

Um modo de sincronizar as threads consumidora e produtora é utilizar as classes do pacote de concorrência do Java que encapsulam a sincronização para você. O Java inclui a classe **`ArrayBlockingQueue`** (do pacote `java.util.concurrent`) — uma classe de buffers totalmente implementada, segura para threads que implementa a interface **`BlockingQueue`**. A interface estende a interface `Queue` discutida no Capítulo 20 e declara os métodos **`put`** e **`take`**, equivalentes de bloqueio dos métodos `Queue` `offer` e `poll`, respectivamente. O método `put` coloca um elemento no fim da `BlockingQueue`, esperando se a fila estiver cheia. O método `take` removerá um elemento da cabeça da `BlockingQueue`, esperando se a fila estiver vazia. Esses métodos fazem da classe `ArrayBlockingQueue` uma boa escolha para implementar um buffer compartilhado. Como o método `put` bloqueia até haver espaço no buffer para gravar dados, e o método `take` bloqueia até que haja novos dados para leitura, o produtor deve produzir um valor primeiro, o consumidor consumir corretamente apenas depois de o produtor gravar um valor e o produtor produzir corretamente o valor seguinte (depois do primeiro) apenas depois de o consumidor ler o valor anterior (ou primeiro). `ArrayBlockingQueue` armazena os dados compartilhados em um array. O tamanho do array é especificado como um argumento para o construtor `ArrayBlockingQueue`. Uma vez criado, um `ArrayBlockingQueue` tem o tamanho fixado e não expandirá para acomodar elementos extras.

As figuras 26.16–26.17 demonstram um `Producer` e um `Consumer` acessando um `ArrayBlockingQueue`. A classe `BlockingBuffer` (Figura 26.16) utiliza um objeto `ArrayBlockingQueue` que armazena um `Integer` (linha 7). A linha 11 cria o `ArrayBlockingQueue` e passa `1` para o construtor a fim de que o objeto mantenha um valor único, como fizemos com o `UnsynchronizedBuffer` da Figura 26.14. Observe que as linhas 7 e 11 usam genéricos, que discutimos nos capítulos 20–21. Discutimos buffers de múltiplos elementos na Seção 26.9. Como nossa classe `BlockingBuffer` utiliza a classe `ArrayBlockingQueue` segura para threads para gerenciar o acesso ao buffer compartilhado, `BlockingBuffer` é em si segura para threads, embora não tivéssemos implementado a sincronização.

```
1  // Figura 26.16: BlockingBuffer.java
2  // Criando um buffer sincronizado com um ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private final ArrayBlockingQueue<Integer> buffer; // buffer compartilhado
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>( 1 );
12    } // fim do construtor BlockingBuffer
13
14    // coloca o valor no buffer
15    public void set( int value ) throws InterruptedException
16    {
17       buffer.put( value ); // coloca o valor no buffer
18       System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
19          "Buffer cells occupied: ", buffer.size() );
20    } // fim do método set
21
22    // retorna valor do buffer
23    public int get() throws InterruptedException
24    {
25       int readValue = buffer.take(); // remove o valor do buffer
26       System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
27          readValue, "Buffer cells occupied: ", buffer.size() );
28
29       return readValue;
30    } // fim do método get
31 } // fim da classe BlockingBuffer
```

Figura 26.16 | Criando um buffer sincronizado com um `ArrayBlockingQueue`.

`BlockingBuffer` implementa a interface `Buffer` (Figura 26.11) e utiliza as classes `Producer` (Figura 26.12 modificada para remover a linha 28) e `Consumer` (Figura 26.13 modificada para remover a linha 28) do exemplo na Seção 26.6. Essa abordagem demonstra que as threads acessando o objeto compartilhado não estão cientes de que seus acessos ao buffer estão sincronizados agora. A sincronização é tratada inteiramente nos métodos `set` e `get` de `BlockingBuffer` chamando os métodos sincronizados de `ArrayBlockingQueue` `put` e `take`, respectivamente. Assim, os `Runnables` `Producer` e `Consumer` são sincronizados de forma adequada simplesmente chamando os métodos `set` e `get` do objeto compartilhado.

A linha 17 no método `set` (Figura 26.16, linhas 15–20) chama o método `put` do objeto `ArrayBlockingQueue`. Essa chamada de método bloqueia, se necessário, até haver espaço no `buffer` para colocar o `value`. O método `get` (linhas 23–30) chama o método `take` do objeto `ArrayBlockingQueue` (linha 25). Essa chamada de método bloqueia se, necessário, até haver um elemento no `buffer` para remover. As linhas 18–19 e 26–27 utilizam o método **`size`** do objeto `ArrayBlockingQueue` para exibir o número total de elementos atualmente em `ArrayBlockingQueue`.

A classe `BlockingBufferTest` (Figura 26.17) contém o método `main` que carrega o aplicativo. A linha 12 cria um `ExecutorService`, e a linha 15 cria um objeto `BlockingBuffer` e atribui sua referência à variável `Buffer sharedLocation`. As linhas 17–18 executam `Producer` e `Consumer` `Runnables`. A linha 19 chama o método `shutdown` para encerrar o aplicativo quando as threads terminarem de executar as tarefas de `Producer` e `Consumer`.

```
1  // Figura 26.17: BlockingBufferTest.java
2  // Duas threads que manipulam corretamente um buffer de bloqueio
3  // implementa o relacionamento produtor/consumidor.
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.Executors;
6
7  public class BlockingBufferTest
8  {
9     public static void main( String[] args )
10    {
11       // cria novo pool de threads com duas threads
12       ExecutorService application = Executors.newCachedThreadPool();
13
14       // cria BlockingBuffer para armazenar ints
15       Buffer sharedLocation = new BlockingBuffer();
16
17       application.execute( new Producer( sharedLocation ) );
18       application.execute( new Consumer( sharedLocation ) );
19
20       application.shutdown();
21    } // fim de main
22 } // fim da classe BlockingBufferTest
```

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1
```

```
Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Figura 26.17 | Duas threads que manipulam um buffer de bloqueio que corretamente implementa o relacionamento produtor/consumidor.

Observe que enquanto os métodos `put` e `take` de `ArrayBlockingQueue` são corretamente sincronizados, os métodos `BlockingBuffer` `set` e `get` (Figura 26.16) não são declarados para ser sincronizados. Assim, as instruções realizadas no método `set` — a operação `put` (linha 17) e a saída (linhas 18–19) — não são atômicas; nem são as instruções no método `get` — a operação `take` (linha 25) e a saída (linhas 26–27). Assim não há nenhuma garantia de que cada saída ocorrerá imediatamente depois da operação `put` ou `take` correspondente, e as saídas podem aparecer fora de ordem. Mesmo se aparecerem fora da ordem, o objeto `ArrayBlockingQueue` está sincronizando corretamente o acesso aos dados, como evidenciado pelo fato de a soma dos valores lidos pelo consumidor estar sempre correta.

26.8 Relacionamento entre produtor e consumidor com sincronização

O exemplo anterior mostrou como múltiplas threads podem compartilhar um buffer de um único elemento de maneira segura para threads utilizando a classe `ArrayBlockingQueue` que encapsula a sincronização necessária para proteger os dados compartilhados. Com propósitos educativos, agora explicamos como você mesmo pode implementar um buffer compartilhado utilizando a palavra-chave `synchronized` e os métodos da classe `Object`. Utilizar um `ArrayBlockingQueue` resultará em um código com maior capacidade de manutenção e melhor execução.

O primeiro passo na sincronização de acesso ao buffer é implementar os métodos `get` e `set` como métodos `synchronized`. Isso exige que uma thread obtenha o bloqueio de monitor no objeto `Buffer` antes de tentar acessar os dados do buffer, mas não assegura automaticamente que a thread realizará uma operação somente se o buffer estiver no estado apropriado. Precisamos de um modo de permitir que as nossas threads esperem, o que depende de certas condições serem ou não verdadeiras. Em caso de colocar um novo item no buffer, a condição que permite à operação prosseguir é que o buffer não esteja cheio. No caso da busca de um item do buffer, a condição que permite que a operação prossiga é que o buffer não esteja vazio. Se a condição em questão for verdadeira, a operação pode prosseguir; se falsa, a thread deve esperar até que ela se torne verdadeira. Quando uma thread estiver esperando uma condição, ela é removida da disputa do processador e colocada no estado de *espera* e o bloqueio que ela mantém é liberado.

Métodos `wait`, `notify` e `notifyAll`

Os métodos `Object` `wait`, `notify` e `notifyAll`, que são herdados por todas outras classes, podem ser utilizados com condições para fazer as threads esperarem quando não puderem realizar suas tarefas. Se uma thread obtiver o bloqueio de monitor em um objeto, ela determina então que não pode continuar com sua tarefa nesse objeto até alguma condição ser satisfeita, a thread pode chamar método `Object` **`wait`** no objeto `synchronized`; isso lança o bloqueio de monitor no objeto, e a thread permanece no estado de *espera* enquanto outras threads tentam inserir a(s) instrução(ões) ou método(s) do objeto `synchronized`. Quando uma thread que executa uma instrução (ou método) `synchronized` completa ou satisfaz a condição que outra thread pode estar esperando, ela pode chamar o método `Object` **`notify`** no objeto `synchronized` para permitir que uma thread em espera transite para o estado *executável* novamente. Nesse ponto, a thread que transitou do estado de *espera* para o estado *executável* pode tentar readquirir o bloqueio de monitor no objeto. Mesmo se a thread for capaz de readquirir o bloqueio de monitor, ela ainda pode não ser capaz de realizar sua tarefa nesse momento — caso em que irá entrar novamente no estado de *espera* e liberará implicitamente o bloqueio de monitor. Se uma thread chamar **`notifyAll`** em um objeto `synchronized`, então todas as threads que esperam o bloqueio de monitor se tornarão elegíveis para readquirir o bloqueio (isto é, todas elas transitarão para o estado *executável*). Lembre-se de que somente uma thread por vez pode obter o bloqueio de monitor no objeto — as outras threads que tentarem adquirir o mesmo bloqueio de monitor serão *bloqueadas* até que o bloqueio de monitor torne-se disponível novamente (isto é, até que nenhuma outra thread esteja executando em uma instrução `synchronized` desse objeto).

Erro comum de programação 26.1

É um erro se uma thread emitir um `wait`, `notify` ou `notifyAll` sobre um objeto sem adquirir um bloqueio para ele. Isso causa uma `IllegalMonitorStateException`.

Dica de prevenção de erro 26.2

É uma boa prática utilizar `notifyAll` para notificar threads em espera *para tornarem-se* executáveis. *Fazer isso evita a possibilidade de o programa se esquecer das threads em espera, que, se não fosse por isso, morreriam de inanição.*

O aplicativo nas figuras 26.18–26.19 demonstra uma `Producer` e uma `Consumer` acessando um buffer compartilhado com sincronização. Nesse caso, `Producer` sempre produz um valor primeiro, `Consumer` só consome corretamente depois de `Producer` produzir um valor e `Producer` produzir corretamente o próximo valor apenas depois de `Consumer` consumir o valor anterior (ou o primeiro valor). Reutilizamos a interface `Buffer` e as classes `Producer` e `Consumer` do exemplo na Seção 26.6, exceto a linha 28 que é removida das classes `Producer` e `Consumer`. A sincronização é tratada nos métodos `set` e `get` da classe `SynchronizedBuffer` (Figura 26.18), que implementa a interface `Buffer` (linha 4). Assim, os métodos `run` de `Producer` e `Consumer` simplesmente chamam os métodos `set` e `get` `synchronized` do objeto compartilhado.

```
1  // Figura 26.18: SynchronizedBuffer.java
2  // Sincronizando acesso a dados compartilhados
3  // com  os métodos wait e notifyAll de Object.
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // compartilhado pelas threads producer e consumer
7     private boolean occupied = false; // se o buffer estiver ocupado
8
9     // coloca o valor no buffer
10    public synchronized void set( int value ) throws InterruptedException
11    {
12       // enquanto não houver posições vazias, coloca a thread em estado de espera
13       while ( occupied )
14       {
15          // envia informações de thread e as informações de buffer para a saída, então espera
16          System.out.println( "Producer tries to write." );
17          displayState( "Buffer full. Producer waits." );
18          wait();
19       } // fim do while
20
21       buffer = value; // configura novo valor de buffer
22
23       // indica que a produtora não pode armazenar outro valor
24       // até a consumidora recuperar valor atual de buffer
25       occupied = true;
26
27       displayState( "Producer writes " + buffer );
28
29       notifyAll(); // instrui thread(s) em espera a entrar no estado executável
30    } // fim do método set; libera bloqueio em SynchronizedBuffer
31
32    // retorna valor do buffer
33    public synchronized int get() throws InterruptedException
34    {
35       // enquanto os dados não são lidos, coloca thread em estado de espera
36       while ( !occupied )
37       {
38          // envia informações de thread e as informações de buffer para a saída, então espera
39          System.out.println( "Consumer tries to read." );
40          displayState( "Buffer empty. Consumer waits." );
41          wait();
42       } // fim do while
43
44       // indica que a produtora pode armazenar outro valor
45       // porque a consumidora acabou de recuperar o valor do buffer
46       occupied = false;
47
48       displayState( "Consumer reads " + buffer );
49
50       notifyAll(); // instrui thread(s) em espera a entrar no estado executável
51
52       return buffer;
53    } // fim do método get; libera bloqueio em SynchronizedBuffer
54
55    // exibe a operação atual e o estado de buffer
56    public void displayState( String operation )
57    {
58       System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
59          occupied );
60    } // fim do método displayState
61 } // fim da classe SynchronizedBuffer
```

Figura 26.18 | Sincronizando acesso aos dados compartilhados utilizando os métodos `Object` `wait` e `notifyAll`.

Campos e métodos da classe SynchronizedBuffer

A classe SynchronizedBuffer contém os campos buffer (linha 6) e occupied (linha 7). Os métodos set (linhas 10–30) e get (linhas 33–53) são declarados como synchronized — somente uma thread pode chamar qualquer desses métodos de uma vez em um pormenor objeto SynchronizedBuffer. O campo occupied é utilizado para determinar se é a vez do Producer ou do Consumer de realizar uma tarefa. Esse campo é utilizado nas expressões condicionais tanto no método set como no get. Se occupied for false, então buffer está vazio e, assim Consumer não poderá ler o valor do buffer, mas Producer poderá colocar um valor no buffer. Se occupied for true, Consumer poderá ler um valor do buffer, mas Producer não poderá colocar um valor em buffer.

O método set e a thread Producer

Quando o método run da thread Producer invoca o método synchronized set, a thread tenta implicitamente adquirir o bloqueio de monitor do objeto SynchronizedBuffer. Se o bloqueio de monitor estiver disponível, a thread Producer adquire implicitamente o bloqueio. Então o loop while nas linhas 13–19 determina se occupied é true. Se for, buffer está cheio, então a linha 16 gera saída de uma mensagem indicando que a thread está tentando gravar um valor, e a linha 17 invoca o método displayState (linhas 56–60) para gerar saída de outra mensagem indicando que o buffer está cheio e que a thread Producer está esperando até haja espaço. A linha 18 invoca o método wait (herdado de Object por SynchronizedBuffer) para colocar a thread que chamou o método set (isto é, a thread Producer) no estado de *espera* pelo objeto SynchronizedBuffer. A chamada a wait faz com que a thread chamadora libere implicitamente o bloqueio do objeto SynchronizedBuffer. Isso é importante porque a thread não pode realizar atualmente sua tarefa e porque outras threads (nesse caso, a Consumer) devem ter permissão de acessar o objeto para permitir que a condição (occupied) mude. Agora outra thread pode tentar adquirir o bloqueio do objeto SynchronizedBuffer e invocar o método set ou get do objeto.

A thread Producer permanece no estado de *espera* até outra thread notificar a thread Producer de que ela pode prosseguir — ponto em que a Producer retorna ao estado de *espera* e tenta implicitamente readquirir o bloqueio do objeto SynchronizedBuffer. Se o bloqueio estiver disponível, a thread Producer o readquire e o método set continua a executar com a próxima instrução depois da chamada wait. Como wait é chamado em um loop, a condição de continuação do loop é testada novamente para determinar se a thread pode prosseguir. Se não puder, então wait é invocado novamente — caso contrário, o método set continua com a próxima instrução depois do loop.

A linha 21 no método set atribui o value ao buffer. A linha 25 configura occupied como true para indicar que o buffer agora contém um valor (isto é, uma consumidora pode ler o valor, mas a Producer ainda não pode colocar outro valor lá). A linha 27 invoca o método displayState para gerar saída de uma mensagem que indica que Producer está gravando um novo valor no buffer. A linha 29 invoca o método notifyAll (herdado de Object). Se quaisquer threads estiverem esperando o bloqueio de monitor do objeto SynchronizedBuffer, essas threads entram no estado *executável* e podem tentar agora readquirir o bloqueio. O método notifyAll retorna imediatamente, e o método set então retorna ao chamador (isto é, o método run de Producer). Quando o método set retorna, ele libera implicitamente o bloqueio de monitor do objeto SynchronizedBuffer.

O método get e a thread Consumer

Os métodos get e set são implementados de maneira semelhante. Quando o método run da thread Consumer invoca o método synchronized get, a thread tenta adquirir o bloqueio de monitor do objeto SynchronizedBuffer. Se o bloqueio estiver disponível, a thread Consumer o adquire. Então o loop while nas linhas 36–42 determina se occupied é false. Se estiver, o buffer estará vazio, então a linha 39 gera saída de uma mensagem indicando que a thread está tentando ler um valor e a linha 40 invoca o método displayState para gerar saída de uma mensagem indicando que o buffer está vazio e que a thread Consumer está esperando. A linha 41 invoca o método wait para colocar a thread que chamou o método get (isto é, a Consumer) no estado de *espera* para o objeto SynchronizedBuffer. Novamente, a chamada a wait faz com que a thread de chamada libere implicitamente o bloqueio do objeto SynchronizedBuffer, então outra thread pode tentar adquirir o bloqueio SynchronizedBuffer do objeto e invocar o método set ou get do objeto. Se o bloqueio no SynchronizedBuffer não estiver disponível (por exemplo, se a Producer ainda não retornou do método set), a Consumer é bloqueada até que o bloqueio se torne disponível.

A thread Consumer permanece no estado de *espera* até que seja notificada por outra thread de que ela pode prosseguir — ponto em que a thread Consumer retorna ao estado *executável* e tenta readquirir o bloqueio do objeto SynchronizedBuffer. Se o bloqueio estiver disponível, Consumer readquire-o, e o método get continua executando com a próxima instrução depois de wait. Como wait é chamado em um loop, a condição de continuação do loop é testada novamente para determinar se a thread pode prosseguir com sua execução. Se não estiver, wait é invocado novamente — caso contrário, o método get continua com a próxima instrução depois do loop. A linha 46 configura occupied como false para indicar que agora o buffer está vazio (isto é, uma Consumer não pode ler o valor, mas uma Producer pode colocar outro valor no buffer), a linha 48 chama o método displayState para indicar que a consumidora está lendo e a linha 50 invoca o método notifyAll. Se quaisquer threads estiverem no estado de *espera* pelo bloqueio nesse objeto SynchronizedBuffer, elas entrarão no estado *executável* e poderão tentar agora readquirir o bloqueio. O método notifyAll retorna imediatamente, então o método get retorna o valor de buffer ao seu chamador. Quando o método get retornar, o bloqueio no objeto SynchronizedBuffer será implicitamente liberado.

Dica de prevenção de erro 26.3

Sempre invoque o método `wait` em um loop que testa a condição na qual a tarefa está esperando. É possível que uma thread entre novamente no estado executável (via uma thread no estado de espera sincronizada ou outra thread chamando `notifyAll`) antes de a condição ser satisfeita. Testar a condição novamente assegura que a thread não executará de maneira errada se tiver sido notificada anteriormente.

Testando a classe SynchronizedBuffer

A classe `SharedBufferTest2` (Figura 26.19) é semelhante à `SharedBufferTest` (Figura 26.15). `SharedBufferTest2` contém o método `main` (linhas 8–24), que dispara o aplicativo. A linha 11 cria um `ExecutorService` para executar as tarefas `Producer` e `Consumer`. A linha 14 cria um objeto `SynchronizedBuffer` e atribui sua referência à variável `Buffer` `sharedLocation`. Esse objeto armazena os dados que serão compartilhados entre `Producer` e `Consumer`. As linhas 16–17 exibem os títulos de coluna na saída. As linhas 20–21 executam uma thread `Producer` e um `Consumer`. Por fim, a linha 23 chama o método `shutdown` para encerrar o aplicativo quando `Producer` e `Consumer` completarem suas tarefas. Quando o método `main` conclui (linha 24), a thread principal de execução termina.

```
1  // Figura 26.19: SharedBufferTest2.java
2  // Duas threads que manipulam corretamente um buffer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // cria um newCachedThreadPool
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // cria SynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       // executa as tarefas Producer e Consumer
20       application.execute( new Producer( sharedLocation ) );
21       application.execute( new Consumer( sharedLocation ) );
22
23       application.shutdown();
24    } // fim de main
25 } // fim da classe SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------
Consumer tries to read.
Buffer empty. Consumer waits.           -1              false

Producer writes 1                       1               true

Consumer reads 1                        1               false

Consumer tries to read.
Buffer empty. Consumer waits.           1               false

Producer writes 2                       2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false
```

```
Producer writes 4                       4               true

Producer tries to write.
Buffer full. Producer waits.            4               true

Consumer reads 4                        4               false

Producer writes 5                       5               true

Consumer reads 5                        5               false

Producer writes 6                       6               true

Producer tries to write.
Buffer full. Producer waits.            6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Producer tries to write.
Buffer full. Producer waits.            7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Consumer tries to read.
Buffer empty. Consumer waits.           8               false

Producer writes 9                       9               true

Consumer reads 9                        9               false

Consumer tries to read.
Buffer empty. Consumer waits.           9              false

Producer writes 10                      10              true

Consumer reads 10                       10              false

Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Figura 26.19 | Duas threads que manipulam corretamente um buffer sincronizado.

Estude as saídas na Figura 26.19. Observe que cada inteiro produzido é consumido exatamente uma vez — nenhum valor é perdido e nenhum valor é consumido mais de uma vez. A sincronização assegura que `Producer` produz um valor apenas quando o buffer estiver vazio e que `Consumer` consome apenas quando o buffer estiver cheio. A `Producer` sempre vai primeiro, a `Consumer` espera se a `Producer` não tiver produzido desde que a `Consumer` foi consumida pela última vez e a `Producer` espera se a `Consumer` ainda não tiver consumido o valor que a `Producer` produziu mais recentemente. Execute esse programa várias vezes para confirmar que todo inteiro produzido é consumido exatamente uma vez. Na saída de exemplo, observe as linhas que indicam quando a `Producer` e a `Consumer` devem esperar para realizar suas respectivas tarefas.

26.9 Relacionamento de produtor/consumidor: buffers limitados

O programa na Seção 26.8 utiliza sincronização de thread para garantir que duas threads manipulam corretamente os dados em um buffer compartilhado. Entretanto, o aplicativo não pode apresentar um ótimo desempenho. Se as duas threads operarem em diferentes velocidades, uma delas gastará mais (ou a maior parte) de seu tempo na espera. Por exemplo, no programa na Seção 26.8 compartilhamos uma única variável de inteiro entre as duas threads. Se a thread `Producer` produzir valores mais rápido do que a `Consumer` possa consumi-los,

então a thread `Producer` esperará a `Consumer`, porque não haverá nenhuma outra posição no buffer em que colocar o próximo valor. De modo semelhante, se `Consumer` consome valores mais rápido do que `Producer` os produz, `Consumer` espera até `Producer` colocar o próximo valor no buffer compartilhado. Mesmo quando temos threads que operam nas mesmas velocidades relativas, essas threads podem ficar ocasionalmente "fora de sincronia" por um período de tempo, fazendo com que uma delas espere a outra. *Não podemos fazer suposições a respeito das velocidades relativas de threads concorrentes* — as interações que ocorrem com o sistema operacional, a rede, o usuário e outros componentes podem fazer com que as threads operem em diferentes velocidades. Quando isso acontece, as threads esperam. Quando as threads esperam excessivamente, os programas tornam-se menos eficientes, os programas interativos tornam-se menos responsivos e os aplicativos sofrem retardos mais longos.

Buffers limitados

Para minimizar a quantidade de tempo de espera por threads que compartilham recursos e operam nas mesmas velocidades médias, podemos implementar um **buffer delimitado** que fornece um número fixo de células de buffer em que a `Producer` pode colocar valores e a partir do qual a `Consumer` pode recuperar esses valores. (De fato, já fizemos isso com a classe `ArrayBlockingQueue` na Seção 26.7.) Se a `Producer` produzir temporariamente valores mais rápido do que a `Consumer` pode consumi-los, a `Producer` pode escrever valores adicionais nas células de buffer extras, se algum estiver disponível). Essa capacidade permite à `Producer` realizar sua tarefa mesmo que a `Consumer` não esteja pronta para receber o valor atual que está sendo produzido. De maneira semelhante, se a `Consumer` consumir mais rápido do que a capacidade da `Producer` de produzir novos valores, a `Consumer` pode ler valores adicionais (se houver algum) do buffer. Isso permite que a `Consumer` se mantenha ocupada mesmo que a `Producer` não esteja pronta para produzir valores adicionais.

Mesmo um buffer limitado é inadequado se as threads `Producer` e `Consumer` operarem consistentemente em velocidades diferentes. Se `Consumer` sempre executar mais rápido do que `Producer`, então um buffer contendo uma única localização será suficiente. As posições adicionais simplesmente desperdiçariam memória. Se a `Producer` sempre executar mais rápido, somente um buffer com um número "infinito" de posições seria capaz de absorver a produção extra. Contudo, se a thread `Producer` e `Consumer` executarem mais ou menos na mesma velocidade média, um buffer limitado ajuda a atenuar os efeitos de qualquer aceleração ou lentidão ocasional na execução de qualquer thread.

A chave para utilizar um buffer delimitado com uma `Producer` e uma `Consumer` que operam quase na mesma velocidade é fornecer ao buffer posições suficientes para tratar a produção "extra" antecipada. Se, em um período de tempo, determinarmos que a `Producer` produz frequentemente até mais três valores do que a `Consumer` pode consumir, podemos fornecer um buffer de pelo menos três células para tratar a produção extra. Criar um buffer pequeno demais faria as threads esperarem por mais tempo; criar um buffer muito grande desperdiçaria memória.

Dica de desempenho 26.3

Mesmo ao utilizar um buffer delimitado, é possível que uma thread produtora pudesse preencher o buffer, o que forçaria a produtora esperar até que uma consumidora consumisse um valor para liberar um elemento no buffer. De maneira semelhante, se o buffer estiver vazio em qualquer dado momento, uma thread consumidora deve esperar até que a produtora produza outro valor. A chave para utilizar um buffer delimitado é otimizar o tamanho do buffer para minimizar a quantidade de tempo de espera da thread, sem desperdiçar espaço.

Buffers limitados com `ArrayBlockingQueue`

O modo mais simples de implementar um buffer limitado é utilizar um `ArrayBlockingQueue` para o buffer para que *todos os detalhes de sincronização sejam tratados por você*. Isso pode ser feito reutilizando o exemplo da Seção 26.7 e simplesmente passando o tamanho desejado do buffer limitado no construtor `ArrayBlockingQueue`. Antes de repetir o nosso exemplo `ArrayBlockingQueue` anterior com um tamanho diferente, apresentamos um exemplo que ilustra como você mesmo pode construir um buffer limitado. Novamente, observe que utilizar um `ArrayBlockingQueue` resultará em um código com maior capacidade de manutenção e de execução. No Exercício 26.11, solicitamos a reimplementação do exemplo desta seção, utilizando as técnicas de API de concorrência do Java apresentadas na Seção 26.10.

Implementando seu próprio buffer limitado como um buffer circular

O programa nas figuras 26.20–26.21 demonstra uma `Producer` e uma `Consumer` acessando um buffer delimitado com sincronização. Novamente, reutilizamos a interface `Buffer` e as classes `Producer` e `Consumer` do exemplo na Seção 26.6, exceto que a linha 28 é removida da classe `Producer` e classe `Consumer`. Implementamos o buffer limitado na classe `CircularBuffer` (Figura 26.20) como um **buffer circular** que utiliza um array compartilhado de três elementos. Um buffer circular faz gravações e leituras nos elementos do array na ordem, começando na primeira célula e movendo-se para a última. Quando uma thread `Producer` ou `Consumer` alcança o último elemento, ela retorna ao primeiro e começa a gravar ou ler, respectivamente, a partir daí. Nessa versão do relacionamento produtor/consumidor, a `Consumer` consome um valor somente quando o array não estiver vazio e a `Producer` produz um valor somente quando o array não estiver cheio. As instruções que criaram e iniciaram os objetos de thread no método `main` da classe `SharedBufferTest2` (Figura 26.19) agora aparecem na classe `CircularBufferTest` (Figura 26.21).

```
1  // Figura 26.20: CircularBuffer.java
2  // Sincronizando acesso a um buffer limitado de três elementos compartilhado.
3  public class CircularBuffer implements Buffer
4  {
5     private final int[] buffer = { -1, -1, -1 }; // buffer compartilhado
6
7     private int occupiedCells = 0; // conta número de buffers utilizados
8     private int writeIndex = 0; // índice do próximo elemento em que gravar
9     private int readIndex = 0; // índice do próximo elemento a ler
10
11    // coloca o valor no buffer
12    public synchronized void set( int value ) throws InterruptedException
13    {
14       // espera até o buffer ter espaço disponível, então grava o valor;
15       // enquanto não houver posições vazias, põe a thread no estado bloqueado
16       while ( occupiedCells == buffer.length )
17       {
18          System.out.printf( “Buffer is full. Producer waits.\n” );
19          wait(); // espera até uma célula do buffer ser liberada
20       } // fim do while
21
22       buffer[ writeIndex ] = value; // configura novo valor de buffer
23
24       // atualiza índice de gravação circular
25       writeIndex = ( writeIndex + 1 ) % buffer.length;
26
27       ++occupiedCells; // mais uma célula do buffer está cheia
28       displayState( "Producer writes " + value );
29       notifyAll(); // notifica threads que estão esperando para ler a partir do buffer
30    } // fim do método set
31
32    // retorna valor do buffer
33    public synchronized int get() throws InterruptedException
34    {
35       // espera até que o buffer tenha dados, então lê o valor;
36       // enquanto os dados não são lidos, coloca thread em estado de espera
37       while ( occupiedCells == 0 )
38       {
39          System.out.printf( "Buffer is empty. Consumer waits.\n" );
40          wait(); // espera até que uma célula do buffer seja preenchida
41       } // fim do while
42
43       int readValue = buffer[ readIndex ]; // lê valor do buffer
44
45       // atualiza índice de leitura circular
46       readIndex = ( readIndex + 1 ) % buffer.length;
47
48       --occupiedCells; // algumas poucas células de buffers estão ocupadas
49       displayState( "Consumer reads " + readValue );
50       notifyAll(); // notifica threads que estão esperando para gravar no buffer
51
52       return readValue;
53    } // fim do método get
54
55    // exibe a operação atual e o estado de buffer
56    public void displayState( String operation )
57    {
58       // gera saída de operação e número de células de buffers ocupados
59       System.out.printf( "%s%s%d)\n%s", operation,
60          " (buffer cells occupied: ", occupiedCells, "buffer cells:  " );
61
62       for ( int value : buffer )
63          System.out.printf( " %2d  ", value ); // gera a saída dos valores no buffer
64
65       System.out.print( "\n               " );
66
67       for ( int i = 0; i < buffer.length; i++ )
68          System.out.print( "---- " );
```

```
69
70          System.out.print( "\n               " );
71
72          for ( int i = 0; i < buffer.length; i++ )
73          {
74             if ( i == writeIndex && i == readIndex )
75                System.out.print( " WR" ); // índice de gravação e leitura
76             else if ( i == writeIndex )
77                System.out.print( " W   " ); // só grava índice
78             else if ( i == readIndex )
79                System.out.print( "  R  " ); // só lê índice
80             else
81                System.out.print( "     " ); // nenhum dos índices
82          } // for final
83
84          System.out.println( "\n" );
85       } // fim do método displayState
86    } // fim da classe CircularBuffer
```

Figura 26.20 | Sincronizando acesso a um buffer limitado de três elementos compartilhado.

A linha 5 inicializa o array `buffer` como um array `int` de três elementos que representa o buffer circular. A variável `occupiedCells` (linha 7) conta o número de elementos no `buffer` que contêm dados a serem lidos. Quando `occupiedBuffers` for `0`, não haverá dados no buffer circular e `Consumer` deverá esperar — quando `occupiedCells` for `3` (o tamanho do buffer circular), o buffer circular estará cheio e `Producer` deverá esperar. A variável `writeIndex` (linha 8) indica a próxima posição em que um valor pode ser colocado por uma `Producer`. A variável `readIndex` (linha 9) indica a posição a partir da qual o próximo valor pode ser lido por um método `Consumer`.

Método `CircularBuffer` `set`

O método `CircularBuffer` `set` (linhas 12–30) realiza as mesmas tarefas da Figura 26.18, com algumas modificações. O loop `while` nas linhas 16–20 determina se a `Producer` precisa esperar (isto é, todas as células do buffer estão cheias). Se precisar, a linha 18 indica que a `Producer` está esperando para realizar sua tarefa. Em seguida, a linha 19 invoca o método `wait`, fazendo a thread `Producer` lançar o bloqueio `CircularBuffer` e esperar até que haja espaço de um novo valor a ser gravado no buffer. Quando a execução continuar na linha 22 depois do loop `while`, o valor gravado pela `Producer` será colocado no buffer circular na posição `writeIndex`. Depois, a linha 25 atualiza `writeIndex` para a chamada seguinte para o método `CircularBuffer` `set`. Essa linha é a chave para a "circularidade" do buffer. Quando `writeIndex` é incrementado depois do fim do buffer, a linha o configura como `0`. A linha 27 incrementa `occupiedCells`, porque há agora mais um novo valor no buffer que `Consumer` pode ler. Em seguida, a linha 28 invoca o método `displayState` (linhas 56–85) para atualizar a saída com o valor produzido, o número de células de buffer ocupadas, o conteúdo das células de buffer e os `writeIndex` e `readIndex` atuais. A linha 29 invoca o método `notifyAll` para fazer a transição de threads *em espera* para o estado executável, de modo que a thread `Consumer` em espera (se houver alguma) possa agora tentar novamente ler um valor do buffer.

Método `CircularBuffer` `get`

O método `CircularBuffer` `get` (linhas 33–53) também realiza as mesmas tarefas que ele realizou na Figura 26.18, com algumas pequenas modificações. O loop `while` nas linhas 37–41 determina se a `Consumer` deve esperar (isto é, todas as células do buffer estão vazias). Se a `Consumer` precisar esperar, a linha 39 atualizará a saída para indicar que a `Consumer` está esperando para realizar sua tarefa. Em seguida, a linha 40 invoca o método `wait`, fazendo a thread atual lançar o bloqueio no `CircularBuffer` e esperar até que os dados estejam disponíveis para leitura. Quando a execução por fim continuar na linha 43 depois de uma chamada `notifyAll` de `Producer`, `readValue` receberá o valor na localização `readIndex` do buffer circular. Depois, a linha 46 atualiza `readIndex` para a chamada seguinte para o método `CircularBuffer` `get`. Essa linha e a linha 25 implementam a "circularidade" do buffer. A linha 48 decrementa `occupiedCells`, porque agora há mais uma posição aberta no buffer em que a thread `Producer` pode colocar um valor. A linha 49 invoca o método `displayState` para atualizar a saída com o valor consumido, o número de células do buffer ocupadas, o conteúdo das células do buffer e o `writeIndex` e `readIndex` atuais. A linha 50 invoca o método `notifyAll` para permitir quaisquer threads `Producer` em espera gravar no objeto `CircularBuffer` para tentar gravar novamente. Então a linha 52 retorna o valor consumido ao chamador.

Método `CircularBuffer` `displayState`

O método `displayState` (linhas 56–85) gera saída do estado do aplicativo. As linhas 62–63 geram saída dos valores das células de buffer. A linha 63 utiliza o método `printf` com um especificador de formato `"%2d"` para imprimir o conteúdo de cada buffer com um espaço inicial se ele for de um único dígito. As linhas 70–82 geram saída dos `writeIndex` e `readIndex` atuais com as letras `W` e `R`, respectivamente.

Testando a classe `CircularBuffer`

A classe `CircularBufferTest` (Figura 26.21) contém o método `main` que carrega o aplicativo. A linha 11 cria o `ExecutorService`, e a 14 cria um objeto `CircularBuffer` e atribui sua referência à variável `CircularBuffer` `sharedLocation`. A linha 17 invoca método `displayState` do `CircularBuffer` para mostrar o estado inicial do buffer. As linhas 20–21 executam as tarefas `Producer` e `Consumer`. A linha 23 chama o método `shutdown` para encerrar o aplicativo quando as threads completarem as tarefas de `Producer` e `Consumer`.

A cada vez que `Producer` grava um valor ou `Consumer` lê um valor, o programa gera uma mensagem que indica a ação realizada (uma leitura ou uma gravação), o conteúdo de `buffer`, e a localização de `writeIndex` e `readIndex`. Na saída da Figura 26.21, o `Producer` primeiro grava o valor `1`. O buffer então contém o valor `1` na primeira célula e o valor `-1` (o valor padrão que utilizamos com propósitos de saída) nas outras duas células. O índice de gravação é atualizado para a segunda célula, enquanto o índice de leitura permanece na primeira célula. Em seguida, `Consumer` lê `1`. O buffer contém os mesmos valores, mas o índice de leitura foi atualizado na segunda célula. A `Consumer` então tenta ler novamente, mas o buffer está vazio e `Consumer` é forçada a esperar. Observe que só uma vez nessa execução do programa foi necessário que qualquer thread esperasse.

```
1  // Figura 26.21: CircularBufferTest.java
2  // Threads Producer e Consumer que manipulam um buffer circular.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class CircularBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // cria CircularBuffer para armazenar ints
14       CircularBuffer sharedLocation = new CircularBuffer();
15
16       // exibe o estado inicial do CircularBuffer
17       sharedLocation.displayState( "Initial State" );
18
19       // executa as tarefas Producer e Consumer
20       application.execute( new Producer( sharedLocation ) );
21       application.execute( new Consumer( sharedLocation ) );
22
23       application.shutdown();
24    } // fim de main
25 } // fim da classe CircularBufferTest
```

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R   W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R   W
```

```
Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                         WR


Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                  R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                  R        W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
               ---- ---- ----
                       R   W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
               ---- ---- ----
                W     R

Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
               ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                     W     R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
               ---- ---- ----
                         WR

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
               ---- ---- ----
                  R        W

Consumer reads 7 (buffer cells occupied: 1)
buffer cells:    7    8    6
               ---- ---- ----
                       R   W

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
               ---- ---- ----
                W     R
```

```
Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
               ---- ---- ----
                W          R

Consumer reads 9 (buffer cells occupied: 0)
buffer cells:    7    8    9
               ---- ---- ----
                WR

Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10    8    9
               ---- ---- ----
                R    W


Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
               ---- ---- ----
                     WR


Consumer read values totaling: 55
Terminating Consumer
```

Figura 26.21 | Threads `Producer` e `Consumer` que manipulam um buffer circular.

26.10 Relacionamento de produtor/consumidor: as interfaces Lock e Condition

Embora a palavra-chave `synchronized` forneça as necessidades mais básicas de sincronização de threads, o Java fornece outras ferramentas para ajudar a desenvolver programas concorrentes. Nesta seção, discutimos as interfaces `Lock` e `Condition`, introduzidas no Java SE 5. Essas interfaces fornecem controle mais preciso sobre a sincronização de threads, mas são mais complicadas de utilizar.

Interface Lock e classe ReentrantLock

Qualquer objeto pode conter uma referência a um objeto que implementa a interface **`Lock`** (do pacote `java.util.concurrent.locks`). Uma thread chama o método **`lock`** de `Lock` para obter o bloqueio. Uma vez que um `Lock` foi obtido por uma thread, o objeto `Lock` não permitirá que outra thread obtenha o `Lock` até que a primeira thread libere o `Lock` (chamando o método **`unlock`** de `Lock`). Se várias threads estiverem tentando chamar o método `lock` no mesmo objeto `Lock` simultaneamente, apenas uma dessas threads poderá obter o bloqueio — todas as outras serão colocadas no estado de *espera* desse bloqueio. Quando uma thread chama o método `unlock`, o bloqueio do objeto é liberado e uma thread na espera tentando bloquear o objeto prossegue.

A classe **`ReentrantLock`** (do pacote `java.util.concurrent.locks`) é uma implementação básica da interface `Lock`. O construtor de um `ReentrantLock` aceita um argumento `boolean` que especifica se o bloqueio tem uma **diretiva de imparcialidade**. Se o argumento for `true`, a diretiva de imparcialidade do `ReentrantLock` é "a thread em espera mais longa que irá adquirir o bloqueio quando ele estiver disponível". Essa diretiva de imparcialidade garante que o adiamento indefinido (também chamado de inanição) não ocorra. Se o argumento da diretiva de imparcialidade estiver configurado como `false`, não se sabe qual thread na espera irá adquirir o bloqueio quando estiver disponível.

Observação de engenharia de software 26.2

Utilizar um `ReentrantLock` com uma diretiva de imparcialidade evita o adiamento indefinido.

Dica de desempenho 26.4

Utilizar um `ReentrantLock` com uma diretiva de imparcialidade pode reduzir muito o desempenho de programa.

Objetos condição e interface `Condition`

Se uma thread que possui um `Lock` determina que não é possível continuar sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em um **objeto condição**. Utilizar objetos `Lock` permite declarar explicitamente os objetos condição nos quais uma thread pode precisar esperar. Por exemplo, no relacionamento produtor/consumidor, os produtores podem esperar em um único objeto e os consumidores podem esperar em outro. Isso não é possível utilizando palavras-chave `synchronized` e o bloqueio de monitor predefinido de um objeto. Os objetos condição estão associados com um `Lock` específico e são criados chamando o método **`newCondition`** de um `Lock`, que retorna um objeto que implementa a interface **`Condition`** (do pacote `java.util.concurrent.locks`). Para esperar um objeto condição, a thread pode chamar o método **`await`** de `Condition`. Isso libera imediatamente o `Lock` associado e coloca a thread no estado de *espera* dessa `Condition`. Outras threads podem então tentar obter o `Lock`. Quando uma thread *executável* completar uma tarefa e determinar que a thread na *espera* agora pode continuar, a thread *executável* pode chamar o método `Condition` **`signal`** para permitir que uma thread no estado de *espera* dessa `Condition` retorne ao estado *executável*. Nesse ponto, a thread que fez a transição do estado de *espera* para o estado *executável* pode tentar readquirir o `Lock`. Mesmo se for capaz de readquirir o `Lock`, a thread talvez ainda não possa ser capaz de realizar sua tarefa nesse momento — nesse caso, a thread pode chamar o método `await` de `Condition` para liberar o `Lock` e entrar novamente no estado de *espera*. Se múltiplas threads estiverem no estado de *espera* de uma `Condition` quando `signal` for chamado, a implementação padrão de `Condition` sinaliza a thread de espera mais longa para fazer a transição para o estado *executável*. Se uma thread chamar método `Condition` **`signalAll`**, então todas as threads que esperam essa condição fazem a transição para o estado *executável* e tornam-se elegíveis para readquirir o `Lock`. Somente uma dessas threads pode obter o `Lock` no objeto — as outros esperarão até o `Lock` tornar-se novamente disponível. Se o `Lock` tiver uma diretiva de imparcialidade, a thread em espera mais longa adquire o `Lock`. Quando uma thread concluir sua tarefa com um objeto compartilhado, ela deve chamar o método `unlock` para liberar o `Lock`.

Erro comum de programação 26.2

*O **impasse** (deadlock) ocorre quando uma thread em espera (vamos chamá-la de thread1) não pode prosseguir porque está esperando (direta ou indiretamente) outra thread (vamos chamá-la de thread2) prosseguir, enquanto simultaneamente a thread2 não pode prosseguir porque está esperando (direta ou indiretamente) a thread1 prosseguir. As duas threads estão esperando uma à outra, então as ações que permitiriam a cada thread continuar a execução podem jamais ocorrer.*

Dica de prevenção de erro 26.4

Quando múltiplas threads manipulam um objeto compartilhado utilizando bloqueios, assegure que se uma thread chamar o método `await` para entrar no estado de espera por um objeto condição, uma thread separada, por fim, chame o método `Condition` `signal` para fazer a transição da thread em espera pelo objeto condição de volta para o estado executável. Se múltiplas threads podem estar esperando o objeto condição, uma thread separada pode chamar o método `Condition` `signalAll` como uma salvaguarda para assegurar que todas as threads na espera tenham outra oportunidade de realizar suas tarefas. Se isso não for feito, poderia ocorrer inanição.

Erro comum de programação 26.3

Um `IllegalMonitorStateException` ocorre se uma thread executar um `await`, um `signal` ou um `signalAll` em um objeto `Condition` criado a partir de um `ReentrantLock` sem ter adquirido o bloqueio para esses objetos `Condition`.

`Lock` e `Condition` versus palavra-chave `synchronized`

Em alguns aplicativos, utilizar os objetos `Lock` e `Condition` pode ser preferível a utilizar a palavra-chave `synchronized`. `Lock`s permitem que você interrompa threads em espera ou especifique um tempo-limite de espera para adquirir um bloqueio, o que não é possível utilizando a palavra-chave `synchronized`. Além disso, um `Lock` não é forçado a ser adquirido e liberado no mesmo bloco do código, que é o caso com a palavra-chave `synchronized`. Os objetos `Condition` permitem especificar múltiplas condições nas quais as threads podem esperar. Assim, é possível indicar para as threads em espera que um objeto condição específico é verdadeiro agora chamando `signal` ou `signallAll` nesse objeto `Condition`. Com a palavra-chave `synchronized`, não há como afirmar explicitamente a condição que as threads estão esperando, e, portanto, nenhum modo de notificar as threads que esperam de uma condição que elas podem prosseguir sem também sinalizar as threads que esperam qualquer outra condição. Há outras possíveis vantagens de utilizar objetos `Lock` e `Condition`, mas note que, em geral, é melhor usar a palavra-chave `synchronized` a menos que o aplicativo exija capacidades de sincronização avançadas.

Dica de prevenção de erro 26.5

O uso das interfaces `Lock` e `Condition` é propenso a erros — não é garantido que `unlock` será chamado, ao passo que o monitor em uma instrução `synchronized` sempre será liberado quando a instrução completar a execução.

Utilizando Locks e Conditions para implementar sincronização

Para ilustrar como utilizar as interfaces `Lock` e `Condition`, implementamos agora o relacionamento produtor/consumidor utilizando os objetos `Lock` e `Condition` para coordenar o acesso a um buffer de um único elemento compartilhado (figuras 26.22– 26.23). Nesse caso, cada valor produzido é corretamente consumido exatamente uma vez. Novamente, reutilizamos a interface `Buffer` e as classes `Producer` e `Consumer` do exemplo na Seção 26.6, exceto que a linha 28 é removida da classe `Producer` e da classe `Consumer`.

A classe `SynchronizedBuffer` (Figura 26.22) contém cinco campos. A linha 11 cria um novo objeto de tipo `ReentrantLock` e atribui sua referência à variável `Lock accessLock`. O `ReentrantLock` é criado sem a diretiva de imparcialidade porque em qualquer momento apenas uma única `Producer` ou `Consumer` estará esperando para adquirir o `Lock` nesse exemplo. As linhas 14–15 criam duas `Conditions` utilizando o método `Lock newCondition`. `Condition canWrite` contém uma fila para uma thread `Producer` que espera o buffer tornar-se cheio (isto é, há dados no buffer que a `Consumer` ainda não leu). Se o buffer estiver cheio, `Producer` chama o método `await` nessa `Condition`. Quando a `Consumer` lê os dados de um buffer cheio, chama o método `signal` dessa `Condition`. `Condition canRead` contém uma fila para a thread `Consumer` que espera o buffer esvaziar-se (isto é, não há dados no buffer para a `Consumer` ler). Se o buffer estiver vazio, a `Consumer` chama o método `await` dessa `Condition`. Quando `Producer` gravar no buffer vazio, ele chama o método `signal` nessa `Condition`. A variável `int buffer` (linha 17) mantém os dados compartilhados. A variável `boolean occupied` (linha 18) controla se o buffer mantém atualmente os dados (que `Consumer` deve ler).

```
1  // Figura 26.22: SynchronizedBuffer.java
2  // Sincronizando acesso a um número inteiro compartilhado com Lock e Condition
3  // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer
9  {
10    // Bloqueio para controlar sincronização com esse buffer
11    private final Lock accessLock = new ReentrantLock();
12
13    // condições para controlar leitura e gravação
14    private final Condition canWrite = accessLock.newCondition();
15    private final Condition canRead = accessLock.newCondition();
16
17    private int buffer = -1; // compartilhado pelas threads producer e consumer
18    private boolean occupied = false; // se o buffer estiver ocupado
19
20    // coloca o valor int no buffer
21    public void set( int value ) throws InterruptedException
22    {
23       accessLock.lock(); // bloqueia esse objeto
24
25       // envia informações de thread e as informações de buffer para a saída, então espera
26       try
27       {
28          // enquanto o buffer não estiver vazio, coloca thread no estado de espera
29          while ( occupied )
30          {
31             System.out.println( "Producer tries to write." );
32             displayState( "Buffer full. Producer waits." );
33             canWrite.await(); // espera até que o buffer esteja vazio
34          } // fim do while
35
36          buffer = value; // configura novo valor de buffer
37
38          // indica que a produtora não pode armazenar outro valor
39          // até a consumidora recuperar o valor atual de buffer
40          occupied = true;
41
42          displayState( "Producer writes " + buffer );
43
44          // sinaliza a thread que está esperando para ler a partir do buffer
45          canRead.signal();
46       } // fim do try
```

```
47          finally
48          {
49             accessLock.unlock(); // desbloqueia esse objeto
50          } // fim de finally
51       } // fim do método set
52
53       // retorna o valor do buffer
54       public int get() throws InterruptedException
55       {
56          int readValue = 0; // inicializa o valor lido a partir do buffer
57          accessLock.lock(); // bloqueia esse objeto
58
59          // envia informações de thread e as informações de buffer para a saída, então espera
60          try
61          {
62             // se não houver dados para serem lidos, coloca a thread em estado de espera
63             while ( !occupied )
64             {
65                System.out.println( "Consumer tries to read." );
66                displayState( "Buffer empty. Consumer waits." );
67                canRead.await(); // espera até o buffer tornar-se cheio
68             } // fim do while
69
70             // indica que a produtora pode armazenar outro valor
71             // porque a consumidora acabou de recuperar o valor do buffer
72             occupied = false;
73
74             readValue = buffer; // recupera o valor do buffer
75             displayState( "Consumer reads " + readValue );
76
77             // sinaliza a thread que está esperando o buffer tornar-se vazio
78             canWrite.signal();
79          } // fim do try
80          finally
81          {
82             accessLock.unlock(); // desbloqueia esse objeto
83          } // fim de finally
84
85          return readValue;
86       } // fim do método get
87
88       // exibe a operação atual e o estado de buffer
89       public void displayState( String operation )
90       {
91          System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
92             occupied );
93       } // fim do método displayState
94    } // fim da classe SynchronizedBuffer
```

Figura 26.22 | Sincronizando acesso a um número inteiro compartilhado com as interfaces `Lock` e `Condition`.

A linha 23 no método `set` chama o método `lock` sobre o `accessLock` de `SynchronizedBuffer`. Se o bloqueio estiver disponível (isto é, nenhuma outra thread o tiver adquirido), essa thread possuirá o bloqueio e continuará. Se o bloqueio não estiver disponível (isto é, outra thread o segura), o método `lock` espera até o bloqueio ser liberado. Depois de o bloqueio ser adquirido, as linhas 26–46 executam. A linha 29 testa `occupied` para determinar se `buffer` está cheio. Se estiver, as linhas 31–32 exibem uma mensagem que indica que a thread irá esperar. A linha 33 chama o método `Condition await` do objeto condição `canWrite`, que libera temporariamente o `Lock` de `SynchronizedBuffer` e espera um sinal da `Consumer` de que o buffer está disponível para gravação. Quando `buffer` estiver disponível, o método prossegue, gravando no `buffer` (linha 36), configurando `occupied` como `true` (linha 40) e exibindo uma mensagem que indica que o produtor gravou um valor (linha 42). A linha 45 chama o método `Condition signal` do objeto condição `canRead` para notificar a `Consumer` em espera (se houver algum) que o buffer tem novos dados a serem lidos. A linha 49 chama o método `unlock` de um bloco `finally` para liberar o bloqueio e permitir que a `Consumer` prossiga.

Dica de prevenção de erro 26.6

Coloque as chamadas para o método `Lock` `unlock` em um bloco `finally`. Se uma exceção for lançada, o `unlock` ainda deve ser chamado ou o impasse pode ocorrer.

A linha 57 do método `get` (linhas 54–86) chama o método `lock` para adquirir o `Lock`. Esse método espera até o `Lock` estar disponível. Depois que `Lock` é adquirido, a linha 63 testa se `occupied` é `false`, indicando que o buffer está vazio. Nesse caso a linha 67 chama o método `await` no objeto condição `canRead`. Lembre-se de que o método `signal` é chamado na variável `canRead` no método `set` (linha 45). Quando o objeto `Condition` é sinalizado, o método `get` continua. As linhas 72–74 configuram `occupied` como `false`, armazenam o valor de `buffer` em `readValue` e geram saída de `readValue`. Então a linha 78 sinaliza o objeto condição `canWrite`. Isso desperta a `Producer` se ela estiver de fato esperando pelo buffer a ser esvaziado. A linha 82 chama o método `unlock` de um bloco `finally` para liberar o bloqueio, e a linha 85 retorna `readValue` para o chamador.

Erro comum de programação 26.4

Esquecer de sinalizar uma thread em espera é um erro de lógica. A thread permanecerá no estado de espera, o que a impedirá de prosseguir. Tal espera pode levar ao adiamento indefinido ou impasse.

A classe `SharedBufferTest2` (Figura 26.23) é idêntica à da Figura 26.19. Estude as saídas da Figura 26.23. Observe que cada número inteiro produzido é consumido exatamente uma vez — nenhum valor é perdido, e nenhum valor é consumido mais de uma vez. Os objetos `Lock` e `Condition` asseguram que `Producer` e `Consumer` não possam realizar suas tarefas a menos que seja sua vez. A `Producer` deve ir primeiro, a `Consumer` deve esperar se a `Producer` não tiver produzido desde que a `Consumer` foi consumida pela última vez e a `Producer` deve esperar se a `Consumer` ainda não tiver consumido o valor que a `Producer` produziu mais recentemente. Execute esse programa várias vezes para confirmar que todo inteiro produzido é consumido exatamente uma vez. Na saída de exemplo, observe as linhas que indicam quando a `Producer` e a `Consumer` devem esperar para realizar suas respectivas tarefas.

```
1  // Figura 26.23: SharedBufferTest2.java
2  // Duas threads que manipulam um buffer sincronizado.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // cria novo pool de threads com duas threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // cria SynchronizedBuffer para armazenar ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       // executa as tarefas Producer e Consumer
20       application.execute( new Producer( sharedLocation ) );
21       application.execute( new Consumer( sharedLocation ) );
22
23       application.shutdown();
24    } // fim de main
25 } // fim da classe SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------
Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true

Consumer reads 1                        1               false

Producer writes 2                       2               true
```

```
Producer tries to write.
Buffer full. Producer waits.            2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Consumer reads 4                        4               false
Consumer tries to read.
Buffer empty. Consumer waits.           4               false

Producer writes 5                       5               true
Consumer reads 5                        5               false

Consumer tries to read.
Buffer empty. Consumer waits.           5               false

Producer writes 6                       6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Producer writes 9                       9               true

Consumer reads 9                        9               false

Producer writes 10                      10              true


Producer done producing
Terminating Producer
Consumer reads 10                       10              false


Consumer read values totaling 55
Terminating Consumer
```

Figura 26.23 | Duas threads que manipulam um buffer sincronizado.

26.11 Multithreading com GUI

Os aplicativos Swing apresentam uma série excepcional de desafios para a programação de múltiplas threads. Todos os aplicativos Swing têm uma única thread, chamada **thread de despacho de eventos**, para tratar interações com os componentes GUI do aplicativo. As interações típicas são atualizar componentes GUI ou processar ações de usuário, como cliques de mouse. Todas as tarefas que exigem interação com a GUI de um aplicativo são colocadas em uma fila de eventos e executadas em sequência pela thread de despacho de eventos.

Os componentes GUI Swing não são seguros para threads — não podem ser manipulados por múltiplas threads sem o risco de resultados incorretos. Diferentemente de outros exemplos apresentados neste capítulo, a segurança para threads em aplicativos GUI é alcançada não sincronizando as ações de thread, mas, sim, assegurando que os componentes Swing são acessados apenas a partir de uma única thread — a thread de despacho de eventos. Essa técnica é chamada **confinamento de thread**. Permitir que apenas uma thread acesse os objetos "não seguros para threads" elimina a possibilidade de corrupção em virtude do acesso simultâneo a esses objetos por parte de múltiplas threads.

Normalmente, é suficiente realizar cálculos simples na thread de despacho de eventos na sequência com manipulações de componentes GUI. Se um aplicativo deve realizar um cálculo longo em resposta a uma interação de interface com o usuário, a thread de despacho de eventos não pode ocupar-se de outras tarefas na fila de evento enquanto a thread estiver presa a nesse cálculo. Isso faz os componentes GUI tornarem-se indiferentes. É preferível tratar um cálculo demorado em uma thread separada, liberando a thread de despacho de eventos para

continuar o gerenciamento de outras interações de GUI. Obviamente, para atualizar a GUI com base nos resultados do cálculo, você deve atualizar a GUI a partir da thread de despacho de eventos, em vez da thread trabalhadora que realizou o cálculo.

Classe SwingWorker

O Java SE 6 fornece a classe **SwingWorker** (no pacote `javax.swing`) para realizar cálculos demorados em uma thread trabalhadora e atualizar componentes Swing a partir da thread de despacho de eventos com base nos resultados dos cálculos. `SwingWorker` implementa a interface `Runnable`, o que significa que um objeto `SwingWorker` pode ser agendado para executar em uma thread separada. A classe `SwingWorker` fornece vários métodos para simplificar a realização de cálculos em uma thread trabalhadora e disponibilizar os resultados para exibição em uma GUI. Alguns métodos `SwingWorker`s comuns são descritos na Figura 26.24.

Método	Descrição
`doInBackground`	Define um cálculo longo e é chamado em uma thread trabalhadora.
`done`	Executa na thread de despacho de eventos quando `doInBackground` retorna.
`execute`	Agenda o objeto `SwingWorker` a ser executado em uma thread trabalhadora.
`get`	Espera o cálculo ser concluído e retorna o resultado do cálculo (isto é, o valor de retorno de `doInBackground`).
`publish`	Envia resultados intermediários a partir do método `doInBackground` para o método `process` para processamento na thread de despacho de eventos.
`process`	Recebe resultados intermediários a partir do método `publish` e os processa na thread de despacho de eventos.
`setProgress`	Configura a propriedade de progresso para notificar todos os ouvintes de alteração de propriedade na thread de despacho de eventos de atualizações de barra de progresso.

Figura 26.24 | Métodos `SwingWorker` comumente utilizados.

26.11.1 Realizando cálculos em uma thread worker

No próximo exemplo, o usuário insere um número *n* e o programa obtém o *enésimo* número de Fibonacci, que calculamos para utilizar o algoritmo recursivo discutido na Seção 18.4. Como o algoritmo é demorado para valores grandes, utilizamos um objeto `SwingWorker` para realizar o cálculo em uma thread trabalhadora. A GUI também fornece um conjunto separado de componentes que obtêm o próximo número de Fibonacci na sequência a cada clique de um botão, começando com `fibonacci(1)`. Esse conjunto de componentes realiza o breve cálculo diretamente na thread de despacho de eventos. Observe que esse programa é capaz de produzir até o 92º número de Fibonacci — os valores subsequentes estão fora do intervalo que pode ser representado por um `long`. Lembre-se de que você pode utilizar a classe `BigInteger` para representar valores inteiros arbitrariamente grandes.

A classe `BackgroundCalculator` (Figura 26.25) faz o cálculo Fibonacci recursivo em uma thread trabalhadora. Essa classe estende `SwingWorker` (linha 8), sobrescrevendo os métodos `doInBackground` e `done`. O método `doInBackground` (linhas 21–25) calcula o *enésimo* número de Fibonacci em uma thread trabalhadora e retorna o resultado. O método `done` (linhas 28–44) exibe o resultado em um `JLabel`.

```
1  // Figura 26.25: BackgroundCalculator.java
2  // Subclasse SwingWorker para calcular números de Fibonacci
3  // em uma thread de segundo plano.
4  import javax.swing.SwingWorker;
5  import javax.swing.JLabel;
6  import java.util.concurrent.ExecutionException;
7
8  public class BackgroundCalculator extends SwingWorker< Long, Object >
9  {
10    private final int n; // Número de Fibonacci a calcular
11    private final JLabel resultJLabel; // JLabel para exibir o resultado
12
13    // construtor
14    public BackgroundCalculator( int number, JLabel label )
15    {
16       n = number;
17       resultJLabel = label;
18    } // fim do construtor BackgroundCalculator
19
20    // código demorado a ser executado em uma thread trabalhadora
```

```
21       public Long doInBackground()
22       {
23          long nthFib = fibonacci( n );
24          return String.valueOf( nthFib );
25       } // fim do método doInBackground
26
27       // código a executar na thread de despacho de eventos quando doInBackground retorna
28       protected void done()
29       {
30          try
31          {
32             // obtém o resultado de doInBackground e exibe-o
33             resultJLabel.setText( get().toString() );
34          } // fim do try
35          catch ( InterruptedException ex )
36          {
37             resultJLabel.setText( "Interrupted while waiting for results." );
38          } // fim do catch
39          catch ( ExecutionException ex )
40          {
41             resultJLabel.setText(
42                "Error encountered while performing calculation." );
43          } // fim do catch
44       } // fim do método done
45
46       // método recursivo fibonacci; calcula o enésimo número de Fibonacci
47       public long fibonacci( long number )
48       {
49          if ( number == 0 || number == 1 )
50             return number;
51          else
52             return fibonacci( number - 1 ) + fibonacci( number - 2 );
53       } // fim do método fibonacci
54    } // fim da classe BackgroundCalculator
```

Figura 26.25 | Subclasse `SwingWorker` para calcular números de Fibonacci em uma thread em segundo plano.

Observe que `SwingWorker` é uma classe genérica. Na linha 8, o primeiro parâmetro de tipo é `Long` e o segundo é `Object`. O primeiro parâmetro de tipo indica o tipo retornado pelo método `doInBackground`; o segundo indica o tipo que é passado entre os métodos `publish` e `process` para tratar dos resultados intermediários. Como não utilizamos `publish` e `process` nesse exemplo, basta utilizarmos `Object` como o segundo parâmetro de tipo. Discutimos `publish` e `process` na Seção 26.11.2.

Um objeto `BackgroundCalculator` pode ser instanciado a partir de uma classe que controla uma GUI. Um `BackgroundCalculator` mantém variáveis de instância para um número inteiro que representa o número de Fibonacci a ser calculado e um `JLabel` que exibe os resultados do cálculo (linhas 10–11). O construtor `BackgroundCalculator` (linhas 14–18) inicializa essas variáveis de instância com os argumentos que são passados para o construtor.

Observação de engenharia de software 26.3

Todos os componentes GUI que forem manipulados pelos métodos `SwingWorker`, como os componentes que serão atualizados a partir dos métodos `process` ou `done`, devem ser passados para o construtor da subclasse `SwingWorker` e armazenados no objeto de subclasse. Isso dá a esses métodos acesso aos componentes GUI que eles manipularão.

Quando o método `execute` é chamado no objeto `BackgroundCalculator`, o objeto é agendado para a execução em uma thread trabalhadora. O método `doInBackground` é chamado a partir da thread trabalhadora e invoca o método `fibonacci` (linhas 47–53), passando a variável de instância `n` como um argumento (linha 23). O método `fibonacci` utiliza a recursão para computar o Fibonacci de `n`. Quando `fibonacci` retorna, o método `doInBackground` retorna o resultado.

Depois de `doInBackground` retornar, o método `done` é chamado a partir da thread de despacho de eventos. Esse método tenta configurar o resultado `JLabel` com o valor de retorno de `doInBackground` chamando o método `get` para recuperar esse valor de retorno (linha 33). Se necessário, o método `get` espera que o resultado esteja pronto, mas como o chamamos a partir do método `done`, o cálculo estará completo antes de `get` ser chamado. As linhas 35–38 capturam `InterruptedException` se a thread atual for interrompida enquanto espera `get` retornar. Essa exceção não ocorrerá nesse exemplo já que o cálculo já terá sido concluído no momento em que `get` for chamado. As linhas 39–43 capturam `ExecutionException`, que é lançada se ocorrer uma exceção durante o cálculo.

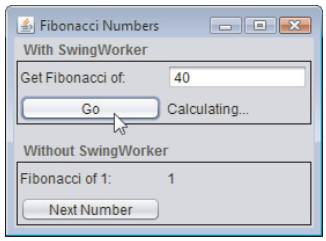
Classe FibonacciNumbers

A classe `FibonacciNumbers` (Figura 26.26) exibe uma janela que contém dois conjuntos de componentes GUI — um configurado para calcular um número de Fibonacci em uma thread trabalhadora e o outro para obter o próximo número de Fibonacci em resposta ao clique de usuário em um `JButton`. O construtor (linhas 38–109) coloca esses componentes em `JPanels` separados com títulos diferentes. As linhas 46–47 e 78–79 adicionam dois `JLabels`, um `JTextField` e um `JButton` ao `workerJPanel` para permitir ao usuário inserir um número inteiro cujo número de Fibonacci será calculado pelo `BackgroundWorker`. As linhas 84–85 e 103 adicionam dois `JLabels` e um `JButton` ao painel de threads de despacho de eventos para permitir ao usuário obter o próximo número de Fibonacci na sequência. As variáveis de instância `n1` e `n2` contêm dois números de Fibonacci anteriores na sequência e são inicializadas como `0` e `1`, respectivamente (linhas 29–30). A variável de instância `count` armazena o número de sequência mais recentemente calculado e é inicializada como `1` (linha 31). Os dois `JLabels` exibem `count` e `n2` inicialmente, para que o usuário veja o texto `Fibonacci of 1: 1` no `eventThreadJPanel` quando a GUI iniciar.

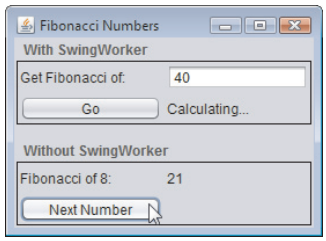
```
1  // Figura 26.26: FibonacciNumbers.java
2  // Utilizando SwingWorker para fazer um cálculo longo com
3  // resultados exibidos em uma GUI.
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10 import javax.swing.JLabel;
11 import javax.swing.JTextField;
12 import javax.swing.border.TitledBorder;
13 import javax.swing.border.LineBorder;
14 import java.awt.Color;
15 import java.util.concurrent.ExecutionException;
16
17 public class FibonacciNumbers extends JFrame
18 {
19    // componentes para calcular o Fibonacci de um número inserido pelo usuário
20    private final JPanel workerJPanel =
21       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
22    private final JTextField numberJTextField = new JTextField();
23    private final JButton goJButton = new JButton( "Go" );
24    private final JLabel fibonacciJLabel = new JLabel();
25
26    // componentes e variáveis para obter o próximo número de Fibonacci
27    private final JPanel eventThreadJPanel =
28       new JPanel( new GridLayout( 2, 2, 5, 5 ) );
29    private long n1 = 0; // inicializa com o primeiro número de Fibonacci
30    private long n2 = 1; // inicializa com o segundo número de Fibonacci
31    private int count = 1; // número de Fibonacci atual a exibir
32    private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
33    private final JLabel nFibonacciJLabel =
34       new JLabel( String.valueOf( n2 ) );
35    private final JButton nextNumberJButton = new JButton( "Next Number" );
36
37    // construtor
38    public FibonacciNumbers()
39    {
40       super( "Fibonacci Numbers" );
41       setLayout( new GridLayout( 2, 1, 10, 10 ) );
42
43       // adiciona componentes GUI ao painel SwingWorker
44       workerJPanel.setBorder( new TitledBorder(
45          new LineBorder( Color.BLACK ), "With SwingWorker" ) );
46       workerJPanel.add( new JLabel( "Get Fibonacci of:" ) );
47       workerJPanel.add( numberJTextField );
48       goJButton.addActionListener(
49          new ActionListener()
50          {
51             public void actionPerformed( ActionEvent event )
52             {
53                int n;
54
```

```
55                try
56                {
57                   // recupera a entrada de usuário como um número inteiro
58                   n = Integer.parseInt( numberJTextField.getText() );
59                } // fim do try
60                catch( NumberFormatException ex )
61                {
62                   // exibe uma mensagem de erro se o usuário não
63                   // inseriu um número inteiro
64                   fibonacciJLabel.setText( "Enter an integer." );
65                   return;
66                } // fim do catch
67
68                // indica que o cálculo iniciou
69                fibonacciJLabel.setText( "Calculating..." );
70
71                // cria uma tarefa para fazer o cálculo em segundo plano
72                BackgroundCalculator task =
73                   new BackgroundCalculator( n, fibonacciJLabel );
74                task.execute(); // executa a tarefa
75             } // fim do método actionPerformed
76          } // fim da classe interna anônima
77       ); // fim da chamada para addActionListener
78       workerJPanel.add( goJButton );
79       workerJPanel.add( fibonacciJLabel );
80
81       // adiciona componentes GUI ao painel da thread de despacho de eventos
82       eventThreadJPanel.setBorder( new TitledBorder(
83          new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
84       eventThreadJPanel.add( nJLabel );
85       eventThreadJPanel.add( nFibonacciJLabel );
86       nextNumberJButton.addActionListener(
87          new ActionListener()
88          {
89             public void actionPerformed( ActionEvent event )
90             {
91                // calcula o número de Fibonacci depois de n2
92                long temp = n1 + n2;
93                n1 = n2;
94                n2 = temp;
95                ++count;
96
97                // exibe o seguinte número de Fibonacci
98                nJLabel.setText( "Fibonacci of " + count + ": " );
99                nFibonacciJLabel.setText( String.valueOf( n2 ) );
100            } // fim do método actionPerformed
101         } // fim da classe interna anônima
102      ); // fim da chamada para addActionListener
103      eventThreadJPanel.add( nextNumberJButton );
104
105      add( workerJPanel );
106      add( eventThreadJPanel );
107      setSize( 275, 200 );
108      setVisible( true );
109   } // fim do construtor
110
111   // método main inicia a execução de programa
112   public static void main( String[] args )
113   {
114      FibonacciNumbers application = new FibonacciNumbers();
115      application.setDefaultCloseOperation( EXIT_ON_CLOSE );
116   } // fim de main
117 } // fim da classe FibonacciNumbers
```

a) Começa a calcular uma sequência Fibonacci de 40 números no segundo plano.



b) Calculando outras sequências de valores Fibonacci enquanto a Fibonacci de 40 números continua calculando.



c) O cálculo de uma sequência Fibonacci de 40 números é finalizado.

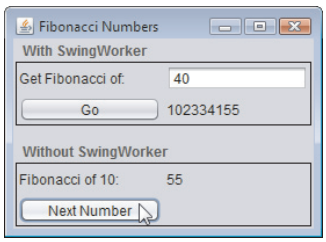


Figura 26.26 | Utilizando `SwingWorker` para fazer um cálculo longo exibindo os resultados em uma GUI.

As linhas 48–77 registram o handler de evento para o `goJButton`. Se o usuário clicar nesse `JButton`, a linha 58 obtém o valor inserido no `numberJTextField` e tenta analisá-lo como um número inteiro. As linhas 72–73 criam um novo objeto `BackgroundCalculator`, passando o valor inserido pelo usuário e o `fibonacciJLabel` que é utilizado para exibir os resultados do cálculo. A linha 74 chama o método `execute` no `BackgroundCalculator`, agendando-o para execução em uma thread trabalhadora separada. O método `execute` não espera `BackgroundCalculator` terminar de executar. Ele retorna imediatamente, permitindo à GUI continuar o processamento de outros eventos enquanto o cálculo é realizado.

Se o usuário clicar em `nextNumberJButton` no `eventThreadJPanel`, o handler de evento registrado nas linhas 86–102 executa. As linhas 92–95 adicionam dois números de Fibonacci anteriores armazenados em `n1` e `n2` para determinar o próximo número na sequência, atualizar `n1` e `n2` como seus novos valores e incrementar `count`. Então, as linhas 98–99 atualizam a GUI para exibir o próximo número. O código desses cálculos está no método `actionPerformed`, portanto eles são realizados na thread de despacho de eventos. O tratamento desses cálculos breves na thread de despacho de eventos não faz a GUI tornar-se irresponsiva, como ocorre com o algoritmo recursivo para calcular o Fibonacci de um número grande. Como o cálculo Fibonacci mais longo é realizado em uma thread trabalhadora separada utilizando o `SwingWorker`, é possível obter o número de Fibonacci seguinte enquanto o cálculo recursivo ainda está em andamento.

26.11.2 Processando resultados intermediários com `SwingWorker`

Apresentamos um exemplo que utiliza a classe `SwingWorker` para executar um processo longo em uma thread de segundo plano e atualizar a GUI quando o processo terminar. Agora apresentamos um exemplo de atualização da GUI com resultados intermediários antes que o processo longo se complete. A Figura 26.27 apresenta a classe `PrimeCalculator`, que estende `SwingWorker` para calcular os *n* primeiros números primos em uma thread trabalhadora. Além dos métodos `doInBackground` e `done` utilizados no exemplo anterior, essa classe utiliza os métodos `SwingWorker` `publish`, `process` e `setProgress`. Nesse exemplo, o método `publish` envia números primos ao método `process` à medida que eles são localizados, o método `process` exibe esses primos em um componente GUI e o método `setProgress` atualiza a propriedade de progresso. Mais adiante mostramos como utilizar essa propriedade para atualizar um `JProgressBar`.

```
1  // Figura 26.27: PrimeCalculator.java
2  // Calcula os n primeiros primos, exibindo-os à medida que são localizados.
3  import javax.swing.JTextArea;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import javax.swing.SwingWorker;
7  import java.util.Random;
8  import java.util.List;
9  import java.util.concurrent.CancellationException;
10 import java.util.concurrent.ExecutionException;
11
12 public class PrimeCalculator extends SwingWorker< Integer, Integer >
13 {
14    private final Random generator = new Random();
```

```
15       private final JTextArea intermediateJTextArea; // exibe os primos localizados
16       private final JButton getPrimesJButton;
17       private final JButton cancelJButton;
18       private final JLabel statusJLabel; // exibe o status do cálculo
19       private final boolean[] primes; // array booleano para localizar primos
20
21       // construtor
22       public PrimeCalculator( int max, JTextArea intermediate, JLabel status,
23          JButton getPrimes, JButton cancel )
24       {
25          intermediateJTextArea = intermediate;
26          statusJLabel = status;
27          getPrimesJButton = getPrimes;
28          cancelJButton = cancel;
29          primes = new boolean[ max ];
30
31          // inicializa todos os valores de array primos como verdadeiros
32          for ( int i = 0; i < max; i ++ )
33             primes[ i ] = true;
34       } // fim do construtor
35
36       // localiza todos os primos até o máximo utilizando o Crivo de Eratóstenes
37       public Integer doInBackground()
38       {
39          int count = 0; // o número de primos localizados
40
41          // iniciando no terceiro valor, circula pelo array e coloca
42          // falso como o valor de qualquer número maior que for um múltiplo
43          for ( int i = 2; i < primes.length; i++ )
44          {
45             if ( isCancelled() ) // se o cálculo tiver sido cancelado
46                return count;
47             else
48             {
49                setProgress( 100 * ( i + 1 ) / primes.length );
50
51                try
52                {
53                   Thread.sleep( generator.nextInt( 5 ) );
54                } // fim do try
55                catch ( InterruptedException ex )
56                {
57                   statusJLabel.setText( "Worker thread interrupted" );
58                   return count;
59                } // fim do catch
60
61                if ( primes[ i ] ) // i é primo
62                {
63                   publish( i ); // disponibiliza para exibição na lista de primos
64                   ++count;
65
66                   for ( int j = i + i; j < primes.length; j += i )
67                      primes[ j ] = false; // i não é primo
68                } // fim do if
69             } // fim de else
70          } // for final
71
72          return count;
73       } // fim do método doInBackground
74
75       // exibe valores publicados na lista de primos
76       protected void process( List< Integer > publishedVals )
77       {
78          for ( int i = 0; i < publishedVals.size(); i++ )
79             intermediateJTextArea.append( publishedVals.get( i ) + "\n" );
80       } // fim do método process
81
82       // código para executar quando doInBackground se completa
83       protected void done()
```

```
84      {
85         getPrimesJButton.setEnabled( true ); // ativa o botão Get Primes
86         cancelJButton.setEnabled( false ); // desativa o botão Cancel
87
88         int numPrimes;
89
90         try
91         {
92            numPrimes = get(); // recupera o valor de retorno de doInBackground
93         } // fim do try
94         catch ( InterruptedException ex )
95         {
96            statusJLabel.setText( "Interrupted while waiting for results." );
97            return;
98         } // fim do catch
99         catch ( ExecutionException ex )
100        {
101           statusJLabel.setText( "Error performing computation." );
102           return;
103        } // fim do catch
104        catch ( CancellationException ex )
105        {
106           statusJLabel.setText( "Cancelled." );
107           return;
108        } // fim do catch
109
110        statusJLabel.setText( "Found " + numPrimes + " primes." );
111     } // fim do método done
112  } // fim da classe PrimeCalculator
```

Figura 26.27 | Calcula os *n* primeiros primos, exibindo-os à medida que são localizados.

A classe `PrimeCalculator` estende `SwingWorker` (linha 12), com o primeiro parâmetro de tipo que indica o tipo de retorno do método `doInBackground` e o segundo que indica que o tipo de resultados intermediários passados entre os métodos `publish` e `process`. Nesse caso, ambos os parâmetros de tipo são `Integer`s. O construtor (linhas 22–34) aceita como argumentos um número inteiro que indica o limite superior dos números primos a localizar, uma `JTextArea` utilizada para exibir primos na GUI, um `JButton` para iniciar um cálculo e um para cancelá-lo, e um `JLabel` utilizado para exibir o status do cálculo.

Crivo de Eratóstenes

As linhas 32–33 inicializam os elementos do array `boolean` `primes` como `true`. `PrimeCalculator` utiliza esse array e o algoritmo de **Crivo de Eratóstenes** (descrito no Exercício 7.27) para localizar todos os primos menores que `max`. O Crivo de Eratóstenes aceita uma lista de números inteiros e, iniciando com o primeiro número primo, filtra todos os múltiplos desse primo. Ele então avança para o próximo primo, que será o próximo número que ainda não foi filtrado, e elimina todos seus múltiplos. Ele continua até que o fim da lista seja alcançado e todos os não primos tenham sido filtrados. Algoritmicamente, iniciamos com o elemento `2` do array `boolean` e configuramos as células correspondentes a todos os valores que são múltiplos de 2 como `false` para indicar que eles são divisíveis por 2 e, portanto, não primos. Então avançamos para o próximo elemento do array, verificamos se é `true` e, se for, configuramos todos os seus múltiplos como `false` para indicar que são divisíveis pelo índice atual. Quando o array de inteiros tiver sido percorrido desse modo, todos os índices que contêm `true` serão primos, uma vez que não têm divisores.

Método doInBackground

No método `doInBackground` (linhas 37–73), a variável de controle `i` do loop (linhas 43–70) controla o índice atual para implementar o Crivo de Eratóstenes. A linha 45 chama o método `SwingWorker` **`isCancelled`** herdado para determinar se o usuário clicou no botão Cancel. Se `isCancelled` retornar `true`, o método `doInBackground` retornará o número de primos localizados até esse momento (linha 46) sem terminar o cálculo.

Se o cálculo não for cancelado, a linha 49 chamará `setProgress` para atualizar a porcentagem do array que foi percorrida até agora. A linha 53 coloca a thread em execução atualmente para dormir até 4 milissegundos. Discutimos a razão disso a seguir. A linha 61 testa se o elemento do array `primes` no índice atual é `true` (e, portanto, primo). Nesse caso, a linha 63 passa o índice para o método `publish` para que ele possa ser exibido como um resultado intermediário na GUI e a linha 64 incrementa o número de primos localizados. As linhas 66–67 configuram todos os múltiplos do índice atual como `false` para indicar que eles não são primos. Quando o array de inteiros tiver sido percorrido, a linha 72 retorna o número de primos localizados.

Método process

As linhas 76–80 declaram o método `process`, que executa na thread de despacho de eventos e recebe seu argumento `publishedVals` do método `publish`. A passagem de valores entre `publish` na thread trabalhadora e `process` na thread de despacho de eventos é assíncrona; `process` pode não ser invocado por toda chamada a `publish`. Todos `Integers` publicados desde a última chamada a `process` são recebidos como uma `List` pelo método `process`. As linhas 78–79 iteram por essa lista e exibem os valores publicados em uma `JTextArea`. Como o cálculo no método `doInBackground` avança rapidamente, publicando valores frequentemente, as atualizações para `JTextArea` podem acumular-se na thread de despacho de eventos, tornando a GUI lenta. De fato, ao procurar por um grande número de primos, a thread de despacho de eventos pode receber tantas solicitações em rápida sucessão para atualizar a `JTextArea` que ela fica sem memória em sua fila de eventos. É por isso que colocamos a thread trabalhadora para dormir por alguns milissegundos entre as chamadas a `publish`. A velocidade do cálculo é reduzida o suficiente para permitir que a thread de despacho de eventos acompanhe as solicitações de atualização da `JTextArea` com os novos primos, permitindo à GUI atualizar-se facilmente e permanecer responsiva.

Método done

As linhas 83–111 definem o método `done`. Quando o cálculo é concluído ou cancelado, o método `done` ativa o botão **Get Primes** e desativa o botão **Cancel** (linhas 85–86). A linha 92 obtém o valor de retorno — número de primos localizados — a partir do método `doInBackground`. As linhas 94–108 capturam as exceções lançadas pelo método `get` e exibem uma mensagem apropriada no `statusJLabel`. Se não ocorrer exceções, a linha 110 configura o `statusJLabel` para indicar o número de primos localizados.

Classe FindPrimes

A classe `FindPrimes` (Figura 26.28) exibe um `JTextField` que permite ao usuário inserir um número, um `JButton` para começar a localizar todos os primos menores que aquele número e uma `JTextArea` para exibir os primos. Um `JButton` permite que o usuário cancele o cálculo, e um `JProgressBar` indica o progresso do cálculo. O construtor `FindPrimes` (linhas 32–125) inicializa esses componentes e os exibe em um `JFrame` usando `BorderLayout`.

As linhas 42–94 registram o handler de evento para o `getPrimesJButton`. Quando o usuário clica nesse `JButton`, as linhas 47–49 reinicializam a `JProgressBar` e limpam a `displayPrimesJTextArea` e a `statusJLabel`. As linhas 53–63 analisam o valor no `JTextField` e exibem uma mensagem de erro se o valor não for um número inteiro. As linhas 66–68 constroem um novo objeto `PrimeCalculator`, passando como argumentos o número inteiro inserido pelo usuário, a `displayPrimesJTextArea` para exibir os primos, o `statusJLabel` e os dois `JButtons`.

```
1  // Figura 26.28: FindPrimes.java
2  // Utilizando um SwingWorker para exibir números primos e atualizar um JProgressBar
3  // enquanto os números primos estão sendo calculados.
4  import javax.swing.JFrame;
5  import javax.swing.JTextField;
6  import javax.swing.JTextArea;
7  import javax.swing.JButton;
8  import javax.swing.JProgressBar;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JScrollPane;
12 import javax.swing.ScrollPaneConstants;
13 import java.awt.BorderLayout;
14 import java.awt.GridLayout;
15 import java.awt.event.ActionListener;
16 import java.awt.event.ActionEvent;
17 import java.util.concurrent.ExecutionException;
18 import java.beans.PropertyChangeListener;
19 import java.beans.PropertyChangeEvent;
20
21 public class FindPrimes extends JFrame
22 {
23    private final JTextField highestPrimeJTextField = new JTextField();
24    private final JButton getPrimesJButton = new JButton( "Get Primes" );
25    private final JTextArea displayPrimesJTextArea = new JTextArea();
26    private final JButton cancelJButton = new JButton( "Cancel" );
27    private final JProgressBar progressJProgressBar = new JProgressBar();
28    private final JLabel statusJLabel = new JLabel();
29    private PrimeCalculator calculator;
30
31    // construtor
32    public FindPrimes()
33    {
```

```
34          super( "Finding Primes with SwingWorker" );
35          setLayout( new BorderLayout() );
36
37          // inicializa o painel para obter um número do usuário
38          JPanel northJPanel = new JPanel();
39          northJPanel.add( new JLabel( "Find primes less than: " ) );
40          highestPrimeJTextField.setColumns( 5 );
41          northJPanel.add( highestPrimeJTextField );
42          getPrimesJButton.addActionListener(
43             new ActionListener()
44             {
45                public void actionPerformed( ActionEvent e )
46                {
47                   progressJProgressBar.setValue( 0 ); // redefine JProgressBar
48                   displayPrimesJTextArea.setText( "" ); // limpa JTextArea
49                   statusJLabel.setText( "" ); // limpa JLabel
50
51                   int number; // pesquisa primos para cima até esse valor
52
53                   try
54                   {
55                      // obtém entrada de usuário
56                      number = Integer.parseInt(
57                         highestPrimeJTextField.getText() );
58                   } // fim do try
59                   catch ( NumberFormatException ex )
60                   {
61                      statusJLabel.setText( "Enter an integer." );
62                      return;
63                   } // fim do catch
64
65                   // constrói um novo objeto PrimeCalculator
66                   calculator = new PrimeCalculator( number,
67                      displayPrimesJTextArea, statusJLabel, getPrimesJButton,
68                      cancelJButton );
69
70                   // ouve alterações de propriedade na barra de progresso
71                   calculator.addPropertyChangeListener(
72                      new PropertyChangeListener()
73                      {
74                         public void propertyChange( PropertyChangeEvent e )
75                         {
76                            // se a propriedade alterada for progress,
77                            // atualiza a barra de progresso
78                            if ( e.getPropertyName().equals( “progress” ) )
79                            {
80                               int newValue = ( Integer ) e.getNewValue();
81                               progressJProgressBar.setValue( newValue );
82                            } // fim do if
83                         } // fim do método propertyChange
84                      } // fim da classe interna anônima
85                   ); // termina a chamada para addPropertyChangeListener
86
87                   // desativa o botão Get Primes e ativa o botão Cancel
88                   getPrimesJButton.setEnabled( false );
89                   cancelJButton.setEnabled( true );
90
91                   calculator.execute(); // executa o objeto PrimeCalculator
92                } // fim do método ActionPerformed
93             } // fim da classe interna anônima
94          ); // fim da chamada para addActionListener
95          northJPanel.add( getPrimesJButton );
96
97          // adiciona uma JList rolável para exibir os resultados do cálculo
98          displayPrimesJTextArea.setEditable( false );
99          add( new JScrollPane( displayPrimesJTextArea,
100            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
101            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );
102
```

```
103         // inicializa um painel para exibir cancelJButton,
104         // progressJProgressBar e statusJLabel
105         JPanel southJPanel = new JPanel( new GridLayout( 1, 3, 10, 10 ) );
106         cancelJButton.setEnabled( false );
107         cancelJButton.addActionListener(
108            new ActionListener()
109            {
110               public void actionPerformed( ActionEvent e )
111               {
112                  calculator.cancel( true ); // cancela o cálculo
113               } // fim do método ActionPerformed
114            } // fim da classe interna anônima
115         ); // fim da chamada para addActionListener
116         southJPanel.add( cancelJButton );
117         progressJProgressBar.setStringPainted( true );
118         southJPanel.add( progressJProgressBar );
119         southJPanel.add( statusJLabel );
120
121         add( northJPanel, BorderLayout.NORTH );
122         add( southJPanel, BorderLayout.SOUTH );
123         setSize( 350, 300 );
124         setVisible( true );
125      } // fim do construtor
126
127      // método main inicia a execução de programa
128      public static void main( String[] args )
129      {
130         FindPrimes application = new FindPrimes();
131         application.setDefaultCloseOperation( EXIT_ON_CLOSE );
132      } // fim de main
133   } // fim da classe FindPrimes
```

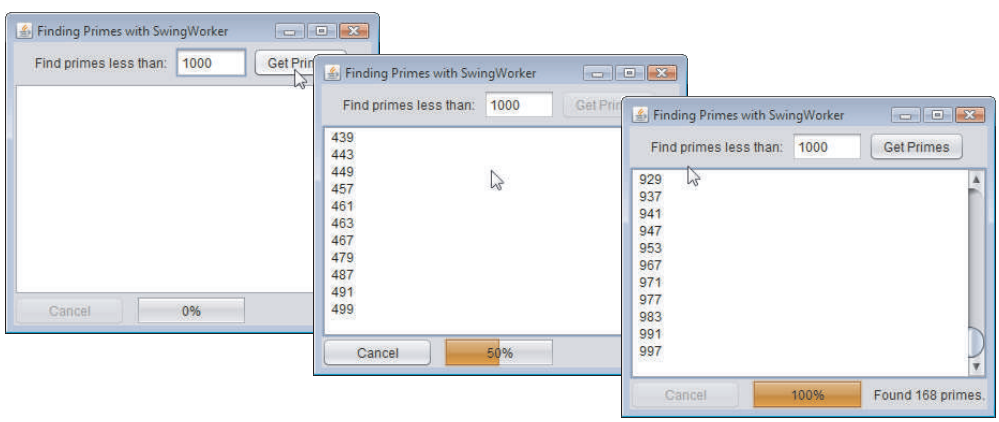


Figura 26.28 | Utilizando um `SwingWorker` para exibir números primos e atualizar uma `JProgressBar` enquanto os números primos estão sendo calculados.

As linhas 71–85 registram um objeto `PropertyChangeListener` para o `PrimeCalculator`. **`PropertyChangeListener`** é uma interface do pacote `java.beans` que define um único método, `propertyChange`. Toda vez que o método `setProgress` é invocado em um `PrimeCalculator`, este gera um `PropertyChangeEvent` para indicar que a propriedade de progresso mudou. O método `PropertyChange` ouve esses eventos. A linha 78 testa se um dado `PropertyChangeEvent` indica uma alteração na propriedade de progresso. Nesse caso, a linha 80 obtém o novo valor da propriedade e a linha 81 atualiza a `JProgressBar` com o novo valor da propriedade de progresso.

O `JButton` **Get Primes** é desativado (linha 88) para que somente um cálculo que atualiza a GUI possa executar por vez, e o `JButton` **Cancel** é ativado (linha 89) para permitir ao usuário parar o cálculo antes de ele ser completado. A linha 91 executa o `PrimesCalculator` para começar a localizar os primos. Se o usuário clicar no `cancelJButton`, o handler de evento registrado nas linhas 107–115 chama o método `PrimeCalculator` **`cancel`** (linha 112), que é herdado da classe `SwingWorker`, e o cálculo retorna mais cedo. O argumento `true` para o método `cancel` indica que a thread que realiza a tarefa deve ser interrompida em uma tentativa de cancelar a tarefa.

26.12 Interfaces Callable e Future

A interface Runnable fornece apenas as funcionalidades mais básicas para a programação de múltiplas threads. De fato, essa interface tem várias limitações. Suponha que um Runnable encontre um problema e tente lançar uma exceção verificada. O método run não é declarado para lançar nenhuma exceção, desse modo, o problema deve ser tratado dentro do Runnable — a exceção não pode ser passada para a thread chamadora. Agora suponha que um Runnable esteja realizando um cálculo longo e o aplicativo queira recuperar o resultado desse cálculo. O método run não pode retornar um valor, então o aplicativo deve utilizar dados compartilhados para passar o valor de volta para a thread chamadora. Isso também envolve o overhead de sincronizar acesso aos dados. Os desenvolvedores das APIs de concorrência introduzidas no Java SE 5 reconheceram essas limitações e criaram uma nova interface para corrigi-las. A interface **Callable** (do pacote java.util.concurrent) declara um único método chamado **call**. Essa interface é projetada para ser semelhante à interface Runnable — permitindo que uma ação seja realizada concorrentemente em uma thread separada — mas o método call permite que a thread retorne um valor ou lance uma exceção verificada.

Um aplicativo que cria uma Callable provavelmente quer executá-la concorrentemente com outras Runnables e Callables. A interface ExecutorService fornece o método **submit**, que executará uma Callable passada como seu argumento. O método submit retorna um objeto do tipo **Future** (do pacote java.util.concurrent), que é uma interface que representa a Callable em execução. A interface Future declara o método **get** para retornar o resultado da Callable e fornece outros métodos para gerenciar a execução de uma Callable.

26.13 Conclusão

Neste capítulo, você aprendeu que a concorrência tem sido historicamente implementada com primitivos de sistema operacional disponíveis apenas a programadores de sistemas experientes, mas que o Java a disponibiliza para você por meio da linguagem e das APIs. Também aprendeu que a própria JVM cria threads para executar um programa, e que também pode criar threads para realizar tarefas de limpeza, como coleta de lixo.

Discutimos o ciclo de vida de uma thread e os estados que uma thread pode ocupar durante seu tempo de vida. Também discutimos prioridades de threads do Java, que ajudam o sistema a agendar threads para execução. Você aprendeu que deve evitar manipular prioridades de thread do Java diretamente e conheceu os problemas associados com prioridades de thread, como o adiamento indefinido (às vezes chamado de inanição).

Em seguida, apresentamos a interface Runnable, utilizada para especificar uma tarefa que pode executar concorrentemente com outras tarefas. O método run dessa interface é invocado pela thread que executa a tarefa. Mostramos como executar um objeto Runnable associando-o com um objeto da classe Thread. Depois mostramos como utilizar a interface Executor para gerenciar a execução de objetos Runnable via grupos de threads, que podem reutilizar threads existentes para eliminar o overhead de criar uma nova thread para cada tarefa e que podem aprimorar o desempenho otimizando o número de threads para assegurar que o processador permaneça ocupado.

Você aprendeu que, quando múltiplas threads compartilham um objeto e uma ou várias delas modificam esse objeto, podem ocorrer resultados indeterminados a menos que o acesso ao objeto compartilhado seja gerenciado corretamente. Mostramos como resolver esse problema via sincronização de threads, que coordena o acesso a dados compartilhados por múltiplas threads concorrentes. Também aprendeu várias técnicas para realizar a sincronização — primeiro com a classe predefinida ArrayBlockingQueue (que trata todos os detalhes de sincronização para você), então com monitores predefinidos do Java e a palavra-chave synchronized e, por fim, com as interfaces Lock e Condition.

Discutimos o fato de as GUI Swing não serem seguras para thread, portanto todas as interações com e as modificações ao GUI devem ser realizadas na thread de despacho de eventos. Também discutimos os problemas associados com a realização de cálculos muito extensos na thread de despacho de eventos. Mostramos então como é possível utilizar a classe SwingWorker do Java SE 6 para realizar cálculos muito longos em threads trabalhadoras. Você aprendeu a exibir os resultados de um SwingWorker em uma GUI quando o cálculo for completado e a exibir resultados intermediários enquanto o cálculo ainda está em andamento.

Por fim, discutimos as interfaces Callable e Future, que permitem executar tarefas que retornam resultados e obter esses resultados, respectivamente. Utilizamos as técnicas de multithreading introduzidas aqui novamente no Capítulo 27, "Redes", para ajudar a construir servidores de múltiplas threads que podem interagir com múltiplos clientes concorrentemente.

Resumo

Seção 26.1 Introdução

- Historicamente, a concorrência foi implementada com os primitivos de sistemas operacionais disponíveis apenas para programadores de sistemas experientes.
- A linguagem de programação Ada tornou os primitivos de concorrência amplamente disponíveis.
- O Java disponibiliza a concorrência por meio da linguagem e das APIs.
- A JVM cria threads para executar um programa e para tarefas de faxina, a exemplo da coleta de lixo.

Seção 26.2 Estados de thread: ciclo de vida de uma thread

- Uma nova thread inicia seu ciclo de vida no estado *novo*. Quando o programa inicia a thread, ela é colocada no estado *executável*. Considera-se que uma thread no estado *executável* está executando sua tarefa.
- Uma thread *executável* entra no estado de *espera* aguardando outra thread realizar uma tarefa. Uma thread de *espera* transita para o estado *executável* quando outra thread a notifica para continuar em execução.
- Uma thread *executável* pode entrar no estado de *espera sincronizada* por um intervalo de tempo especificado, retornando ao estado *executável* quando esse intervalo expirar ou quando o evento pelo qual ela estiver esperando ocorrer.
- Uma thread *executável* pode transitar para o estado de *espera sincronizada* se fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa. Essa thread voltará ao estado *executável* quando for notificada por outra thread ou quando o intervalo sincronizado expirar.
- Uma thread *adormecida* permanece no estado de *espera sincronizada* por um período de tempo designado, depois do qual ela retorna ao estado *executável*.
- Uma thread *adormecida* transita para o estado *bloqueado* quando tenta realizar uma tarefa que não pode ser completada imediatamente e a thread deve esperar temporariamente até essa tarefa ser concluída. Nesse ponto, a thread *bloqueada* transita para o estado *executável*, portanto pode retomar a execução.
- Uma thread *executável* entra no estado *terminado* quando completa sua tarefa ou, caso contrário, termina (talvez por causa de um erro).
- No nível de sistema operacional, o estado *executável* geralmente inclui dois estados separados. Quando uma thread entra pela primeira vez no estado *executável* a partir do estado *novo*, ela está no estado *pronto*. Uma thread *pronta* entra no estado *executável* quando o sistema operacional a despacha.
- A maioria dos sistemas operacionais aloca um quantum ou fração de tempo no qual uma thread realiza sua tarefa. Quando isso expira, a thread retorna ao estado *pronto* e outra thread é atribuída ao processador.
- O agendamento de threads determina qual thread despachar com base nas prioridades de thread.

Seção 26.3 Prioridades de thread e agendamento de thread

- Toda thread do Java tem uma prioridade de thread (de `MIN_PRIORITY` a `MAX_PRIORITY`) que ajuda o sistema operacional a determinar a ordem em que as threads serão agendadas.
- Por padrão, toda thread recebe a prioridade `NORM_PRIORITY` (uma constante de 5). Cada nova thread herda a prioridade da thread que a cria.
- O trabalho de um scheduler de thread do sistema operacional é determinar a próxima thread que entrará em execução.
- Quando uma thread de prioridade mais alta entra no estado *pronto*, o sistema operacional geralmente faz preempção da thread atualmente *em execução* (uma operação conhecida como agendamento preemptivo).
- Dependendo do sistema operacional, as threads de prioridade mais alta poderiam adiar — possivelmente por um tempo indefinido — a execução de threads de prioridade mais baixa.

Seção 26.4 Criando e executando threads

- Um objeto `Runnable` representa uma "tarefa" que pode executar concorrentemente com outras tarefas.
- A interface `Runnable` declara o método `run` no qual você coloca o código que define a tarefa a ser realizada. A thread que executa um `Runnable` chama o método `run` para realizar a tarefa.
- Um programa não terminará até que sua última thread complete a execução.
- Você não pode prever a ordem em que as threads serão agendadas, mesmo se você souber a ordem na qual elas foram criadas e iniciadas.
- Recomenda-se o uso da interface `Executor` para gerenciar a execução de objetos `Runnable`. Em geral, um objeto `Executor` cria e gerencia um grupo de threads — chamado pool de threads.
- Os `Executors` podem reutilizar threads existentes (para eliminar o overhead de criar novas threads) e podem aprimorar o desempenho otimizando o número de threads para assegurar que o processador permaneça ocupado.
- O método `Executor` `execute` recebe um `Runnable` e o atribui a uma thread disponível em um grupo de threads. Se não houver nenhuma, o `Executor` cria uma nova thread ou espera uma se tornar disponível.
- A interface `ExecutorService` (do pacote `java.util.concurrent`) estende a interface `Executor` e declara outros métodos para gerenciar o ciclo de vida de um `Executor`.
- Um objeto que implementa a interface `ExecutorService` pode ser criado com os métodos `static` declarados na classe `Executors` (do pacote `java.util.concurrent`).
- O método `Executors` `newCachedThreadPool` retorna um `ExecutorService` que cria novas threads conforme a necessidade do aplicativo.
- O método `ExecutorService` `execute` executa o seu `Runnable` algum momento no futuro. O método retorna imediatamente de cada invocação — o programa não espera cada tarefa para terminar.
- O método `ExecutorService` `shutdown` notifica o `ExecutorService` para deixar de aceitar novas tarefas, mas continua executando tarefas existentes e termina quando essas tarefas terminam sua execução.

Seção 26.5 Sincronização de thread

- A sincronização de threads coordena o acesso a dados compartilhados por múltiplas threads concorrentes.
- Sincronizando as threads, você pode assegurar que cada thread que acessa um objeto compartilhado exclui todas as outras threads de fazer isso simultaneamente — isso é chamado exclusão mútua.
- Uma maneira comum de realizar a sincronização é utilizar os monitores predefinidos do Java. Cada objeto tem um monitor e um bloqueio de monitor. O monitor assegura que o bloqueio de monitor do seu objeto é mantido por no máximo uma thread em qualquer dado momento, e assim pode ser utilizado para impor a exclusão mútua.
- Se uma operação exigir que a thread em execução mantenha um bloqueio enquanto a operação é realizada, uma thread deve adquirir o bloqueio antes de poder prosseguir com a operação. Quaisquer outras threads tentando prosseguir com uma operação que requer o mesmo bloqueio será *bloqueada* até que a primeira thread libere o bloqueio, ponto em que as threads *bloqueadas* podem tentar adquirir o bloqueio.
- Para especificar que uma thread deve manter um bloqueio de monitor para executar um bloco do código, o código deve ser colocado em uma instrução `synchronized`. Diz-se que esse código deve ser guardado pelo bloqueio de monitor.
- As instruções `synchronized` são declaradas utilizando a palavra-chave `synchronized`:

```
synchronized ( objeto )
{
   instruções
} // fim da instrução synchronized
```

  onde *objeto* é o objeto cujo bloqueio de monitor será adquirido; *objeto* é normalmente `this` se for o objeto no qual a instrução `synchronized` aparece.
- O Java também permite métodos `synchronized`. Antes de executar, um método `synchronized` não `static` deve adquirir o bloqueio no objeto que é utilizado para chamar o método. De modo semelhante, um método `synchronized static` deve adquirir o bloqueio na classe que é utilizada para chamar o método.
- O método `ExecutorService` `awaitTermination` impõe que um programa espere pelo término das threads. Ele retorna o controle à sua chamadora quando todas as tarefas executando no `ExecutorService` concluíram ou quando o tempo-limite especificado expirou. Se todas as tarefas forem concluídas antes de o tempo-limite expirar, o método retorna `true`; caso contrário, ele retorna `false`.
- Você pode simular a atomicidade assegurando que apenas uma thread realize o conjunto de operações por vez. A atomicidade pode ser alcançada com instruções `synchronized` ou métodos `synchronized`.
- Ao compartilhar dados imutáveis entre threads, você deve declarar os campos de dados `final` correspondentes para indicar que os valores das variáveis não mudarão depois que forem inicializados.

Seção 26.6 Relacionamento entre produtor e consumidor sem sincronização

- Em um relacionamento produtor/consumidor de múltiplas threads, uma thread produtora gera dados e os coloca em um objeto compartilhado chamado buffer. Uma thread consumidora lê dados do buffer.
- As operações em dados de um buffer compartilhado por um produtor e um consumidor devem prosseguir somente se o buffer estiver no estado correto. Se o buffer não estiver cheio, o produtor pode produzir; se o buffer não for vazio, o consumidor pode consumir. Se o buffer estiver cheio quando o produtor tentar gravar nele, o produtor deve esperar até que haja espaço. Se o buffer estiver vazio ou o valor anterior já tiver sido lido, o consumidor deve esperar os novos dados se tornarem disponíveis.

Seção 26.7 Relacionamento de produtor/consumidor: `ArrayBlockingQueue`

- `ArrayBlockingQueue` é uma classe de buffer inteiramente implementada a partir do pacote `java.util.concurrent` que implementa a interface `BlockingQueue`.
- Um `ArrayBlockingQueue` pode implementar um buffer compartilhado em um relacionamento de produtor/consumidor. O método `put` coloca um elemento no fim da `BlockingQueue`, esperando se a fila estiver cheia. O método `take` removerá um elemento da cabeça da `BlockingQueue`, esperando se a fila estiver vazia.
- `ArrayBlockingQueue` armazena dados compartilhados em um array que é dimensionado com um argumento passado para o construtor. Uma vez criado, um `ArrayBlockingQueue` tem o tamanho fixado.

Seção 26.8 Relacionamento entre produtor/consumidor com sincronização

- Você pode implementar um buffer compartilhado utiliza a palavra-chave `synchronized` e os métodos `Object` `wait`, `notify` e `notifyAll`.
- Uma thread pode chamar o método `Object` `wait` para lançar o bloqueio de monitor de um objeto, e esperar no estado de *espera* enquanto outras threads tentam inserir a(s) instrução(ões) ou método(s) `synchronized` do objeto.
- Quando uma thread que executa uma instrução (ou método) `synchronized` completa ou satisfaz a condição que outra thread pode estar esperando, ela pode chamar o método `Object` `notify` para permitir que uma thread em espera transite para o estado *executável* novamente. Nesse ponto, a thread que transitou do estado de *espera* para o estado *bloqueado* pode tentar readquirir o bloqueio de monitor do objeto.
- Se uma thread chamar `notifyAll`, então todas as threads que esperam o bloqueio de monitor se tornarão elegíveis para readquirir o bloqueio (isto é, todas farão a transição para o estado *executável*).

Seção 26.9 Relacionamento de produtor/consumidor: buffers limitados

- Você não pode fazer suposições sobre as velocidades relativas das threads concorrentes.
- Um buffer limitado pode ser utilizado para minimizar o tempo de espera de threads que compartilham recursos e operam nas mesmas velocidades médias. Se a produtora produzir valores temporariamente mais rápidos do que a consumidora pode consumi-los, a produtora pode escrever valores adicionais no espaço extra de buffer (se algum estiver disponível). Se a consumidora consumir mais rápido do que a capacidade da produtora de produzir novos valores, a consumidora poderá ler valores adicionais (se houver algum) do buffer.
- A chave para utilizar um buffer delimitado com uma produtora e uma consumidora que operam quase na mesma velocidade é fornecer ao buffer posições suficientes para tratar a produção "extra" antecipada.
- O modo mais simples de implementar um buffer limitado é utilizar um `ArrayBlockingQueue` para o buffer para que todos os detalhes de sincronização sejam tratados por você.

Seção 26.10 Relacionamento de produtor/consumidor: as interfaces `Lock` e `Condition`

- As interfaces `Lock` e `Condition`, introduzidas no Java SE 5, fornecem aos programadores controle mais preciso sobre a sincronização de threads, mas são mais complicadas de utilizar.
- Qualquer objeto pode conter uma referência a um objeto que implementa a interface `Lock` (do pacote `java.util.concurrent.locks`). Uma thread chama o método `lock` de `Lock` para obter o bloqueio. Uma vez que um `Lock` foi obtido por uma thread, o objeto `Lock` não permitirá que outra thread obtenha o `Lock` até que a primeira thread libere o `Lock` (chamando o método `unlock` de `Lock`).
- Se várias threads estiverem tentando chamar o método `lock` no mesmo objeto `Lock` ao mesmo tempo, apenas uma thread poderá obter o bloqueio — as outras serão colocadas no estado de *espera*. Quando uma thread chama `unlock`, o bloqueio do objeto é liberado e uma thread em espera tentando bloquear o objeto prossegue.
- A classe `ReentrantLock` é uma implementação básica da interface `Lock`.
- O construtor `ReentrantLock` aceita um `boolean` que especifica se o bloqueio tem uma diretiva de imparcialidade. Se `true`, a diretiva de imparcialidade `ReentrantLock` é "a thread em espera mais longa irá adquirir o bloqueio quando ele estiver disponível" — isso impede o adiamento indefinido. Se o argumento estiver configurado como `false`, não é garantido que a thread na espera irá adquirir o bloqueio quando ele estiver disponível.
- Se uma thread que possui um `Lock` determina que não é possível continuar sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em um objeto condição. Utilizar objetos `Lock` permite declarar explicitamente os objetos condição nos quais uma thread pode precisar esperar.
- Os objetos condição estão associados a um `Lock` específico e são criados chamando o método `Lock` `newCondition`, que retorna um objeto `Condition`. Para esperar em uma `Condition`, a thread pode chamar o método de espera da `Condition`. Isso libera imediatamente o `Lock` associado e coloca a thread no estado de *espera* dessa `Condition`. Outras threads podem então tentar obter o `Lock`.
- Quando uma thread *executável* completar uma tarefa e determinar que a thread na *espera* agora pode continuar, a thread *executável* pode chamar o método `Condition` `signal` para permitir que uma thread no estado de *espera* dessa `Condition` retorne ao estado *executável*. Nesse ponto, a thread que fez a transição do estado de *espera* para o estado *executável* pode tentar readquirir o `Lock`.
- Se múltiplas threads estiverem no estado de *espera* de uma `Condition` quando `signal` for chamado, a implementação padrão de `Condition` sinaliza a thread de espera mais longa para fazer a transição para o estado *executável*.
- Se uma thread chamar método `Condition` `signalAll`, então todas as threads que esperam essa condição mudam para o estado *executável* e tornam-se elegíveis para readquirir o `Lock`.
- Quando uma thread concluir sua tarefa com um objeto compartilhado, ela deve chamar o método `unlock` para liberar o `Lock`.
- Os `Lock`s permitem interromper threads em espera ou especificar um tempo-limite a esperar a fim de adquirir um bloqueio — o que é não possível fazer com `synchronized`. Além disso, um objeto `Lock` não é obrigado a ser adquirido e liberado no mesmo bloco do código, que é o caso com a palavra-chave `synchronized`.
- Objetos `Condition` permitem especificar múltiplas condições nas quais a thread podem esperar. Assim, é possível indicar as threads em espera que um objeto condição específico é agora verdadeiro, chamando os métodos `signal` ou `signallAll` desse objeto `Condition`. Com `synchronized`, não há nenhum modo de afirmar explicitamente a condição que as threads estão esperando.

Seção 26.11 Multithreading com GUI

- A thread de despacho de eventos trata interações com os componentes GUI do aplicativo. Todas as tarefas que interagem com a GUI são colocadas em uma fila de eventos e executadas em sequência por essa thread.
- Os componentes GUI Swing não são seguros para thread. A segurança de thread é alcançada assegurando que os componentes Swing são acessados apenas a partir da thread de despacho de eventos.
- Realizar um cálculo longo em resposta a uma interação da interface com usuário amarra a thread de despacho de eventos, impedindo-a de atender a outras tarefas e tornando os componentes GUI não responsivos. Os cálculos demorados devem ser tratados em threads separadas.
- Você pode estender a classe genérica `SwingWorker` (pacote `javax.swing`), que implementa `Runnable`, para realizar cálculos demorados em uma thread trabalhadora e atualizar componentes Swing da thread de despacho de eventos com base nos resultados dos cálculos. Você sobrescreve seus métodos `doInBackground` e `done`. O método `doInBackground` realiza o cálculo e retorna o resultado. O método `done` exibe os resultados na GUI.

- O primeiro parâmetro de tipo da classe `SwingWorker` indica o tipo retornado pelo método `doInBackground`; o segundo indica o tipo que é passado entre os métodos `publish` e `process` para tratar dos resultados intermediários.
- O método `doInBackground` é chamado a partir de uma thread trabalhadora. Depois de `doInBackground` retornar, o método `done` é chamado a partir da thread de despacho de eventos para exibir os resultados.
- Um `ExecutionException` é lançada se ocorrer uma exceção durante o cálculo.
- O método `SwingWorker` `publish` repetidamente envia resultados intermediários ao método `process`, que exibe os resultados em um componente de GUI. O método `setProgress` atualiza a propriedade de progresso.
- O método `process` executa na thread de despacho de eventos e recebe dados do método `publish`. A passagem de valores entre `publish` na thread trabalhadora e `process` na thread de despacho de eventos é assíncrona; `process` não é necessariamente invocado para cada chamada feita a `publish`.
- `PropertyChangeListener` é uma interface do pacote `java.beans` que define um único método, `PropertyChange`. Toda vez que o método `setProgress` é invocado, uma `PropertyChangeEvent` é gerada para indicar que a propriedade de progresso mudou.

Seção 26.12 Interfaces `Callable` e `Future`

- A interface `Callable` (do pacote `java.util.concurrent`) declara um método único chamado `call`. A interface é semelhante a `Runnable` — permitindo que uma ação seja realizada concorrentemente em uma thread separada — mas `call` permite que a thread retorne um valor ou lance uma exceção verificada.
- O método `ExecutorService` `submit` executa um `Callable` passado como seu argumento.
- O método `submit` retorna um objeto do tipo `Future` (do pacote `java.util.concurrent`) que representa `Callable` em execução. A interface `Future` declara o método `get` para retornar o resultado de `Callable` e fornece outros métodos para gerenciar a execução de `Callable`.

Terminologia

adiamento indefinido, 808
adquirir o bloqueio, 812
agendamento de rodízio, 807
agendamento preemptivo, 808
agendando threads, 806
`await`, método da interface `Condition`, 836
`awaitTermination`, método da interface `ExecutorService`, 814
`BlockingQueue`, interface, 823
bloqueio de monitor, 812
bloqueio intrínseco, 812
buffer, 818
buffer circular, 830
buffer limitado, 830
`call`, método da interface `Callable`, 851
`Callable`, interface, 851
`cancel`, método da classe `SwingWorker`, 850
condição, objeto, 836
`Condition`, interface, 836
confinamento de thread, 840
consumidor, 818
Crivo de Eratóstenes, 847
dados mutáveis, 817
dependente de estado, 818
despachar uma thread, 806
diretiva de imparcialidade de um bloqueio, 835
espera sincronizada, estado, 806
estado *bloqueado*, 806
estado de *espera*, 806
estado de *execução*, 806
estado *executável*, 805
estado *morto*, 806
estado *novo*, 805
estado *pronto*, 806
estado *terminado*, 806
exclusão mútua, 812
`execute`, método do `Executor` interface, 810
`Executor`, interface, 810
`Executors`, classe, 810
`ExecutorService`, interface, 810
fila de prioridade de múltiplos níveis, 808
fração de tempo, 806
`Future`, interface, 851
`get`, método da interface `Future`, 851
guardando código com um bloqueio, 812
`IllegalMonitorStateException`, classe, 825
impasse, 836
inanição, 808
`interrupt`, método da classe `Thread`, 809
`InterruptedException`, classe, 809
intervalo de sono, 806
`isCancelled`, método da classe `SwingWorker`, 847
`Lock`, interface, 835
`lock`, método da interface `Lock`, 835
monitor, 812
multithreading, 804
`newCachedThreadPool`, método da classe `Executors`, 810
`newCondition`, método da interface `Lock`, 836
`notify`, método da classe `Object`, 825
`notifyAll`, método da classe `Object`, 825
operação atômica, 816
operações concorrentes, 804
operações paralelas, 804
pool de threads, 810
prioridade de thread, 807
produtor, 818
produtor/consumidor, relacionamento, 818
programação concorrente, 805
`PropertyChangeListener`, interface, 850
`put`, método da interface `BlockingQueue`, 823
quantum, 806
`ReentrantLock`, classe, 835
`run`, método da interface `Runnable`, 808
`Runnable`, interface, 808
scheduler de thread, 807
segurança de thread, 815
`shutdown`, método da classe `ExecutorService`, 811
`signal`, método da interface `Condition`, 836
`signalAll`, método da interface `Condition`, 836
sincronizar threads, 805
`size`, método da classe `ArrayBlockingQueue`, 824
`sleep`, método da classe `Thread`, 809
spooling de impressão, 818
`submit`, método da classe `ExecutorService`, 851
`SwingWorker`, classe, 841
`synchronized`, instrução, 812
`synchronized`, método, 812
`synchronized`, palavra-chave, 812
`take`, método da classe `BlockingQueue`, 823
thread, agendamento, 806
`Thread`, classe, 807
thread, sincronização, 812
thread adormecida, 806
thread consumidor, 818
thread de despacho de evento (*event-dispatch thread* — EDT), 840
thread principal, 810
thread produtor, 818
threads de execução, 804
`unlock`, método da interface `Lock`, 835
valor obsoleto, 815
`wait`, método da classe `Object`, 825

Exercícios de autorrevisão

26.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) C e C++ são linguagens de _____ enquanto o Java é uma linguagem de _____.

b) Uma thread entra no estado *terminado* quando _____.

c) Para pausar um número designado de milissegundos e retomar a execução, uma thread deve chamar o método _____ da classe _____.

d) O método _____ da classe `Condition` move uma única thread no estado de *espera* de um objeto para o estado *executável*.

e) O método _____ da classe `Condition` move toda thread no estado de *espera* de um objeto para o estado *executável*.

f) Uma thread _____ entra no estado _____ quando ela completa sua tarefa ou, de outro modo, termina.

g) Uma thread *executável* pode entrar no estado _____ por um intervalo especificado de tempo.

h) No nível do sistema operacional, o estado *executável* realmente inclui dois estados separados: _____ e _____.

i) `Runnable`s são executadas utilizando uma classe que implementa a interface _____.

j) O método `ExecutorService` _____ termina cada thread em um `ExecutorService` logo que termina de executar sua `Runnable` atual, se houver alguma.

k) Uma thread pode chamar o método _____ em um objeto `Condition` para liberar o `Lock` associado e colocar essa thread no estado _____.

l) Em um relacionamento_____, o(a) _____ gera dados e os armazena em um objeto compartilhado, e o(a) _____ lê os dados do objeto compartilhado.

m) A classe _____ implementa a interface `BlockingQueue` que utiliza um array.

n) A palavra-chave _____ indica que somente uma thread por vez deve executar em um objeto.

26.2 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) Uma thread não é *executável* se tiver terminado.

b) Alguns sistemas operacionais utilizam fracionamento de tempo com threads. Portanto, eles podem permitir os threads fazer preempção de threads da mesma prioridade.

c) Quando o quantum da thread expira, a thread retorna ao estado de *execução* enquanto o sistema operacional a atribui a um processador.

d) Em um sistema de um único processador sem fracionamento de tempo, cada thread em um conjunto de threads de igual prioridade (sem outras threads presentes) executa até a conclusão antes de outras threads de igual prioridade obterem uma chance de executar.

Respostas dos exercícios de autorrevisão

26.1 a) uma única thread, múltiplas threads. b) seu método `run` é encerrado. c) `sleep`, `Thread`. d) `signal`. e) `signalAll`. f) *executável*, *terminado*. g) *espera sincronizada*. h) *pronto*, *em execução*. i) `Executor`. j) `shutdown`. k) `await`, *em espera*. l) produtor/consumidor, produtora, consumidora. m) `ArrayBlockingQueue`. n) `synchronized`.

26.2 a) Verdadeira b) Falsa. Fracionamento de tempo permite uma thread executar até que sua fração de tempo (ou quantum) expire. Então outras threads de igual prioridade podem executar. c) Falsa. Quando o quantum de uma thread expira, a thread retorna ao estado *pronto* e o sistema operacional atribui ao processador outra thread. d) Verdadeira.

Exercícios

26.3 Determine se cada uma das seguintes alternativas é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

a) O método `sleep` não consome tempo de processador enquanto uma thread dorme.

b) Declarar um método `synchronized` garante que o impasse não ocorra.

c) Uma vez que um `ReentrantLock` foi obtido por uma thread, o objeto `ReentrantLock` não permitirá que outra thread obtenha o bloqueio até que a primeira thread libere-o.

d) Os componentes Swing são seguros para thread.

26.4 Defina cada um dos seguintes termos.

a) thread

b) multithreading

c) estado *executável*

d) estado de *espera sincronizada*

e) agendamento preemptivo

f) interface `Runnable`

g) método `notifyAll`

h) produtor/consumidor, relacionamento

i) quantum

26.5 Discuta cada um dos seguintes termos no contexto de mecanismos de thread do Java:
a) `synchronized`
b) produtor
c) consumidor
d) `wait`
e) `notify`
f) `Lock`
g) `Condition`

26.6 Liste as razões para entrar no estado *bloqueado*. Para cada uma delas, descreva como o programa normalmente deixará o estado *bloqueado* e entrará no estado *executável*.

26.7 Dois problemas que podem ocorrer em sistemas que permitem às threads esperar são os impasses, em que uma ou mais threads esperarão eternamente por um evento que pode não ocorrer, e o adiamento indefinido, em que uma ou mais threads serão retardadas por um tempo imprevisivelmente longo. Dê um exemplo de como cada um desses problemas podem ocorrer em programas Java de múltiplas threads.

26.8 (***Rebatendo a bola***) Escreva um programa que faz uma bola azul rebater dentro de um `JPanel`. A bola deve começar a se mover com um evento `mousePressed`. Quando a bola atingir a borda do `JPanel`, ela deve rebater fora da borda e continuar na direção oposta. A bola deve ser atualizada com uma interface `Runnable`.

26.9 (***Rebatendo as bolas***) Modifique o programa no Exercício 26.8 para adicionar uma nova bola toda vez que o usuário clicar no mouse. Ofereça um mínimo de 20 bolas. Escolha a cor para cada nova bola aleatoriamente.

26.10 (***Bolas que rebatem com sombras***) Modifique o programa do Exercício 26.9 para adicionar sombras. À medida que uma bola se mover, desenhe uma oval sólida preta na parte inferior do `JPanel`. Você pode considerar a adicionar um efeito 3-D aumentando ou diminuindo o tamanho de cada bola quando ela atingir a borda do `JPanel`.

26.11 (***Buffer circular com `Locks` e `Conditions`***) Reimplemente o exemplo da Seção 26.9 utilizando os conceitos `Lock` e `Condition` apresentados na Seção 26.10.

27

Se a presença da eletricidade pode ser tornada visível em qualquer parte de um circuito, não vejo razão por que a inteligência não possa ser transmitida instantaneamente pela eletricidade.
— **Samuel F. B. Morse**

Protocolo é tudo.
— **François Giuliani**

O que as redes de ferrovias, rodovias e canais foram em outra era, as redes de telecomunicações, informações e computação ... são hoje em dia.
— **Bruno Kreisky**

The port is near, the bells I hear, the people all exulting.
[A porta está próxima, ouço os sinos, o povo está exultante.]
— **Walt Whitman**

Redes

Objetivos

Neste capítulo, você aprenderá:

- A entender a tecnologia de redes do Java com URLs, sockets e datagramas.
- A implementar aplicativos de rede de Java utilizando sockets e datagramas.
- A entender como implementar clientes e servidores Java que se comunicam entre si.
- A entender como implementar aplicativos colaborativos baseados em rede.
- A construir um servidor multiencadeado.

Sumário

27.1 Introdução
27.2 Manipulando URLs
27.3 Lendo um arquivo em um servidor da Web
27.4 Estabelecendo um servidor simples utilizando sockets de fluxo
27.5 Estabelecendo um cliente simples utilizando sockets de fluxo
27.6 Interação cliente/servidor com conexões de socket de fluxo
27.7 Interação cliente/servidor sem conexão com datagramas
27.8 Jogo da velha cliente/servidor que utiliza um servidor com multithread
27.9 [Bônus Web] Estudo de caso: Servidor e Cliente `DeitelMessenger`
27.10 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

27.1 Introdução

Há um grande entusiasmo com a Internet e a World Wide Web. A internet une o mundo da informação. A World Wide Web torna a Internet fácil de utilizar e se beneficia da "onda" multimídia. As organizações consideram a Internet e a Web como cruciais para suas estratégias na área de sistemas de informações. O Java fornece diversas capacidades de rede predefinidas que tornam fácil desenvolver aplicativos com base na Internet e na Web. O Java pode permitir que programas pesquisem informações no mundo e colaborem com programas que são executados em outros computadores internacionalmente, nacionalmente ou simplesmente em uma empresa. O Java pode permitir que applets e aplicativos se comuniquem entre si (sujeito a restrições de segurança).

Os recursos fundamentais das redes em Java são declarados pelas classes e interfaces do pacote **java.net**, por meio do qual o Java oferece **comunicações baseadas em fluxo** que permitem aos aplicativos visualizar as redes como fluxos de dados. As classes e interfaces do pacote `java.net` também oferecem **comunicações baseadas em pacotes** para transmitir **pacotes** individuais de informações — comumente utilizados para transmitir dados, imagens, áudio e vídeo pela Internet. Neste capítulo, mostramos como criar e manipular sockets e como se comunicar com pacotes e fluxos de dados.

Focalizamos os dois lados do **relacionamento cliente/servidor**. O **cliente** solicita que alguma ação seja realizada e o **servidor** realiza a ação e responde para o cliente. Uma implementação comum do modelo de solicitação e resposta é aquela encontrada nos navegadores e servidores Web. Quando um usuário seleciona um site da Web para navegar com seu navegador (o aplicativo cliente), uma solicitação é enviada para o servidor da Web apropriado (o aplicativo servidor). O servidor normalmente responde para o cliente enviando uma página Web apropriada.

Introduzimos as **comunicações baseadas em socket** do Java, que permitem aos aplicativos visualizar a rede como se fosse uma E/S de arquivo — um programa pode ler um **socket** ou gravar em um socket tão simplesmente quanto lê de um arquivo ou grava em um arquivo. O socket é simplesmente uma construção de software que representa uma extremidade final de uma conexão. Mostramos como criar e manipular sockets de fluxo e sockets de datagrama.

Com **sockets de fluxo**, um processo estabelece uma **conexão** com outro processo. Enquanto a conexão estiver no ar, os dados fluem entre os processos em **fluxos** contínuos. Dizemos que os sockets de fluxo fornecem um **serviço orientado para conexão**. O protocolo utilizado para transmissão é o popular **TCP (Transmission Control Protocol)**.

Com **sockets de datagrama**, são transmitidos **pacotes** individuais de informações. Isso não é apropriado para programadores rotineiros, porque o protocolo utilizado — **UDP**, o **User Datagram Protocol** — é um **serviço sem conexão** e, assim, não garante que pacotes cheguem em uma ordem particular. Com o UDP, os pacotes podem até mesmo ser perdidos ou duplicados. É necessária uma quantidade significativa de programação extra de sua parte para lidar com esses problemas (se optar por fazer isso). O UDP é mais apropriado para aplicativos de rede que não requerem verificação de erros e confiabilidade do TCP. Os sockets de fluxo e o protocolo TCP serão os mais desejáveis para ampla maioria dos programadores Java.

Dica de desempenho 27.1

Serviços sem conexão geralmente oferecem desempenho maior, mas menos confiabilidade que os serviços orientados para conexão.

Dica de portabilidade 27.1

O TCP, UDP e protocolos relacionados permitem que sistemas de computadores heterogêneos (isto é, sistemas de computadores com diferentes processadores e diferentes sistemas operacionais) se intercomuniquem.

Apresentamos um estudo de caso que implementa um aplicativo de chat ("bate-papo") cliente/servidor semelhante aos serviços de sistema de troca de mensagens instantâneas populares na Web hoje em dia. Esse estudo de caso é fornecido em inglês como um bônus Web em `www.deitel.com/books/jhtp8/`. O aplicativo incorpora muitas técnicas para redes introduzidas neste capítulo. Ele também introduz o **multicasting**, com o qual um servidor pode publicar informações e muitos clientes podem assiná-las. Toda vez que o servidor publica informações adicionais, todos os assinantes recebem essas informações. Por todos os exemplos deste capítulo, veremos que vários detalhes das redes são tratados pelas Java APIs.

27.2 Manipulando URLs

[*Nota*: O exemplo nesta seção requer um pouco de conhecimento de XHTML. Para aprender mais sobre a XHTML, visite nosso XHTML Resource Center em `www.deitel.com/xhtml/`. Talvez você também esteja interessado na capacidade de formatação de páginas Web conhecida como Cascading Style Sheets (CSS). Para informações adicionais, visite nosso CSS Resource Center em `www.deitel.com/CSS21/`.]

A Internet oferece muitos protocolos. O **HyperText Transfer Protocol** (**HTTP**), que forma a base da World Wide Web, utiliza **URIs** (**Uniform Resource Identifiers**) para identificar dados na Internet. URIs que especificam as localizações dos documentos são chamados **URLs** (**Uniform Resource Locators**). URLs comuns fazem referência a arquivos ou diretórios e podem também fazer referências a objetos que realizam tarefas complexas, como pesquisas no banco de dados e pesquisas na Internet. Se conhecer o URL de uma página Web publicamente disponível, você poderá acessá-la por meio do HTTP.

O Java torna fácil manipular os URLs. Ao utilizar um URL que faz referência à localização exata de um recurso (por exemplo, uma página Web) como um argumento para o método `showDocument` da interface `AppletContext`, o navegador em que o applet está executando acessará e exibirá esse recurso. O applet nas figuras 27.1–27.2 demonstra capacidades simples de rede. Ele permite que o usuário selecione uma página Web a partir de uma `JList` e faz com que o navegador exiba a página correspondente. Neste exemplo, a conexão de rede é realizada pelo navegador.

```
1  <html>
2  <head>
3     <title>Site Selector</title>
4  </head>
5  <body>
6     <applet code = "SiteSelector.class" width = "300" height = "75">
7        <param name = "title0" value = "Java Home Page">
8        <param name = "location0" value = "http://java.sun.com/">
9        <param name = "title1" value = "Deitel">
10       <param name = "location1" value = "http://www.deitel.com/">
11       <param name = "title2" value = "JGuru">
12       <param name = "location2" value = "http://www.jGuru.com/">
13       <param name = "title3" value = "JavaWorld">
14       <param name = "location3" value = "http://www.javaworld.com/">
15    </applet>
16 </body>
17 </html>
```

Figura 27.1 | O documento XHTML para carregar o applet `SiteSelector`.

```
1  // Figura 27.2: SiteSelector.java
2  // Carregando um documento de um URL em um navegador.
3  import java.net.MalformedURLException;
4  import java.net.URL;
5  import java.util.HashMap;
6  import java.util.ArrayList;
7  import java.awt.BorderLayout;
8  import java.applet.AppletContext;
9  import javax.swing.JApplet;
10 import javax.swing.JLabel;
11 import javax.swing.JList;
12 import javax.swing.JScrollPane;
13 import javax.swing.event.ListSelectionEvent;
14 import javax.swing.event.ListSelectionListener;
15
16 public class SiteSelector extends JApplet
17 {
18    private HashMap< String, URL > sites; // nomes e URLs de site
```

```
19       private ArrayList< String > siteNames; // nomes de site
20       private JList siteChooser; // lista de sites a escolher
21
22       // lê os parâmetros e configura a GUI
23       public void init()
24       {
25          sites = new HashMap< String, URL >(); // cria HashMap
26          siteNames = new ArrayList< String >(); // cria a ArrayList
27
28          // obtém os parâmetros do documento XHTML
29          getSitesFromHTMLParameters();
30
31          // cria componentes GUI e a interface de layout
32          add( new JLabel( "Choose a site to browse" ), BorderLayout.NORTH );
33
34          siteChooser = new JList( siteNames.toArray() ); // preenche a JList
35          siteChooser.addListSelectionListener(
36             new ListSelectionListener() // classe interna anônima
37             {
38                // vai ao site selecionado pelo usuário
39                public void valueChanged( ListSelectionEvent event )
40                {
41                   // obtém o nome do site selecionado
42                   Object object = siteChooser.getSelectedValue();
43
44                   // utiliza o nome do site para localizar o URL correspondente
45                   URL newDocument = sites.get( object );
46
47                   // obtém o contêiner de applets
48                   AppletContext browser = getAppletContext();
49
50                   // instrui o contêiner de applets a mudar as páginas
51                   browser.showDocument( newDocument );
52                } // fim do método valueChanged
53             } // fim da classe interna anônima
54          ); // fim da chamada para addListSelectionListener
55
56          add( new JScrollPane( siteChooser ), BorderLayout.CENTER );
57       } // fim do método init
58
59       // obtém os parâmetros do documento XHTML
60       private void getSitesFromHTMLParameters()
61       {
62          String title; // título do site
63          String location; // localização do site
64          URL url; // URL da localização
65          int counter = 0; // conta número de sites
66
67          title = getParameter( "title" + counter ); // obtém o primeiro título do site
68
69          // faz um loop até que não haja mais parâmetros no documento XHTML
70          while ( title != null )
71          {
72             // obtém a localização do site
73             location = getParameter( "location" + counter );
74
75             try // coloca título/URL no HashMap e título na ArrayList
76             {
77                url = new URL( location ); // converte a localização em URL
78                sites.put( title, url ); // coloca título/URL no HashMap
79                siteNames.add( title ); // coloca o título na ArrayList
80             } // fim do try
81             catch ( MalformedURLException urlException )
82             {
83                urlException.printStackTrace();
84             } // fim do catch
85
86             ++counter;
87             title = getParameter( "title" + counter ); // obtém o próximo título do site
```

```
88          } // fim do while
89       } // fim do método getSitesFromHTMLParameters
90    } // fim da classe SiteSelector
```

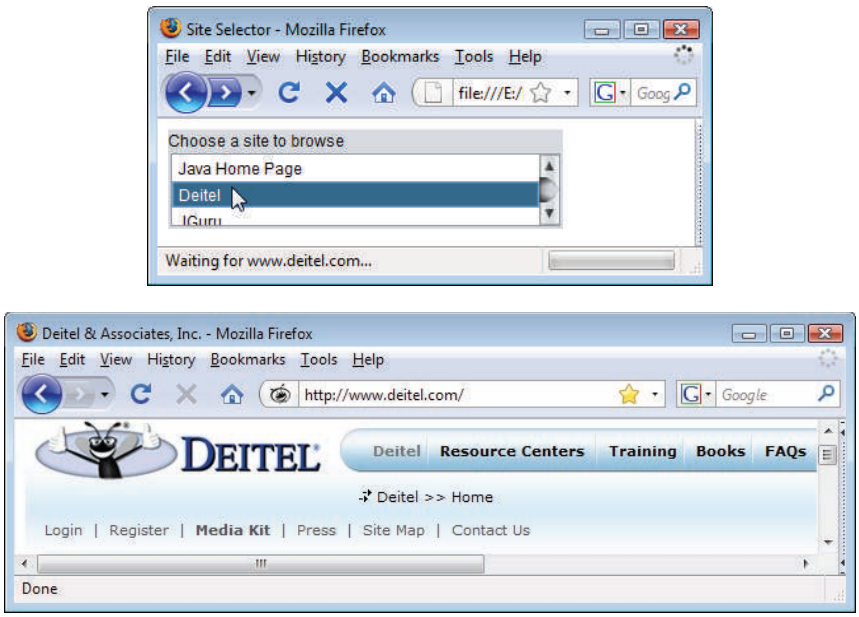


Figura 27.2 | Carregando um documento de um URL em um navegador.

Processando parâmetros de applet

Esse applet tira proveito dos **parâmetros de applet** especificados no documento XHTML que invoca o applet. Ao navegar pela World Wide Web, você irá frequentemente se deparar com applets que são de domínio público — você pode utilizá-los gratuitamente em suas próprias páginas Web (normalmente em troca do reconhecimento da autoria do applet). Muitos applets podem ser personalizados via parâmetros fornecidos no arquivo XHTML que invoca o applet. Por exemplo, a Figura 27.1 contém o XHTML que invoca o applet `SiteSelector` na Figura 27.2.

O documento XHTML contém oito parâmetros especificados com o **`param element`** — essas linhas devem aparecer entre os tags `applet` de abertura e fechamento. O applet pode ler e utilizar esses valores para se personalizar. Qualquer número de elementos `param` pode aparecer entre os tags `applet` inicial e final. Cada parâmetro tem um **nome** e um **valor** únicos. O método `Applet` **`getParameter`** retorna o `value` associado a um nome de parâmetro específico como uma `String`. O argumento passado para `getParameter` é uma `String` contendo o nome do parâmetro no elemento `param`. Nesse exemplo, os parâmetros representam o título e a localização de cada site da Web que o usuário pode selecionar. Os parâmetros especificados para esse applet são nomeados `title#`, onde o valor de `#` inicia em `0` e incrementa por 1 para cada novo título. Cada título deve ter um parâmetro de localização correspondente na forma `location#`, onde o valor de `#` inicia em `0` e incrementa por 1 para cada nova localização. A instrução

```
String title = getParameter( "title0" );
```

obtém o valor associado com o parâmetro `"title0"` e o atribui à referência `title`. Se não houver um tag `param` que contém o parâmetro especificado, `getParameter` retornará `null`.

Armazenando nomes e URLs de sites Web

O applet (Figura 27.2) obtém no documento XHTML (Figura 27.1) as opções que serão exibidas na `JList` do applet. A classe `SiteSelector` utiliza um `HashMap` (pacote `java.util`) para armazenar os nomes e URLs de sites da Web. Neste exemplo, a *chave* é a `String` na `JList` que representa o nome do site da Web e o *valor* é um objeto `URL` que armazena a localização do site que será exibido no navegador.

A classe `SiteSelector` também contém uma `ArrayList` (pacote `java.util`) em que os nomes de site são colocados de modo que possam ser usados para inicializar a `JList` (uma versão do construtor `JList` recebe um array de `Objects` que é retornado pelo método `ArrayList` `toArray`). Uma `ArrayList` é um array dinamicamente redimensionável de referências. A classe `ArrayList` fornece o método `add` para adicionar um novo elemento ao final da `ArrayList`. (`ArrayList` e `HashMap` foram discutidos no Capítulo 20.)

As linhas 25–26 no método `init` do applet (linhas 23–57) criam um objeto `HashMap` e um objeto `ArrayList`. A linha 29 chama nosso método utilitário `getSitesFromHTMLParameters` (declarado nas linhas 60–89) para obter os parâmetros XHTML no documento XHTML que invocou o applet.

O método `getSitesFromHTMLParameters` utiliza o método `Applet` `getParameter` (linha 67) para obter o título de um site Web. Se `title` não for `null`, as linhas 73–87 são executadas. A linha 73 utiliza o método `Applet` `getParameter` para obter a localização do site da Web. A linha 77 utiliza `location` como o valor de um novo objeto `URL`. O construtor `URL` determina se seu argumento representa um URL válido. Se não representar, o construtor `URL` lança uma **`MalformedURLException`**. Observe que o construtor `URL` deve ser chamado em um bloco `try`. Se o construtor `URL` gerar uma `MalformedURLException`, a chamada a `printStackTrace` (linha 83) faz com que o programa mostre um rastreamento de pilha no console Java. Em máquinas Windows, o console Java pode ser visualizado dando um clique com botão direito do mouse no ícone Java na área de notificação da barra de tarefas. Em outras plataformas, isso é tipicamente acessível por meio de um ícone de área de trabalho. Então, o programa tenta obter o próximo título de site da Web. O programa não adiciona o site do URL inválido ao `HashMap`, portanto o título não será exibido na `JList`.

Para um `URL` apropriado, a linha 78 insere `title` e `URL` no `HashMap`, e a linha 79 adiciona `title` a `ArrayList`. A linha 87 obtém o próximo título do documento XHTML. Quando a chamada a `getParameter` na linha 87 retorna `null`, o loop termina.

Criando a GUI do applet

Quando o método `getSitesFromHTMLParameters` retorna a `init`, as linhas 32–56 constroem a GUI do applet. A linha 32 adiciona o `JLabel` `Choose a site to browse` ao `NORTH` do `BorderLayout` do `JApplet`. A linha 34 cria `JList` `siteChooser` para permitir que o usuário selecione uma página da Web para visualizar. As linhas 35–54 registram um `ListSelectionListener` para tratar os eventos `JList`. A linha 56 adiciona `siteChooser` ao `CENTER` do `BorderLayout` do `JFrame`.

Processando uma seleção de usuário

Quando o usuário seleciona um dos sites Web listados em `siteChooser`, o programa chama o método `valueChanged` (linhas 39–52). A linha 42 obtém o nome do sites selecionados na `JList`. A linha 45 passa o nome do site selecionado (a *chave*) para o método `HashMap` `get`, que localiza e retorna uma referência ao objeto `URL` correspondente (o *valor*) atribuído à referência `newDocument`.

A linha 48 utiliza o método `Applet` **`getAppletContext`** para obter uma referência a um objeto `AppletContext` que representa o contêiner de applets. A linha 51 utiliza a referência `AppletContext` `browser` para invocar o método `showDocument`, que recebe um objeto `URL` como um argumento e o passa para o `AppletContext` (isto é, o navegador). O navegador exibe na janela de navegador atual o recurso associado com esse `URL`. Neste exemplo, todos os recursos são documentos XHTML.

Especificando o quadro-alvo para o método `showDocument`

Uma segunda versão do método `AppletContext` `showDocument` permite a um applet especificar o chamado **frame-alvo** em que o recurso da Web será exibido. Essa segunda versão recebe dois argumentos — um objeto `URL` que especifica o recurso a ser exibido e uma `String` que representa o frame-alvo. Há alguns frames especiais de alvo que podem ser utilizados como o segundo argumento. O frame-alvo **`_blank`** resulta em uma nova janela de navegador Web para exibir o conteúdo do URL especificado. O frame-alvo **`_self`** determina que o conteúdo do URL especificado deve ser exibido no mesmo frame do applet (nesse caso, a página XHTML do applet é substituída). O frame-alvo **`_top`** especifica que o navegador deve remover os frames atuais da janela do navegador e, então, exibir o conteúdo do URL especificado na janela atual.

Dica de prevenção de erro 27.1

O applet na Figura 27.2 deve ser executado a partir de um navegador Web, como o Mozilla ou o Microsoft Internet Explorer, para ver os resultados da exibição de outra página Web. O `appletviewer` é capaz de executar somente applets — ele ignora todos os outros tags de XHTML. Se os sites da Web no programa contivessem applets Java, somente esses applets apareceriam no `appletviewer` quando o usuário selecionasse um site. Cada applet executaria em uma janela `appletviewer` separada.

27.3 Lendo um arquivo em um servidor da Web

Nosso próximo exemplo mais uma vez oculta os detalhes de conexão de rede. O aplicativo na Figura 27.3 utiliza o componente GUI Swing **`JEditorPane`** (do pacote `javax.swing`) para exibir o conteúdo de um arquivo em um servidor da Web. O usuário insere um URL no `JTextField` na parte superior da janela e o aplicativo exibe o documento correspondente (se existir) no `JEditorPane`. A classe `JEditorPane` é capaz de renderizar texto simples e texto formatado em XHTML, como ilustrado nas duas capturas de tela (Figura 27.4); assim, esse aplicativo atua como um simples navegador Web. O aplicativo também demonstra como processar **`HyperlinkEvents`** quando o usuário clica em um hiperlink do documento XHTML. As técnicas mostradas nesse exemplo também podem ser utilizadas em applets. Entretanto, um applet tem permissão para ler arquivos somente no servidor a partir do qual foi feito o download. [*Nota*: Esse programa talvez não funcione se o navegador Web precisar acessar a Web por meio de um servidor proxy. Se você criar um documento JNLP para esse programa e utilizar o Java Web Start para inicializá-lo, o Java Web Start utilizará as configurações do servidor proxy a partir do navegador Web padrão.]

```
1  // Figura 27.3: ReadServerFile.java
2  // Ler um arquivo abrindo uma conexão por meio de um URL.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.io.IOException;
7  import javax.swing.JEditorPane;
8  import javax.swing.JFrame;
9  import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextField;
12 import javax.swing.event.HyperlinkEvent;
13 import javax.swing.event.HyperlinkListener;
14
15 public class ReadServerFile extends JFrame
16 {
17    private JTextField enterField; // JTextField para inserir o nome de site
18    private JEditorPane contentsArea; // para exibir o site Web
19
20    // configura a GUI
21    public ReadServerFile()
22    {
23       super( "Simple Web Browser" );
24
25       // cria o enterField e registra seu ouvinte
26       enterField = new JTextField( "Enter file URL here" );
27       enterField.addActionListener(
28          new ActionListener()
29          {
30             // obtém o documento especificado pelo usuário
31             public void actionPerformed( ActionEvent event )
32             {
33                getThePage( event.getActionCommand() );
34             } // fim do método actionPerformed
35          } // fim da classe inner
36       ); // fim da chamada para addActionListener
37
38       add( enterField, BorderLayout.NORTH );
39
40       contentsArea = new JEditorPane(); // cria a contentsArea
41       contentsArea.setEditable( false );
42       contentsArea.addHyperlinkListener(
43          new HyperlinkListener()
44          {
45             // se o usuário clicou no hyperlink, vai para a página especificada
46             public void hyperlinkUpdate( HyperlinkEvent event )
47             {
48                if ( event.getEventType() ==
49                     HyperlinkEvent.EventType.ACTIVATED )
50                   getThePage( event.getURL().toString() );
51             } // fim do método hyperlinkUpdate
52          } // fim da classe inner
53       ); // fim da chamada a addHyperlinkListener
54
55       add( new JScrollPane( contentsArea ), BorderLayout.CENTER );
56       setSize( 400, 300 ); // configura o tamanho da janela
57       setVisible( true ); // mostra a janela
58    } // fim do construtor ReadServerFile
59
60    // carrega o documento
61    private void getThePage( String location )
62    {
63       try // carrega o documento e exibe a localização
64       {
65          contentsArea.setPage( location ); // configura a página
66          enterField.setText( location ); // configura o texto
67       } // fim do try
68       catch ( IOException ioException )
69       {
70          JOptionPane.showMessageDialog( this,
71             "Error retrieving specified URL", "Bad URL",
```

```
72                JOptionPane.ERROR_MESSAGE );
73             } // fim do catch
74          } // fim do método getThePage
75       } // fim da classe ReadServerFile
```

Figura 27.3 | Lendo um arquivo ao abrir uma conexão por meio de um URL.

```
1  // Figura 27.4: ReadServerFileTest.java
2  // Cria e inicia um ReadServerFile.
3  import javax.swing.JFrame;
4
5  public class ReadServerFileTest
6  {
7     public static void main( String[] args )
8     {
9        ReadServerFile application = new ReadServerFile();
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11    } // fim de main
12 } // fim da classe ReadServerFileTest
```

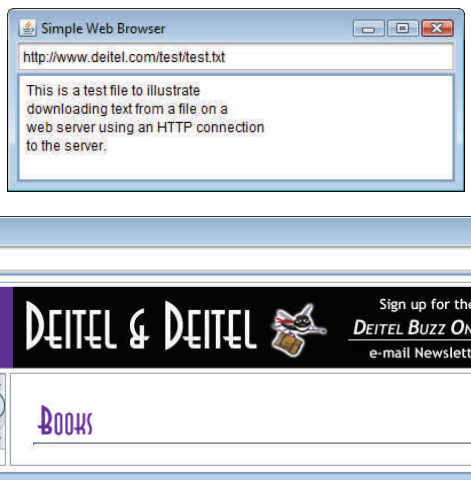


Figura 27.4 | Classe de teste para o `ReadServerFile`.

A classe de aplicativo `ReadServerFile` contém `JTextField enterField`, em que o usuário insere o URL do arquivo para ler e o `JEditorPane contentsArea` para exibir o conteúdo do arquivo. Quando o usuário pressiona a tecla *Enter* no `enterField`, o aplicativo chama o método `actionPerformed` (linhas 31–34). A linha 33 utiliza o método `ActionEvent getActionCommand` para obter a `String` que o usuário inseriu no `JTextField` e passa a `String` para o método utilitário `getThePage` (linhas 61–74).

A linha 65 invoca o método `JEditorPane` **`setPage`** para fazer o download do documento especificado pela `location` e exibi-lo no `JEditorPane`. Se houver um erro ao fazer o download do documento, o método `setPage` lançará uma `IOException`. Além disso, se um URL inválido for especificado, uma `MalformedURLException` (uma subclasse de `IOException`) ocorrerá. Se o documento carregar com sucesso, a linha 66 exibe a localização atual no `enterField`.

Tipicamente, um documento XHTML contém **hiperlinks** que, quando clicados, fornecem acesso rápido a outro documento na Web. Se um `JEditorPane` contiver um documento XHTML e o usuário clicar em um hiperlink, o `JEditorPane` irá gerar um **`HyperlinkEvent`** (pacote `javax.swing.event`) e notificará todos os **`HyperlinkListener`**s registrados (pacote `javax.swing.event`) desse evento. As linhas 42–53 registram um `HyperlinkListener` para tratar `HyperlinkEvents`. Quando um `HyperlinkEvent` ocorre, o programa chama o método **`hyperlinkUpdate`** (linhas 46–51). As linhas 48–49 utilizam o método `HyperlinkEvent` **`getEventType`** para determinar o tipo do `HyperlinkEvent`. A classe `HyperlinkEvent` contém uma classe aninhada `public` chamada **`EventType`** que declara três objetos `static EventType`, que representam os tipos de eventos de hiperlink. **`ACTIVATED`** indica que o usuário clicou em um hiperlink para mudar páginas Web, **`ENTERED`** indica que o usuário moveu o mouse sobre um hiperlink e **`EXITED`** indica que o usuário tirou o mouse de cima de um hiperlink. Se um hiperlink estiver `ACTIVATED`, a linha 50 utiliza o método `HyperlinkEvent` **`getURL`** para obter o `URL` representado pelo hiperlink. O método `toString` converte o `URL` retornado em uma `String` que pode ser passada ao método utilitário `getThePage`.

Observações sobre a aparência e comportamento 27.1

Um `JEditorPane` gera `HyperlinkEvents` somente se eles não forem editáveis.

27.4 Estabelecendo um servidor simples utilizando sockets de fluxo

Os dois exemplos discutidos até agora utilizam recursos de alto nível de Java para redes a fim de se comunicarem entre aplicativos. Nos exemplos, não era sua responsabilidade estabelecer a conexão entre um cliente e um servidor. O primeiro programa contava com o navegador Web para se comunicar com um servidor Web. O segundo programa contava com um `JEditorPane` para realizar a conexão. Esta seção inicia nossa discussão sobre a criação dos seus próprios aplicativos que podem se comunicar entre si.

Passo 1: Cria um ServerSocket

Estabelecer um servidor simples em Java requer cinco passos. O passo 1 é criar um objeto **ServerSocket**. Uma chamada ao construtor `ServerSocket` como

```
ServerSocket server = new ServerSocket( númeroDePorta, comprimentoDaFila );
```

registra um **número de porta** TCP disponível e especifica um número máximo de clientes que podem esperar para se conectar ao servidor (isto é, o **comprimento da fila**). O número da porta é utilizado pelos clientes para localizar o aplicativo servidor no computador servidor. Isso costuma ser chamado **ponto de handshake**. Se a fila estiver cheia, o servidor recusará as conexões do cliente. O construtor estabelece a porta em que o servidor espera as conexões dos clientes — um processo conhecido como **vincular o servidor à porta**. Cada cliente solicitará para conectar-se ao servidor nessa **porta**. Somente um aplicativo por vez pode ser vinculado a uma porta específica no servidor.

Observação de engenharia de software 27.1

Os números de porta podem estar entre 0 e 65.535. A maioria dos sistemas operacionais reserva números de porta abaixo de 1.024 para serviços de sistema (por exemplo, correio eletrônico e servidores da World Wide Web). Geralmente, essas portas não devem ser especificadas como portas de conexão em programas de usuário. De fato, alguns sistemas operacionais requerem privilégios especiais de acesso para se vincularem a números de porta abaixo de 1.024.

Passo 2: Espera uma conexão

Os programas gerenciam cada conexão de cliente com um objeto **`Socket`**. No *Passo 2*, o servidor passa a ouvir indefinidamente (ou **bloquear**) uma tentativa de conexão por um cliente. Para ouvir uma conexão de cliente, o programa chama o método `ServerSocket` **`accept`**, como em

```
Socket connection = server.accept();
```

que retorna um `Socket` quando uma conexão com um cliente é estabelecida. O `Socket` permite ao servidor interagir com o cliente. Na verdade, as interações com o cliente ocorrem em uma porta diferente de servidor a partir do ponto de handshake. Isso permite que a porta especificada no *Passo 1* seja utilizada novamente em um servidor de múltiplas threads para aceitar outra conexão de cliente. Demonstraremos esse conceito na Seção 27.8.

Passo 3: Obtém os fluxos de E/S do Socket

O Passo 3 é obter os objetos `OutputStream` e `InputStream` que permitem ao servidor se comunicar com o cliente enviando e recebendo bytes. O servidor envia informações ao cliente via um `OutputStream` e recebe informações do cliente via um `InputStream`. O servidor invoca o método **`getOutputStream`** no `Socket` para obter uma referência ao `OutputStream` do `Socket` e invoca o método **`getInputStream`** no `Socket` para obter uma referência ao `InputStream` do `Socket`.

Os objetos-fluxo podem ser utilizados para enviar ou receber bytes individuais ou sequências de bytes com o método `write` de `OutputStream` e com o método `read` de `InputStream`, respectivamente. Com frequência, é útil enviar ou receber valores de tipos primitivos (por exemplo, `int` e `double`) ou objetos `Serializable` (por exemplo, `String`s ou outros tipos serializáveis) em vez de enviar bytes. Nesse caso, podemos utilizar as técnicas discutidas no Capítulo 17 para empacotar outros tipos de fluxo (por exemplo, `ObjectOutputStream` e `ObjectInputStream`) em torno do `OutputStream` e `InputStream` associados com o `Socket`. Por exemplo,

```
ObjectInputStream input =
   new ObjectInputStream( connection.getInputStream() );

ObjectOutputStream output =
   new ObjectOutputStream( connection.getOutputStream() );
```

A beleza de estabelecer esses relacionamentos é que tudo o que o servidor grava no `ObjectOutputStream` é enviado via o `OutputStream` e está disponível no `InputStream` do cliente e que tudo o que o cliente grava em seu `OutputStream` (com um `ObjectOutputStream` correspondente) está disponível via o `InputStream` do servidor. A transmissão dos dados sobre a rede é transparência e é tratada completamente pelo Java.

Passo 4: Realiza o processamento

O *Passo 4* é a fase de *processamento* em que o servidor e o cliente comunicam-se via objetos `OutputStream` e `InputStream`.

Passo 5: Fecha a conexão

No *Passo 5*, quando a transmissão está completa, o servidor fecha a conexão invocando o método **`close`** nos fluxos e no `Socket`.

Observação de engenharia de software 27.2

Com os sockets, a E/S de rede aparece para programas Java como similar à E/S de arquivo sequencial. Os sockets ocultam de você muito da complexidade da programação de rede.

Observação de engenharia de software 27.3

Um servidor com múltiplos threads pode receber o `Socket` retornado por cada chamada a `accept` e criar uma nova thread que gerencia a E/S de rede por meio desse `Socket`. Alternativamente, um servidor com múltiplas threads pode manter um pool de threads (um conjunto de threads existente) pronto para gerenciar a E/S de rede por meio dos novos `Sockets` à medida que são criados. Essas técnicas permitem que servidores multithread gerenciem várias conexões cliente simultâneas.

Dica de desempenho 27.2

Em sistemas de alto desempenho nos quais há memória abundante, um servidor multithread pode criar um grupo de threads que podem ser atribuídos rapidamente para tratar E/S de rede para novos `Sockets` à medida que eles são criados. Portanto, quando o servidor recebe uma conexão, ele não precisa incorrer no overhead da criação de thread. Quando a conexão é fechada, a thread é retornada ao pool para reutilização.

27.5 Estabelecendo um cliente simples utilizando sockets de fluxo

Estabelecer um cliente simples em Java exige quatro passos.

Passo 1: Cria um `Socket` para conectar ao servidor

No *Passo 1*, criamos um `Socket` para nos conectarmos ao servidor. O construtor `Socket` estabelece a conexão. Por exemplo, a instrução

```
Socket connection = new Socket( endereçoDoServidor, porta );
```

utiliza o construtor de `Socket` com dois argumentos — o endereço do servidor (*endereçoDoServidor*) e o número da *porta*. Se a tentativa de conexão for bem-sucedida, essa instrução retorna um `Socket`. Uma tentativa de conexão que falha lança uma instância de uma subclasse de `IOException`, muitos programas simplesmente capturam `IOException`. Uma **`UnknownHostException`** ocorre especificamente quando o sistema não consegue converter o nome do servidor especificado na chamada ao construtor de `Socket` para um endereço IP correspondente.

Passo 2: Obtém os fluxos de E/S do `Socket`

No *Passo 2*, o cliente utiliza os métodos `getInputStream` e `getOutputStream` de `Socket` para obter referências ao `InputStream` e `OutputStream` do `Socket`. Como mencionamos na seção anterior, podemos utilizar as técnicas do Capítulo 17 para empacotar outros tipos de fluxos no `InputStream` e `OutputStream` associado com o `Socket`. Se o servidor está enviando informações na forma de tipos reais, o cliente deve receber as informações no mesmo formato. Assim, se o servidor enviar valores com um `ObjectOutputStream`, o cliente deverá ler esses valores com um `ObjectInputStream`.

Passo 3: Realiza o processamento

O *Passo 3* é a fase de processamento em que o cliente e o servidor comunicam-se via objetos `InputStream` e `OutputStream`.

Passo 4: Fecha a conexão

No *Passo 4*, o cliente fecha a conexão quando a transmissão está completa invocando o método `close` nos fluxos e no `Socket`. O cliente deve determinar quando o servidor termina o envio das informações de modo que possa chamar `close` para fechar a conexão no `Socket`. Por exemplo, o método `InputStream` `read` retorna o valor –1 quando detecta o fim de fluxo (também chamado EOF [end-of-file, fim do arquivo]). Se um `ObjectInputStream` ler as informações do servidor, uma `EOFException` ocorrerá quando o cliente tentar ler o valor de um fluxo em que o fim de fluxo for detectado.

27.6 Interação cliente/servidor com conexões de socket de fluxo

As figuras 27.5 e 27.7 utilizam os sockets de fluxo, `ObjectInputStream` e `ObjectOutputStream` para demonstrar um **aplicativo de chat client/server** simples. O servidor espera uma tentativa de conexão do cliente. Quando um cliente se conecta ao servidor, o aplicativo servidor envia ao cliente um objeto `String` (lembre-se de que `String`s são objetos `Serializáveis`) para indicar que a conexão foi bem-sucedida. Em seguida, o cliente exibe a mensagem. Os aplicativos servidores e clientes fornecem campos de texto que permitem ao usuário digitar uma mensagem e enviá-la a outro aplicativo. Quando o cliente ou o servidor envia a `String "TERMINATE"`, a conexão é encerrada. Então o servidor espera o próximo cliente se conectar. A declaração da classe `Server` aparece na Figura 27.5. A declaração da classe `Client` aparece na Figura 27.7. As capturas de tela que mostram a execução entre o cliente e o servidor são mostradas na Figura 27.8.

Classe Server

O construtor da classe `Server` (Figura 27.5, linhas 30–55) cria a GUI do servidor, que contém um `JTextField` e uma `JTextArea`. A classe `Server` exibe sua saída na `JTextArea`. Quando o método `main` (linhas 6–11 da Figura 27.6) executa, ele cria um objeto `Server`, especifica a operação padrão de fechamento de janela e chama o método `runServer` (Figura 27.5, linhas 57–86).

```
1  // Figura 27.5: Server.java
2  // A parte servidor de uma conexão cliente/servidor de socket de fluxo.
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.net.ServerSocket;
8  import java.net.Socket;
9  import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Server extends JFrame
19 {
20    private JTextField enterField; // insere a mensagem do usuário
21    private JTextArea displayArea; // exibe informações para o usuário
22    private ObjectOutputStream output; // gera fluxo de saída para o cliente
23    private ObjectInputStream input; // gera fluxo de entrada a partir do cliente
24    private ServerSocket server; // socket de servidor
25    private Socket connection; // conexão com o cliente
26    private int counter = 1; // contador do número de conexões
27
28    // configura a GUI
29    public Server()
30    {
31       super( "Server" );
32
33       enterField = new JTextField(); // cria enterField
34       enterField.setEditable( false );
35       enterField.addActionListener(
36          new ActionListener()
37          {
38             // envia a mensagem ao cliente
39             public void actionPerformed( ActionEvent event )
40             {
41                sendData( event.getActionCommand() );
42                enterField.setText( "" );
43             } // fim do método actionPerformed
44          } // fim da classe interna anônima
45       ); // fim da chamada para addActionListener
46
47       add( enterField, BorderLayout.NORTH );
48
49       displayArea = new JTextArea(); // cria displayArea
50       add( new JScrollPane( displayArea ), BorderLayout.CENTER );
```

```
51
52          setSize( 300, 150 ); // configura o tamanho da janela
53          setVisible( true ); // mostra a janela
54       } // fim do construtor Server
55
56       // configura e executa o servidor
57       public void runServer()
58       {
59          try // configura o servidor para receber conexões; processa as conexões
60          {
61             server = new ServerSocket( 12345, 100 ); // cria ServerSocket
62
63             while ( true )
64             {
65                try
66                {
67                   waitForConnection(); // espera uma conexão
68                   getStreams(); // obtém fluxos de entrada e saída
69                   processConnection(); // processa a conexão
70                } // fim do try
71                catch ( EOFException eofException )
72                {
73                   displayMessage( "\nServer terminated connection" );
74                } // fim do catch
75                finally
76                {
77                   closeConnection(); // fecha a conexão
78                   ++counter;
79                } // fim de finally
80             } // fim do while
81          } // fim do try
82          catch ( IOException ioException )
83          {
84             ioException.printStackTrace();
85          } // fim do catch
86       } // fim do método runServer
87
88       // espera que a conexão chegue e então exibe informações sobre a conexão
89       private void waitForConnection() throws IOException
90       {
91          displayMessage( "Waiting for connection\n" );
92          connection = server.accept(); // permite que servidor aceite a conexão
93          displayMessage( "Connection " + counter + " received from: " +
94             connection.getInetAddress().getHostName() );
95       } // fim do método waitForConnection
96
97       // obtém fluxos para enviar e receber dados
98       private void getStreams() throws IOException
99       {
100         // configura o fluxo de saída para objetos
101         output = new ObjectOutputStream( connection.getOutputStream() );
102         output.flush(); // esvazia buffer de saída para enviar as informações de cabeçalho
103
104         // configura o fluxo de entrada para objetos
105         input = new ObjectInputStream( connection.getInputStream() );
106
107         displayMessage( "\nGot I/O streams\n" );
108      } // fim do método getStreams
109
110      // processa a conexão com o cliente
111      private void processConnection() throws IOException
112      {
113         String message = "Connection successful";
114         sendData( message ); // envia uma mensagem de conexão bem-sucedida
115
116         // ativa enterField de modo que usuário do servidor possa enviar mensagens
117         setTextFieldEditable( true );
118
119         do // processa as mensagens enviadas pelo cliente
```

```
120         {
121            try // lê e exibe a mensagem
122            {
123               message = ( String ) input.readObject(); // lê uma nova mensagem
124               displayMessage( "\n" + message ); // exibe a mensagem
125            } // fim do try
126            catch ( ClassNotFoundException classNotFoundException )
127            {
128               displayMessage( "\nUnknown object type received" );
129            } // fim do catch
130
131         } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
132      } // fim do método processConnection
133
134      // fecha os fluxos e o socket
135      private void closeConnection()
136      {
137         displayMessage( "\nTerminating connection\n" );
138         setTextFieldEditable( false ); // desativa enterField
139
140         try
141         {
142            output.close(); // fecha o fluxo de saída
143            input.close(); // fecha o fluxo de entrada
144            connection.close(); // fecha o socket
145         } // fim do try
146         catch ( IOException ioException )
147         {
148            ioException.printStackTrace();
149         } // fim do catch
150      } // fim do método closeConnection
151
152      // envia a mensagem ao cliente
153      private void sendData( String message )
154      {
155         try // envia o objeto ao cliente
156         {
157            output.writeObject( "SERVER>>> " + message );
158            output.flush(); // esvazia a saída para o cliente
159            displayMessage( "\nSERVER>>> " + message );
160         } // fim do try
161         catch ( IOException ioException )
162         {
163            displayArea.append( "\nError writing object" );
164         } // fim do catch
165      } // fim do método sendData
166
167      // manipula a displayArea na thread de despacho de eventos
168      private void displayMessage( final String messageToDisplay )
169      {
170         SwingUtilities.invokeLater(
171            new Runnable()
172            {
173               public void run() // atualiza a displayArea
174               {
175                  displayArea.append( messageToDisplay ); // acrescenta a mensagem
176               } // fim do método run
177            } // fim da classe interna anônima
178         ); // fim da chamada para SwingUtilities.invokeLater
179      } // fim do método displayMessage
180
181      // manipula o enterField na thread de despacho de eventos
182      private void setTextFieldEditable( final boolean editable )
183      {
184         SwingUtilities.invokeLater(
185            new Runnable()
186            {
187               public void run() // configura a editabilidade do enterField
188               {
```

```
189                enterField.setEditable( editable );
190             } // fim do método run
191          }  // fim da classe inner
192       ); // fim da chamada para SwingUtilities.invokeLater
193    } // fim do método setTextFieldEditable
194 } // fim da classe Server
```

Figura 27.5 | A parte do servidor de uma conexão cliente/servidor de socket de fluxo.

```
1  // Figura 27.6: ServerTest.java
2  // Testa o aplicativo servidor.
3  import javax.swing.JFrame;
4
5  public class ServerTest
6  {
7     public static void main( String[] args )
8     {
9        Server application = new Server(); // cria o servidor
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       application.runServer(); // executa o aplicativo servidor
12    } // fim de main
13 } // fim da classe ServerTest
```

Figura 27.6 | Classe de texte para `Server`.

Método `runServer`

O método `runServer` (Figura 27.5, linhas 57–86) configura o servidor para receber uma conexão e processar uma conexão em um dado momento. A linha 61 cria um `ServerSocket` chamado `server` para esperar conexões. O `ServerSocket` ouve uma conexão de um cliente na porta `12345`. O segundo argumento para o construtor é o número de conexões que podem esperar em uma fila para se conectar ao servidor (`100` nesse exemplo). Se a fila estiver cheia quando um cliente tenta se conectar, o servidor recusará a conexão.

Erro comum de programação 27.1

Especificar uma porta que já está em utilização ou especificar um número de porta inválido ao criar um `ServerSocket` resulta em uma ***BindException****.*

A linha 67 chama o método `waitForConnection` (declarado nas linhas 89–95) para esperar uma conexão do cliente. Depois que a conexão for estabelecida, a linha 68 chama o método `getStreams` (declarado nas linhas 98–108) para obter referências para os fluxos dessa conexão. A linha 69 chama o método `processConnection` (declarado nas linhas 111–132) para enviar a mensagem inicial de conexão ao cliente e processar todas as mensagens recebidas dele. O bloco `finally` (linhas 75–79) termina a conexão cliente chamando o método `closeConnection` (linhas 135–150), mesmo se uma exceção ocorrer. Esses métodos chamam `displayMessage` (linhas 168–179), que utiliza a thread de despacho de eventos para exibir mensagens em `JTextArea` do aplicativo. O método **`SwingUtilities invokeLater`** recebe um objeto `Runnable` como seu argumento e insere-o na thread de despacho de eventos para execução. Isso assegura que um componente GUI não seja modificado a partir de um thread além da thread de despacho de eventos, o que é importante uma vez que componentes da GUI do Swing não são seguros para threads. Utilizamos uma técnica semelhante no método `setTextFieldEditable` (linhas 182–193), para estabelecer a editabilidade de `enterField`. Para obter informações adicionais sobre a interface `Runnable`, consulte o Capítulo 26.

Método `waitForConnection`

O método `waitForConnection` (linhas 89–95) utiliza o método `ServerSocket accept` (linha 92) para esperar uma conexão de um cliente. Quando ocorre uma conexão, o `Socket` resultante é atribuído a `connection`. O método `accept` bloqueia até que uma conexão seja recebida (isto é, a thread em que `accept` é chamado interrompe a execução até que um cliente se conecte). As linhas 93–94 geram a saída do nome do host do computador que fez a conexão. O método `Socket` **`getInetAddress`** retorna um **`InetAddress`** (pacote `java.net`) que contêm informações sobre o computador cliente. O método `InetAddress` **`getHostName`** retorna o nome do host do computador cliente. Por exemplo, há um endereço IP (**`127.0.0.1`**) e um nome de host (**`localhost`**) especial que é útil para testar aplicações de rede no seu computador local (isso também é conhecido como **endereço de loopback**). Se `getHostName` for chamado em um `InetAddress` contendo `127.0.0.1`, o nome do host correspondente retornado pelo método será `localhost`.

Método `getStreams`

O método `getStreams` (linhas 98–108) obtém os fluxos de `Socket` e utiliza-os para inicializar um `ObjectOutputStream` (linha 101) e um `ObjectInputStream` (linha 105), respectivamente. Observe a chamada ao método `ObjectOutputStream` **`flush`** na linha

102. Essa instrução faz o `ObjectOutputStream` do servidor enviar um **cabeçalho de fluxo** para o correspondente `ObjectInputStream` do cliente. O cabeçalho de fluxo contém informações como a versão de serialização de objeto sendo utilizada para enviar os objetos. Essas informações são necessárias para `ObjectInputStream` poder se preparar para receber esses objetos corretamente.

Observação de engenharia de software 27.4

Ao utilizar `ObjectOutputStream` e `ObjectInputStream` para enviar e receber dados em uma conexão de rede, sempre crie primeiro o `ObjectOutputStream` e esvazie (flush) o fluxo de modo que o `ObjectInputStream` do cliente possa se preparar para receber os dados. Isso é necessário apenas para aplicativos de rede que se comunicam utilizando `ObjectOutputStream` e `ObjectInputStream`.

Dica de desempenho 27.3

Os componentes E/S do computador são geralmente muito mais lentos do que a memória do computador. Os buffers de saída são utilizados para aumentar a eficiência de um aplicativo enviando volumes maiores de dados menos vezes, reduzindo assim o número de vezes que um aplicativo acessa os componentes E/S do computador.

Método `processConnection`

A linha 114 do método `processConnection` (linhas 111–132) chama o método `sendData` para enviar `"SERVER>>> Connection successful"` como uma `String` ao cliente. O loop nas linhas 119–131 executa até que o servidor receba a mensagem `"CLIENT>>> TERMINATE"`. A linha 123 utiliza o método `ObjectInputStream` `readObject` para ler uma `String` a partir do cliente. A linha 124 invoca método `displayMessage` para acrescentar a mensagem ao `JTextArea`.

Método `closeConnection`

Quando a transmissão estiver completa, o método `processConnection` retorna, e o programa chama o método `closeConnection` (linhas 135–150) para fechar os fluxos associados com o `Socket` e fechar o `Socket`. Então, o servidor espera a próxima tentativa de conexão de um cliente continuando com a linha 67 no começo do loop `while`.

Observe que `Server` recebe uma conexão, processa-a, fecha-a e espera a próxima conexão. Um cenário mais provável seria um `Server` que recebe uma conexão, configura essa conexão para ser processada como uma thread de execução separada e então imediatamente espera novas conexões. As threads separadas que processam conexões existentes podem continuar a executar enquanto o `Server` concentra-se em novas solicitações de conexão. Isso torna o servidor mais eficiente, porque múltiplas solicitações de clientes podem ser processadas concorrentemente. Demonstramos um servidor com múltiplas threads na Seção 27.8.

Processando interações de usuário

Quando o usuário do aplicativo servidor insere uma `String` no campo de texto e pressiona a tecla *Enter*, o programa chama o método `actionPerformed` (linhas 39–43), que lê a `String` do campo de texto e chama o método utilitário `sendData` (linhas 153–165) para enviar a `String` ao cliente. O método `sendData` grava o objeto, esvazia o buffer de saída e acrescenta a mesma `String` à área de texto na janela do servidor. Aqui não é necessário invocar `displayMessage` para modificar a área de texto, porque o método `sendData` é chamado a partir de um handler de evento — portanto, `sendData` executa como parte da thread de despacho de eventos.

Classe `Client`

Como ocorre com a classe `Server`, o construtor da classe `Client` (Figura 27.7, linhas 29–56) cria a GUI do aplicativo (um `JTextField` e uma `JTextArea`). `Client` exibe sua saída na área de texto. Quando método `main` (linhas 7–19 da Figura 27.8) executa, ele cria uma instância de classe `Client`, especifica a operação padrão de fechamento da janela e chama o método `runClient` (Figura 27.7, linhas 59–79). Neste exemplo, você pode executar o cliente a partir de qualquer computador na Internet e especificar o endereço IP ou nome de host do computador servidor como um argumento de linha de comando para o programa. Por exemplo, o comando

```
java Client 192.168.1.15
```

tenta se conectar ao `Server` no computador com o endereço IP `192.168.1.15`.

```
1  // Figura 27.7: Client.java
2  // A parte do cliente de uma conexão com socket de fluxo entre cliente e servidor.
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.net.InetAddress;
8  import java.net.Socket;
```

```
9  import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Client extends JFrame
19 {
20    private JTextField enterField; // insere informações fornecidas pelo usuário
21    private JTextArea displayArea; // exibe informações para o usuário
22    private ObjectOutputStream output; // gera o fluxo de saída para o servidor
23    private ObjectInputStream input; // gera o fluxo de entrada a partir do servidor
24    private String message = ""; // mensagem do servidor
25    private String chatServer; // servidor de host para esse aplicativo
26    private Socket client; // socket para comunicação com o servidor
27
28    // inicializa chatServer e configura a GUI
29    public Client( String host )
30    {
31       super( "Client" );
32
33       chatServer = host; // configura o servidor ao qual esse cliente se conecta
34
35       enterField = new JTextField(); // cria enterField
36       enterField.setEditable( false );
37       enterField.addActionListener(
38          new ActionListener()
39          {
40             // envia mensagem ao servidor
41             public void actionPerformed( ActionEvent event )
42             {
43                sendData( event.getActionCommand() );
44                enterField.setText( "" );
45             } // fim do método actionPerformed
46          } // fim da classe interna anônima
47       ); // fim da chamada para addActionListener
48
49       add( enterField, BorderLayout.NORTH );
50
51       displayArea = new JTextArea(); // cria displayArea
52       add( new JScrollPane( displayArea ), BorderLayout.CENTER );
53
54       setSize( 300, 150 ); // configura o tamanho da janela
55       setVisible( true ); // mostra a janela
56    } // fim do construtor Client
57
58    // conecta-se ao servidor e processa as mensagens a partir do servidor
59    public void runClient()
60    {
61       try // conecta-se ao servidor, obtém fluxos, processa a conexão
62       {
63          connectToServer(); // cria um Socket para fazer a conexão
64          getStreams(); // obtém os fluxos de entrada e saída
65          processConnection(); // processa a conexão
66       } // fim do try
67       catch ( EOFException eofException )
68       {
69          displayMessage( "\nClient terminated connection" );
70       } // fim do catch
71       catch ( IOException ioException )
72       {
73          ioException.printStackTrace();
74       } // fim do catch
75       finally
76       {
77          closeConnection(); // fecha a conexão
```

```
78          } // fim de finally
79       } // fim do método runClient
80
81       // conecta-se ao servidor
82       private void connectToServer() throws IOException
83       {
84          displayMessage( "Attempting connection\n" );
85
86          // cria Socket fazer a conexão ao servidor
87          client = new Socket( InetAddress.getByName( chatServer ), 12345 );
88
89          // exibe informações sobre a conexão
90          displayMessage( "Connected to: " +
91             client.getInetAddress().getHostName() );
92       } // fim do método connectToServer
93
94       // obtém fluxos para enviar e receber dados
95       private void getStreams() throws IOException
96       {
97          // configura o fluxo de saída para objetos
98          output = new ObjectOutputStream( client.getOutputStream() );
99          output.flush(); // esvazia buffer de saída para enviar as informações de cabeçalho
100
101         // configura o fluxo de entrada para objetos
102         input = new ObjectInputStream( client.getInputStream() );
103
104         displayMessage( "\nGot I/O streams\n" );
105      } // fim do método getStreams
106
107      // processa a conexão com o servidor
108      private void processConnection() throws IOException
109      {
110         // ativa enterField de modo que o usuário cliente possa enviar mensagens
111         setTextFieldEditable( true );
112
113         do // processa as mensagens enviadas do servidor
114         {
115            try // lê e exibe a mensagem
116            {
117               message = ( String ) input.readObject(); // lê uma nova mensagem
118               displayMessage( "\n" + message ); // exibe a mensagem
119            } // fim do try
120            catch ( ClassNotFoundException classNotFoundException )
121            {
122               displayMessage( "\nUnknown object type received" );
123            } // fim do catch
124
125         } while ( !message.equals( "SERVER>>> TERMINATE" ) );
126      } // fim do método processConnection
127
128      // fecha os fluxos e o socket
129      private void closeConnection()
130      {
131         displayMessage( "\nClosing connection" );
132         setTextFieldEditable( false ); // desativa enterField
133
134         try
135         {
136            output.close(); // fecha o fluxo de saída
137            input.close(); // fecha o fluxo de entrada 1
138            client.close(); // fecha o socket
139         } // fim do try
140         catch ( IOException ioException )
141         {
142            ioException.printStackTrace();
143         } // fim do catch
144      } // fim do método closeConnection
145
146      // envia mensagem ao servidor
```

```
147      private void sendData( String message )
148      {
149         try // envia o objeto ao servidor
150         {
151            output.writeObject( "CLIENT>>> " + message );
152            output.flush(); // esvazia os dados para saída
153            displayMessage( "\nCLIENT>>> " + message );
154         } // fim do try
155         catch ( IOException ioException )
156         {
157            displayArea.append( "\nError writing object" );
158         } // fim do catch
159      } // fim do método sendData
160
161      // manipula a displayArea na thread de despacho de eventos
162      private void displayMessage( final String messageToDisplay )
163      {
164         SwingUtilities.invokeLater(
165            new Runnable()
166            {
167               public void run() // atualiza a displayArea
168               {
169                  displayArea.append( messageToDisplay );
170               } // fim do método run
171            }  // fim da classe interna anônima
172         ); // fim da chamada para SwingUtilities.invokeLater
173      } // fim do método displayMessage
174
175      // manipula o enterField na thread de despacho de eventos
176      private void setTextFieldEditable( final boolean editable )
177      {
178         SwingUtilities.invokeLater(
179            new Runnable()
180            {
181               public void run() // configura a editabilidade do enterField
182               {
183                  enterField.setEditable( editable );
184               } // fim do método run
185            } // fim da classe interna anônima
186         ); // fim da chamada para SwingUtilities.invokeLater
187      } // fim do método setTextFieldEditable
188   } // fim da classe Client
```

Figura 27.7 | A parte do cliente de uma conexão com socket de fluxo entre cliente e um servidor.

```
1   // Figura 27.8: ClientTest.java
2   // A classe que testa o Client.
3   import javax.swing.JFrame;
4
5   public class ClientTest
6   {
7      public static void main( String[] args )
8      {
9         Client application; // declara o aplicativo cliente
10
11        // se não houver nenhum argumento de linha de comando
12        if ( args.length == 0 )
13           application = new Client( "127.0.0.1" ); // conecta-se ao host local
14        else
15           application = new Client( args[ 0 ] ); // utiliza argumentos para se conectar
16
17        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
18        application.runClient(); // executa o aplicativo cliente
19     } // fim de main
20  } // fim da classe ClientTest
```

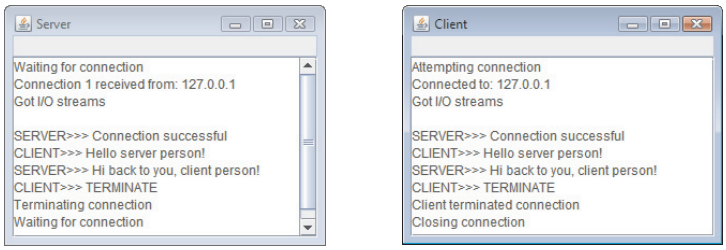


Figura 27.8 | A classe que testa o `Client`.

Método `runClient`

O método `Client` `runClient` (Figura 27.7, linhas 59–79) configura a conexão com o servidor, processa as mensagens recebidas do servidor e fecha a conexão quando a comunicação estiver completa. A linha 63 chama o método `connectToServer` (declarado nas linhas 82–92) para realizar a conexão. Depois de conectar, a linha 64 chama o método `getStreams` (declarado nas linhas 95–105) para obter referências aos objetos fluxo do `Socket`. Em seguida, a linha 65 chama o método `processConnection` (declarado nas linhas 108–126) para receber e exibir as mensagens enviadas a partir do servidor. O bloco `finally` (linhas 75–78) chama `closeConnection` (linhas 129–144) para fechar os fluxos e o `Socket` mesmo se uma exceção ocorreu. O método `displayMessage` (linhas 162–173) é chamado a partir desses métodos para utilizar a thread de despacho de eventos para exibir as mensagens na área de texto do aplicativo.

Método `connectToServer`

O método `connectToServer` (linhas 82–92) cria um `Socket` chamado `client` (linha 87) para estabelecer uma conexão. O método passa dois argumentos para o construtor `Socket` — o endereço IP do computador servidor e o número da porta (12345) em que o aplicativo servidor espera conexões de clientes. No primeiro argumento, o método `static` **`getByName`** retorna um objeto `InetAddress` contendo o endereço IP especificado como um argumento de linha de comando para o aplicativo (ou `127.0.0.1` se nenhum argumento de linha de comando for especificado). O método `getByName` pode receber uma `String` contendo tanto o endereço IP real como o nome de host do servidor. O primeiro argumento também poderia ser escrito de outras maneiras. Para o endereço `localhost` `127.0.0.1`, o primeiro argumento pode ser especificado com qualquer uma das expressões a seguir:

```
InetAddress.getByName( "localhost" )
InetAddress.getLocalHost()
```

Além disso, há versões do construtor `Socket` que recebem o endereço IP ou o nome de host como uma `String`. O primeiro argumento poderia ter sido especificado como o endereço IP `"127.0.0.1"` ou o nome de host `"localhost"`. Escolhemos demonstrar o relacionamento cliente/servidor fazendo conexões entre aplicativos no mesmo computador (`localhost`). Normalmente, esse primeiro argumento seria o endereço IP de outro computador. O objeto `InetAddress` para o outro computador pode ser obtido especificando o endereço IP do computador ou nome de host como o argumento para o método `InetAddress` `getByName`. O segundo argumento do construtor `Socket` é o número da porta do servidor. Esse número deve corresponder ao número de porta em que o servidor está esperando as conexões (chamado ponto de handshake). Depois que a conexão ocorre, as linhas 90–91 exibem uma mensagem na área de texto que indica o nome do computador servidor ao qual o cliente se conectou.

O `Client` utiliza um `ObjectOutputStream` para enviar os dados para o servidor e um `ObjectInputStream` para receber os dados do servidor. O método `getStreams` (linhas 95–105) cria os objetos `ObjectOutputStream` e `ObjectInputStream` que utilizem os fluxos associados com o socket `client`.

Métodos `processConnection` e `closeConnection`

O método `processConnection` (linhas 108–126) contém um loop que executa até que o cliente receba a mensagem `"SERVER>>> TERMINATE"`. A linha 117 lê um objeto `String` a partir do servidor. A linha 118 invoca `displayMessage` para acrescentar a mensagem à área de texto. Quando a transmissão está completa, o método `closeConnection` (linhas 129–144) fecha os fluxos e o `Socket`.

Processando interações de usuário

Quando o usuário do aplicativo cliente digita uma `String` no campo de texto e pressiona *Enter*, o programa chama o método `actionPerformed` (linhas 41–45) para ler a `String` e então invoca o método utilitário `sendData` (147–159) para enviar a `String` ao servidor. O método `sendData` grava o objeto, esvazia o buffer de saída e acrescenta a mesma `String` à `JTextArea` da janela cliente. Mais uma vez, aqui não é necessário invocar o método utilitário `displayMessage` para modificar a área de texto, porque o método `sendData` é chamado a partir de um handler de evento.

27.7 Interação cliente/servidor sem conexão com datagramas

Estávamos discutindo transmissão orientada para conexão e baseada em fluxo. Agora consideramos a **transmissão sem conexão com datagramas**.

A transmissão orientada para conexão é como o sistema de telefonia em que você disca e recebe uma conexão ao telefone da pessoa com quem você deseja se comunicar. A conexão é mantida para sua chamada telefônica, mesmo quando você não está falando.

A transmissão sem conexão com datagramas é mais parecida com a maneira como o correio é transportado via serviço postal. Se uma mensagem grande não couber em um envelope, você a dividirá em partes separadas e a colocará em envelopes numerados sequencialmente. Todas as cartas serão então remetidas de uma vez. As letras poderiam chegar na ordem, fora da ordem ou simplesmente não chegarem (este último caso é raro). A pessoa na extremidade receptora monta os pedaços em ordem sequencial antes de tentar dar sentido à mensagem.

Se a sua mensagem for suficientemente pequena para caber em um envelope, você não precisa se preocupar com o problema "fora de sequência", mas ainda é possível que sua mensagem talvez não chegue. Uma vantagem dos datagramas em relação ao correio postal é que duplicatas dos datagramas podem chegar ao computador receptor.

As figuras 27.9–27.12 utilizam datagramas para enviar pacotes de informações via User Datagram Protocol (UDP) entre um aplicativo cliente e um aplicativo servidor. No aplicativo `Client` (Figura 27.11), o usuário digita uma mensagem em um campo de texto e pressiona *Enter*. O programa converte a mensagem em um array de `bytes` e a coloca em um pacote de datagrama que é enviado para o servidor. O `Server` (figuras 27.9–27.10) recebe o pacote e exibe as informações dele e, então, **ecoa** o pacote de volta para o cliente. Ao receber o pacote, o cliente exibe as informações que ele contém.

Classe Server

A classe `Server` (Figura 27.9) declara dois **`DatagramPacket`**s que o servidor utiliza para enviar e receber informações e um **`DatagramSocket`** que envia e recebe os pacotes. O construtor (linhas 19–37), que é chamado a partir de `main` (Figura 27.10, linhas 7–12), cria a GUI no qual os pacotes de informação serão exibidos. A linha 30 cria o `DatagramSocket` em um bloco `try`. A linha 30 na Figura 27.9 utiliza o construtor `DatagramSocket` que recebe um argumento de número inteiro para a porta (nesse exemplo, `5000`) a fim de vincular o servidor a uma porta em que possa receber pacotes dos clientes. `Clients` que enviam pacotes para esse `Server` especificam o mesmo número da porta nos pacotes por eles enviados. Um **`SocketException`** é lançado se o construtor `DatagramSocket` não conseguir vincular o `DatagramSocket` à porta especificada.

Erro comum de programação 27.2

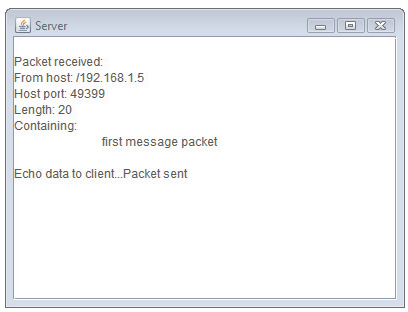
Especificar uma porta que já está em utilização ou especificar um número de porta inválido ao criar um `DatagramSocket` resulta em uma `SocketException`.

```
1  // Figura 27.9: Server.java
2  // Lado do servidor da computação cliente/servidor sem conexão com datagramas.
3  import java.io.IOException;
4  import java.net.DatagramPacket;
5  import java.net.DatagramSocket;
6  import java.net.SocketException;
7  import java.awt.BorderLayout;
8  import javax.swing.JFrame;
9  import javax.swing.JScrollPane;
10 import javax.swing.JTextArea;
11 import javax.swing.SwingUtilities;
12
13 public class Server extends JFrame
14 {
15    private JTextArea displayArea; // exibe os pacotes recebidos
16    private DatagramSocket socket; // socket para conectar ao cliente
17
18    // configura o DatagramSocket e a GUI
19    public Server()
20    {
21       super( "Server" );
22
23       displayArea = new JTextArea(); // cria displayArea
24       add( new JScrollPane( displayArea ), BorderLayout.CENTER );
25       setSize( 400, 300 ); // configura o tamanho da janela
26       setVisible( true ); // mostra a janela
27
28       try // cria DatagramSocket para envio e recebimento de pacotes
29       {
```

```
30             socket = new DatagramSocket( 5000 );
31          } // fim do try
32          catch ( SocketException socketException )
33          {
34             socketException.printStackTrace();
35             System.exit( 1 );
36          } // fim do catch
37       } // fim do construtor Server
38
39       // espera que os pacotes cheguem, exibe os dados e ecoa o pacote para o cliente
40       public void waitForPackets()
41       {
42          while ( true )
43          {
44             try // recebe o pacote, exibe o conteúdo, retorna uma cópia ao cliente
45             {
46                byte[] data = new byte[ 100 ]; // configura o pacote
47                DatagramPacket receivePacket =
48                   new DatagramPacket( data, data.length );
49
50                socket.receive( receivePacket ); // espera receber o pacote
51
52                // exibe informações a partir do pacote recebido
53                displayMessage( "\nPacket received:" +
54                   "\nFrom host: " + receivePacket.getAddress() +
55                   "\nHost port: " + receivePacket.getPort() +
56                   "\nLength: " + receivePacket.getLength() +
57                   "\nContaining:\n\t" + new String(receivePacket.getData(),
58                      0, receivePacket.getLength() ) );
59
60                sendPacketToClient( receivePacket ); // envia o pacote ao cliente
61             } // fim do try
62             catch ( IOException ioException )
63             {
64                displayMessage( ioException + "\n" );
65                ioException.printStackTrace();
66             } // fim do catch
67          } // fim do while
68       } // fim do método waitForPackets
69
70       // ecoa o pacote para o cliente
71       private void sendPacketToClient( DatagramPacket receivePacket )
72          throws IOException
73       {
74          displayMessage( "\n\nEcho data to client..." );
75
76          // cria o pacote a enviar
77          DatagramPacket sendPacket = new DatagramPacket(
78             receivePacket.getData(), receivePacket.getLength(),
79             receivePacket.getAddress(), receivePacket.getPort() );
80
81          socket.send( sendPacket ); // envia o pacote ao cliente
82          displayMessage( "Packet sent\n" );
83       } // fim do método sendPacketToClient
84
85       // manipula a displayArea na thread de despacho de eventos
86       private void displayMessage( final String messageToDisplay )
87       {
88          SwingUtilities.invokeLater(
89             new Runnable()
90             {
91                public void run() // atualiza a displayArea
92                {
93                   displayArea.append( messageToDisplay ); // exibe a mensagem
94                } // fim do método run
95             } // fim da classe interna anônima
96          ); // fim da chamada para SwingUtilities.invokeLater
97       } // fim do método displayMessage
98    } // fim da classe Server
```

Figura 27.9 | Lado do servidor da computação cliente/servidor sem conexão com datagramas.

```
1  // Figura 27.10: ServerTest.java
2  // A classe que testa o Server.
3  import javax.swing.JFrame;
4
5  public class ServerTest
6  {
7     public static void main( String[] args )
8     {
9        Server application = new Server(); // cria o servidor
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       application.waitForPackets(); // executa o aplicativo servidor
12    } // fim de main
13 } // fim da classe ServerTest
```



Janela `Server` depois que o pacote de dados foi recebido de `Client`

Figura 27.10 | A classe que testa o `Server`.

Método `waitForPackets`

O método `Server` `waitForPackets` (Figura 27.9, linhas 40–68) utiliza um loop infinito para esperar que os pacotes cheguem ao `Server`. As linhas 47–48 criam um `DatagramPacket` em que um pacote recebido de informações pode ser armazenado. O construtor `DatagramPacket` para esse propósito recebe dois argumentos — um array de `bytes` em que os dados serão armazenados e o comprimento do array. A linha 50 utiliza o método `DatagramSocket` **`receive`** para esperar que um pacote chegue ao `Server`. O método `receive` bloqueia até que um pacote chegue e então armazena esse pacote no seu argumento `DatagramPacket`. O método lança uma `IOException` se um erro ocorrer ao receber um pacote.

Método `displayMessage`

Quando um pacote chega, as linhas 53–58 chamam o método `displayMessage` (declarado nas linhas 86–97) para acrescentar o conteúdo do pacote à área de texto. O método `DatagramPacket` **`getAddress`** (linha 54) retorna um objeto `InetAddress` contendo o endereço IP do computador do qual o pacote foi enviado. O método **`getPort`** (linha 55) retorna um inteiro que especifica o número da porta pela qual o computador cliente enviou o pacote. O método **`getLength`** (linha 56) retorna um inteiro que representa o número de bytes dos dados recebidos. O método **`getData`** (linha 57) retorna um array de `bytes` contendo os dados. As linhas 57–58 inicializam um objeto `String` utilizando um construtor de três argumentos que recebe um array de `bytes`, o deslocamento e o comprimento. Essa `String` é então acrescentada ao texto a ser exibido.

Método `sendPacketToClient`

Depois de exibir um pacote, a linha 60 chama o método `sendPacketToClient` (declarado nas linhas 71–83) para criar um novo pacote e enviá-lo ao cliente. As linhas 77–79 criam um `DatagramPacket` e passam quatro argumentos para seu construtor. O primeiro argumento especifica o array de `bytes` a ser enviado. O segundo argumento especifica o número de bytes a enviar. O terceiro argumento especifica o endereço IP do computador cliente para o qual o pacote será enviado. O quarto argumento especifica a porta na qual o cliente está esperando receber os pacotes. A linha 81 envia o pacote pela rede. O método **`send`** do `DatagramSocket` lança uma `IOException` se um erro ocorrer ao enviar um pacote.

Classe `Client`

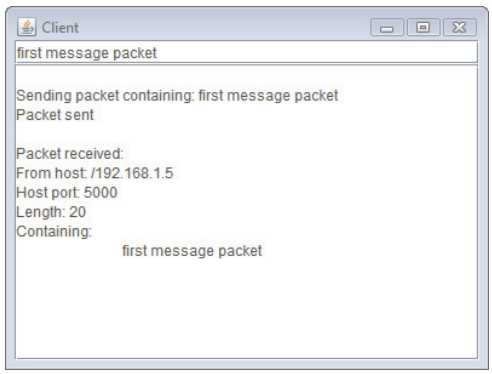
A classe `Client` (figuras 27.11–27.12) funciona de maneira semelhante à classe `Server`, exceto que `Client` envia pacotes apenas quando o usuário digita uma mensagem em um campo de texto e pressiona a tecla *Enter*. Quando isso ocorre, o programa chama o método `actionPerformed` (Figura 27.11, linhas 32–57), que converte a `String` que o usuário inseriu em um array de `bytes` (linha 41). As linhas 44–45 criam um `DatagramPacket` e o inicializam com o array de `bytes`, o comprimento da `String` que foi inserido pelo usuário, o endereço IP ao qual o pacote deve ser enviado (`InetAddress.getLocalHost()`, nesse exemplo) e o número da porta em que o `Server` espera os pacotes (5000, nesse exemplo). A linha 47 envia o pacote. Observe que o cliente nesse exemplo deve saber que o servidor está recebendo pacotes na porta 5000 — caso contrário, o servidor não receberá os pacotes.

```
1  // Figura 27.11: Client.java
2  // Lado cliente da computação cliente/servidor sem conexão com datagramas.
3  import java.io.IOException;
4  import java.net.DatagramPacket;
5  import java.net.DatagramSocket;
6  import java.net.InetAddress;
7  import java.net.SocketException;
8  import java.awt.BorderLayout;
9  import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Client extends JFrame
18 {
19    private JTextField enterField; // para inserir mensagens
20    private JTextArea displayArea; // para exibir mensagens
21    private DatagramSocket socket; // socket para conectar ao servidor
22
23    // configura o DatagramSocket e a GUI
24    public Client()
25    {
26       super( "Client" );
27
28       enterField = new JTextField( "Type message here" );
29       enterField.addActionListener(
30          new ActionListener()
31          {
32             public void actionPerformed( ActionEvent event )
33             {
34                try // cria e envia o pacote
35                {
36                   // obtém a mensagem no campo de texto
37                   String message = event.getActionCommand();
38                   displayArea.append( "\nSending packet containing: " +
39                      message + "\n" );
40
41                   byte[] data = message.getBytes(); // converte em bytes
42
43                   // cria sendPacket
44                   DatagramPacket sendPacket = new DatagramPacket( data,
45                      data.length, InetAddress.getLocalHost(), 5000 );
46
47                   socket.send( sendPacket ); // envia o pacote
48                   displayArea.append( "Packet sent\n" );
49                   displayArea.setCaretPosition(
50                      displayArea.getText().length() );
51                } // fim do try
52                catch ( IOException ioException )
53                {
54                   displayMessage( ioException + "\n" );
55                   ioException.printStackTrace();
56                } // fim do catch
57             } // fim do método actionPerformed
58          } // fim da classe inner
59       ); // fim da chamada para addActionListener
60
61       add( enterField, BorderLayout.NORTH );
62
63       displayArea = new JTextArea();
64       add( new JScrollPane( displayArea ), BorderLayout.CENTER );
65
66       setSize( 400, 300 ); // configura o tamanho da janela
67       setVisible( true ); // mostra a janela
68
69       try // cria DatagramSocket para envio e recebimento de pacotes
70       {
```

```
71            socket = new DatagramSocket();
72         } // fim do try
73         catch ( SocketException socketException )
74         {
75            socketException.printStackTrace();
76            System.exit( 1 );
77         } // fim do catch
78      } // fim do construtor Client
79
80      // espera que os pacotes cheguem do Server, exibe o conteúdo do pacote
81      public void waitForPackets()
82      {
83         while ( true )
84         {
85            try // recebe o pacote e exibe o conteúdo
86            {
87               byte[] data = new byte[ 100 ]; // configura o pacote
88               DatagramPacket receivePacket = new DatagramPacket(
89                  data, data.length );
90
91               socket.receive( receivePacket ); // espera o pacote
92
93               // exibe o conteúdo do pacote
94               displayMessage( "\nPacket received:" +
95                  "\nFrom host: " + receivePacket.getAddress() +
96                  "\nHost port: " + receivePacket.getPort() +
97                  "\nLength: " + receivePacket.getLength() +
98                  "\nContaining:\n\t" + new String(receivePacket.getData(),
99                     0, receivePacket.getLength() ) );
100           } // fim do try
101           catch ( IOException exception )
102           {
103              displayMessage( exception + "\n" );
104              exception.printStackTrace();
105           } // fim do catch
106        } // fim do while
107     } // fim do método waitForPackets
108
109     // manipula a displayArea na thread de despacho de eventos
110     private void displayMessage( final String messageToDisplay )
111     {
112        SwingUtilities.invokeLater(
113           new Runnable()
114           {
115              public void run() // atualiza a displayArea
116              {
117                 displayArea.append( messageToDisplay );
118              } // fim do método run
119           }  // fim da classe inner
120        ); // fim da chamada para SwingUtilities.invokeLater
121     } // fim do método displayMessage
122  }  // fim da classe Client
```

Figura 27.11 | Lado cliente da computação cliente/servidor sem conexão com datagramas.

```
1  // Figura 27.12: ClientTest.java
2  // Testa a classe Client.
3  import javax.swing.JFrame;
4
5  public class ClientTest
6  {
7     public static void main( String[] args )
8     {
9        Client application = new Client(); // cria o cliente
10       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       application.waitForPackets(); // executa o aplicativo cliente
12    } // fim de main
13 }  // fim da classe ClientTest
```



Janela `Client` depois de enviar pacotes para `Server` e receber pacotes de volta de `Server`

Figura 27.12 | A classe que testa o `Client`.

Observe que a chamada do construtor `DatagramSocket` (Figura 27.11, linha 71) nesse aplicativo não especifica quaisquer argumentos. Esse construtor sem argumento permite que o computador selecione o próximo número de porta disponível para o `DatagramSocket`. O cliente não precisa de um número específico de porta porque o servidor recebe o número de porta do cliente como parte de cada `DatagramPacket` enviado pelo cliente. Portanto, o servidor pode enviar pacotes de volta para o mesmo computador e número de porta do qual ele recebe um pacote de informações.

Método `waitForPackets`

O método `Client` `waitForPackets` (linhas 81–107) utiliza um loop infinito para esperar os pacotes do servidor. A linha 91 bloqueia até que um pacote chegue. Isso não impede que o usuário envie um pacote, porque os eventos GUI são tratados na thread de despacho de eventos. Apenas impede que o loop `while` continue até um pacote chegar ao `Client`. Quando um pacote chega, a linha 91 armazena esse pacote em `receivePacket` e as linhas 94–99 chamam o método `displayMessage` (declarado nas linhas 110–121) para exibir o conteúdo do pacote na área de texto.

27.8 Jogo da velha cliente/servidor que utiliza um servidor com multithread

Esta seção apresenta o popular jogo Tic-Tac-Toe, ou jogo da velha, implementado utilizando-se as técnicas cliente/servidor com sockets de fluxo. O programa consiste em um aplicativo `TicTacToeServer` (figuras 27.13–27.14) que permite que dois aplicativos `TicTacToeClient` (figuras 27.15–27.16) se conectem ao servidor e joguem esse jogo. As saídas de exemplo são mostradas na Figura 27.17.

Classe `TicTacToeServer`

À medida que a classe `TicTacToeServer` recebe cada conexão de cliente, ela cria uma instância da classe interna `Player` (Figura 27.13, linhas 182–304) para processar o cliente em uma thread separada. Essas threads permitem que os clientes joguem o jogo independentemente. O primeiro cliente a se conectar ao servidor é o jogador X e o segundo é o jogador O. O jogador X faz a primeira jogada. O servidor mantém as informações sobre o tabuleiro de modo que possa determinar se uma jogada é válida.

```
1  // Figura 27.13: TicTacToeServer.java
2  // Lado do servidor do programa Tic-Tac-Toe cliente/servidor.
3  import java.awt.BorderLayout;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.io.IOException;
7  import java.util.Formatter;
8  import java.util.Scanner;
9  import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13 import java.util.concurrent.locks.Condition;
14 import javax.swing.JFrame;
15 import javax.swing.JTextArea;
16 import javax.swing.SwingUtilities;
17
18 public class TicTacToeServer extends JFrame
19 {
20    private String[] board = new String[ 9 ]; // tabuleiro do jogo da velha
21    private JTextArea outputArea; // para gerar saída das jogadas
```

```
22       private Player[] players; // array de Players
23       private ServerSocket server; // socket de servidor para conectar com clientes
24       private int currentPlayer; // monitora o jogador com a jogada atual
25       private final static int PLAYER_X = 0; // constante para o primeiro jogador
26       private final static int PLAYER_O = 1; // constante para o segundo jogador
27       private final static String[] MARKS = { "X", "O" }; // array de marcas
28       private ExecutorService runGame; // executará os jogadores
29       private Lock gameLock; // para bloquear a sincronização do jogo
30       private Condition otherPlayerConnected; // para esperar outro jogador
31       private Condition otherPlayerTurn; // para esperar a jogada do outro jogador
32
33       // configura o servidor de tic-tac-toe e a GUI que exibe as mensagens
34       public TicTacToeServer()
35       {
36          super( "Tic-Tac-Toe Server" ); // configura o título da janela
37
38          // cria ExecutorService com uma thread para cada jogador
39          runGame = Executors.newFixedThreadPool( 2 );
40          gameLock = new ReentrantLock(); // cria um bloqueio para o jogo
41
42          // variável de condição para os dois jogadores sendo conectados
43          otherPlayerConnected = gameLock.newCondition();
44
45          // variável de condição para a jogada do outro jogador
46          otherPlayerTurn = gameLock.newCondition();
47
48          for ( int i = 0; i < 9; i++ )
49             board[ i ] = new String( "" ); // cria o tabuleiro de jogo da velha
50          players = new Player[ 2 ]; // cria array de jogadores
51          currentPlayer = PLAYER_X; // configura o jogador atual como o primeiro jogador
52
53          try
54          {
55             server = new ServerSocket( 12345, 2 ); // configura ServerSocket
56          } // fim do try
57          catch ( IOException ioException )
58          {
59             ioException.printStackTrace();
60             System.exit( 1 );
61          } // fim do catch
62
63          outputArea = new JTextArea(); // cria JTextArea para saída
64          add( outputArea, BorderLayout.CENTER );
65          outputArea.setText( "Server awaiting connections\n" );
66
67          setSize( 300, 300 ); // configura o tamanho da janela
68          setVisible( true ); // mostra a janela
69       } // fim do construtor TicTacToeServer
70
71       // espera duas conexões para que o jogo possa ser jogado
72       public void execute()
73       {
74          // espera que cada cliente se conecte
75          for ( int i = 0; i < players.length; i++ )
76          {
77             try // espera a conexão, cria Player, inicia o executável
78             {
79                players[ i ] = new Player( server.accept(), i );
80                runGame.execute( players[ i ] ); // executa o executável de jogador
81             } // fim do try
82             catch ( IOException ioException )
83             {
84                ioException.printStackTrace();
85                System.exit( 1 );
86             } // fim do catch
87          } // for final
88
89          gameLock.lock(); // bloqueia o jogo para sinalizar a thread do jogador X
90
```

```
91          try
92          {
93             players[ PLAYER_X ].setSuspended( false ); // retoma o jogador X
94             otherPlayerConnected.signal(); // acorda a thread do jogador X
95          } // fim do try
96          finally
97          {
98             gameLock.unlock(); // desbloqueia o jogo depois de sinalizar para o jogador X
99          } // fim de finally
100      } // fim do método execute
101
102      // exibe uma mensagem na outputArea
103      private void displayMessage( final String messageToDisplay )
104      {
105         // exibe uma mensagem a partir da thread de despacho de eventos da execução
106         SwingUtilities.invokeLater(
107            new Runnable()
108            {
109               public void run() // atualiza a outputArea
110               {
111                  outputArea.append( messageToDisplay ); // adiciona mensagem
112               } // fim do método run
113            } // fim da classe inner
114         ); // fim da chamada para SwingUtilities.invokeLater
115      } // fim do método displayMessage
116
117      // determina se a jogada é válida
118      public boolean validateAndMove( int location, int player )
119      {
120         // enquanto não for o jogador atual, deve esperar a jogada
121         while ( player != currentPlayer )
122         {
123            gameLock.lock(); // bloqueia o jogo para que o outro jogador prossiga
124
125            try
126            {
127               otherPlayerTurn.await(); // espera a jogada do jogador
128            } // fim do try
129            catch ( InterruptedException exception )
130            {
131               exception.printStackTrace();
132            } // fim do catch
133            finally
134            {
135               gameLock.unlock(); // desbloqueia o jogo depois de esperar
136            } // fim de finally
137         } // fim do while
138
139         // se a posição não estiver ocupada, faz a jogada
140         if ( !isOccupied( location ) )
141         {
142            board[ location ] = MARKS[ currentPlayer ]; // configura uma jogada no tabuleiro
143            currentPlayer = ( currentPlayer + 1 ) % 2; // troca o jogador
144
145            // deixa que novo jogador atual saiba que a jogada ocorreu
146            players[ currentPlayer ].otherPlayerMoved( location );
147
148            gameLock.lock(); // bloqueia o jogo para sinalizar ao outro jogador a prosseguir
149
150            try
151            {
152               otherPlayerTurn.signal(); // sinaliza que o outro jogador continue
153            } // fim do try
154            finally
155            {
156               gameLock.unlock(); // desbloqueia o jogo depois de sinalizar
157            } // fim de finally
158
159            return true; // notifica o jogador que a jogada foi válida
```

```
160          } // fim do if
161          else // a jogada não foi válida
162             return false; // notifica o jogador que a jogada foi inválida
163       } // fim do método validateAndMove
164
165       // determina se a posição está ocupada
166       public boolean isOccupied( int location )
167       {
168          if ( board[ location ].equals( MARKS[ PLAYER_X ] ) ||
169             board [ location ].equals( MARKS[ PLAYER_O ] ) )
170             return true; // posição está ocupada
171          else
172             return false; // posição não está ocupada
173       } // fim do método isOccupied
174
175       // coloque o código nesse método para determinar se o jogo terminou
176       public boolean isGameOver()
177       {
178          return false; // isso é deixado como exercício
179       } // fim do método isGameOver
180
181       // classe interna privada Player gerencia cada Player como um executável
182       private class Player implements Runnable
183       {
184          private Socket connection; // conexão com o cliente
185          private Scanner input; // entrada do cliente
186          private Formatter output; // saída para o cliente
187          private int playerNumber; // monitora qual é o jogador
188          private String mark; // marca para esse jogador
189          private boolean suspended = true; // se a thread está suspensa
190
191          // configura a thread Player
192          public Player( Socket socket, int number )
193          {
194             playerNumber = number; // armazena o número desse jogador
195             mark = MARKS[ playerNumber ]; // especifica a marca do jogador
196             connection = socket; // armazena o socket para o cliente
197
198             try // obtém fluxos a partir de Socket
199             {
200                input = new Scanner( connection.getInputStream() );
201                output = new Formatter( connection.getOutputStream() );
202             } // fim do try
203             catch ( IOException ioException )
204             {
205                ioException.printStackTrace();
206                System.exit( 1 );
207             } // fim do catch
208          } // fim do construtor Player
209
210          // envia uma mensagem de que o outro jogador fez uma jogada
211          public void otherPlayerMoved( int location )
212          {
213             output.format( "Opponent moved\n" );
214             output.format( "%d\n", location ); // envia a posição da jogada
215             output.flush(); // esvazia a saída
216          } // fim do método otherPlayerMoved
217
218          // execução da thread de controle
219          public void run()
220          {
221             // envia ao cliente a marca (X ou O), processa as mensagens do cliente
222             try
223             {
224                displayMessage( "Player " + mark + " connected\n" );
225                output.format( "%s\n", mark ); // envia a marca do jogador
226                output.flush(); // esvazia a saída
227
228                // se for o jogador X, espera que o outro jogador chegue
```

```
229             if ( playerNumber == PLAYER_X )
230             {
231                output.format( "%s\n%s", "Player X connected",
232                   "Waiting for another player\n" );
233                output.flush(); // esvazia a saída
234
235                gameLock.lock(); // bloqueia o jogo para esperar o segundo jogador
236
237                try
238                {
239                   while( suspended )
240                   {
241                      otherPlayerConnected.await(); // espera o jogador O
242                   } // fim do while
243                } // fim do try
244                catch ( InterruptedException exception )
245                {
246                   exception.printStackTrace();
247                } // fim do catch
248                finally
249                {
250                   gameLock.unlock(); // desbloqueia o jogo depois do segundo jogador
251                } // fim de finally
252
253                // envia uma mensagem de que o outro jogador se conectou
254                output.format( "Other player connected. Your move.\n" );
255                output.flush(); // esvazia a saída
256             } // fim do if
257             else
258             {
259                output.format( "Player O connected, please wait\n" );
260                output.flush(); // esvazia a saída
261             } // fim de else
262
263             // enquanto jogo não terminou
264             while ( !isGameOver() )
265             {
266                int location = 0; // inicializa a posição da jogada
267
268                if ( input.hasNext() )
269                   location = input.nextInt(); // obtém a posição da jogada
270
271                // verifica uma jogada válida
272                if ( validateAndMove( location, playerNumber ) )
273                {
274                   displayMessage( "\nlocation: " + location );
275                   output.format( "Valid move.\n" ); // notifica o cliente
276                   output.flush(); // esvazia a saída
277                } // fim do if
278                else // jogada foi inválida
279                {
280                   output.format( "Invalid move, try again\n" );
281                   output.flush(); // esvazia a saída
282                } // fim de else
283             } // fim do while
284          } // fim do try
285          finally
286          {
287             try
288             {
289                connection.close(); // fecha a conexão com o cliente
290             } // fim do try
291             catch ( IOException ioException )
292             {
293                ioException.printStackTrace();
294                System.exit( 1 );
295             } // fim do catch
296          } // fim de finally
297       } // fim do método run
```

```
298
299         // configura se a thread está ou não suspensa
300         public void setSuspended( boolean status )
301         {
302            suspended = status; // configura o valor do suspenso
303         } // fim do método setSuspended
304      } // fim da classe Player
305   } // fim da classe TicTacToeServer
```

Figura 27.13 | O lado do servidor do programa Tic-Tac-Toe cliente/servidor.

```
1   // Figura 27.14: TicTacToeServerTest.java
2   // Classe que testa o servidor Tic-Tac-Toe.
3   import javax.swing.JFrame;
4
5   public class TicTacToeServerTest
6   {
7      public static void main( String[] args )
8      {
9         TicTacToeServer application = new TicTacToeServer();
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        application.execute();
12     } // fim de main
13  } // fim da classe TicTacToeServerTest
```

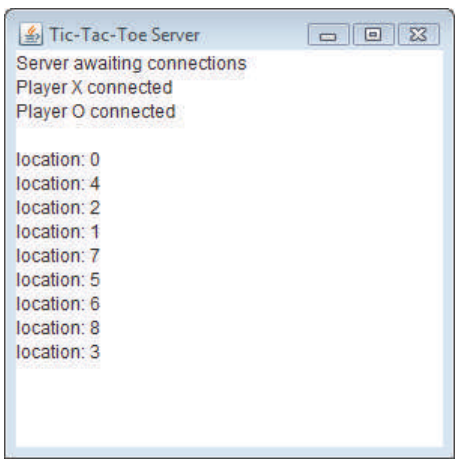


Figura 27.14 | A classe que testa o servidor Tic-Tac-Toe.

Começamos com uma discussão do lado do servidor do jogo Tic-Tac-Toe. Quando o aplicativo `TicTacToeServer` executa, o método `main` (linhas 7–12 da Figura 27.14) cria um objeto TicTacToeServer chamado `application`. O construtor (Figura 27.13, linhas 34–69) tenta configurar um `ServerSocket`. Se bem-sucedido, o programa exibe a janela de servidor e então `main` invoca o método `TicTacToeServer execute` (linhas 72–100). O método `execute` faz um loop duas vezes, bloqueando na linha 79 toda vez enquanto espera a conexão de um cliente. Quando um cliente se conecta, a linha 79 cria um novo objeto `Player` para gerenciar a conexão como uma thread separada e a linha 80 executa o `Player` no pool de threads `runGame`.

Quando o `TicTacToeServer` cria um `Player`, o construtor `Player` (linhas 192–208) recebe o objeto `Socket` que representa a conexão com o cliente e obtém os fluxos de entrada e de saída associados. A linha 201 cria um `Formatter` (ver o Capítulo 17) empacotando-o no fluxo de saída do socket. O método `Player run` (linhas 219–297) controla as informações enviadas e recebidas do cliente. Primeiro, ele passa para o cliente o caractere que o cliente colocará no tabuleiro quando ocorre uma jogada (linha 225). A linha 226 chama o método `Formatter` **flush** para forçar essa saída para o cliente. A linha 241 suspende a thread do jogador X à medida que ele inicia a execução, porque o jogador X só pode jogar depois que jogador O se conectar.

Quando o jogador O se conectar, o jogo poderá ser jogado e o método `run` iniciará a execução da sua instrução `while` (linhas 264–283). Cada iteração desse loop lê um inteiro (linha 269) que representa a posição em que o cliente quer colocar uma marca (bloqueando para esperar uma entrada, se necessário) e a linha 272 invoca o método `TicTacToeServer validateAndMove` (declarado nas linhas 118–163) para verificar a jogada. Se a jogada for válida, a linha 275 enviará uma mensagem ao cliente para esse efeito. Se não, a linha 280 envia uma mensagem que indica que a jogada foi inválida. O programa mantém as localizações do tabuleiro como números de 0 a 8 (0 a 2 para a primeira linha, 3 a 5 para a segunda linha e 6 a 8 para a terceira linha).

O método `validateAndMove` (linhas 118–163 na classe `TicTacToeServer`) permite que apenas um jogador faça uma jogada por vez, impedindo assim que eles modifiquem as informações sobre o estado do jogo simultaneamente. Se o `Player` que tenta validar uma jogada não for o jogador atual (isto é, aquele com permissão para fazer uma jogada), ele é colocado em um estado de *espera* (*wait*) até chegar sua vez de jogar. Se a posição para o movimento sendo validado já estiver ocupada no tabuleiro, `validMove` retorna `false`. Caso contrário, o servidor coloca uma marca para o jogador na sua representação local do tabuleiro (linha 142), notifica o outro objeto `Player` (linha 146) de que uma jogada foi feita (de modo que seja possível enviar uma mensagem ao cliente), invoca o método `signal` (linha 152) para que o `Player` esperando (se houver um) possa validar uma jogada e retorna `true` (linha 159) para indicar que a jogada é válida.

Classe `TicTacToeClient`

Cada aplicativo `TicTacToeClient` (figuras 27.15–27.16; saídas de exemplo na Figura 27.17) mantém sua própria versão da GUI do tabuleiro do Tic-Tac-Toe em que exibe o estado do jogo. Os clientes somente podem colocar uma marca em um quadrado vazio no tabuleiro. A classe interna `Square` (Figura 27.15, linhas 205–261) implementa cada um dos nove quadrados do tabuleiro. Quando um `TicTacToeClient` inicia a execução, ele cria uma `JTextArea` em que as mensagens do servidor e uma representação do tabuleiro que utilizam nove objetos `Square` são exibidas. O método `startClient` (linhas 80–100) abre uma conexão com o servidor e obtém os fluxos de entrada e de saída associados ao objeto `Socket`. As linhas 85–86 fazem uma conexão com o servidor. A classe `TicTacToeClient` implementa a interface `Runnable` de modo que uma thread separada possa ler as mensagens a partir do servidor. Essa abordagem permite ao usuário interagir com o tabuleiro (na thread de despacho de eventos) enquanto espera as mensagens do servidor. Depois de estabelecer a conexão com o servidor, a linha 99 executa o cliente com o `worker ExecutorService`. O método `run` (linhas 103–126) controla a thread separada de execução. O método primeiro lê o caractere de marcação (X ou O) do servidor (linha 105) e então repete um loop continuamente (linhas 121–125) e lê as mensagens do servidor (linha 124). Cada mensagem é passada para o método `processMessage` (linhas 129–156) para processamento.

```
1  // Figura 27.15: TicTacToeClient.java
2  // Lado do cliente de programa cliente/servidor Tic-Tac-Toe.
3  import java.awt.BorderLayout;
4  import java.awt.Dimension;
5  import java.awt.Graphics;
6  import java.awt.GridLayout;
7  import java.awt.event.MouseAdapter;
8  import java.awt.event.MouseEvent;
9  import java.net.Socket;
10 import java.net.InetAddress;
11 import java.io.IOException;
12 import javax.swing.JFrame;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18 import java.util.Formatter;
19 import java.util.Scanner;
20 import java.util.concurrent.Executors;
21 import java.util.concurrent.ExecutorService;
22
23 public class TicTacToeClient extends JFrame implements Runnable
24 {
25    private JTextField idField; // campo de texto para exibir a marca do jogador
26    private JTextArea displayArea; // JTextArea para exibir a saída
27    private JPanel boardPanel; // painel para o tabuleiro do jogo da velha
28    private JPanel panel2; // painel para conter o tabuleiro
29    private Square[][] board; // tabuleiro do jogo da velha
30    private Square currentSquare; // quadrado atual
31    private Socket connection; // conexão com o servidor
32    private Scanner input; // entrada a partir do servidor
33    private Formatter output; // saída para o servidor
34    private String ticTacToeHost; // nome do host para o servidor
35    private String myMark; // marca desse cliente
36    private boolean myTurn; // determina de qual cliente é a vez
37    private final String X_MARK = "X"; // marca para o primeiro cliente
38    private final String O_MARK = "O"; // marca para o segundo cliente
39
40    // configura a interface com o usuário e o tabuleiro
41    public TicTacToeClient( String host )
42    {
```

```
43          ticTacToeHost = host; // configura o nome do servidor
44          displayArea = new JTextArea( 4, 30 ); // configura JTextArea
45          displayArea.setEditable( false );
46          add( new JScrollPane( displayArea ), BorderLayout.SOUTH );
47
48          boardPanel = new JPanel(); // configura o painel para os quadrados no tabuleiro
49          boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
50
51          board = new Square[ 3 ][ 3 ]; // cria o tabuleiro
52
53          // faz um loop pelas linhas no tabuleiro
54          for ( int row = 0; row < board.length; row++ )
55          {
56             // faz um loop pelas colunas no tabuleiro
57             for ( int column = 0; column < board[ row ].length; column++ )
58             {
59                // cria um quadrado
60                board[ row ][ column ] = new Square( ' ', row * 3 + column );
61                boardPanel.add( board[ row ][ column ] ); // adiciona um quadrado
62             } // fim do for interno
63          } // fim do for externo
64
65          idField = new JTextField(); // configura o campo de texto
66          idField.setEditable( false );
67          add( idField, BorderLayout.NORTH );
68
69          panel2 = new JPanel(); // configure o painel que irá conter o boardPanel
70          panel2.add( boardPanel, BorderLayout.CENTER ); // adiciona o painel do tabuleiro
71          add( panel2, BorderLayout.CENTER ); // adiciona o painel contêiner
72
73          setSize( 300, 225 ); // configura o tamanho da janela
74          setVisible( true ); // mostra a janela
75
76          startClient();
77       } // fim do construtor TicTacToeClient
78
79       // inicia a thread do cliente
80       public void startClient()
81       {
82          try // conecta-se ao servidor e obtém fluxos
83          {
84             // faz uma conexão com o servidor
85             connection = new Socket(
86                InetAddress.getByName( ticTacToeHost ), 12345 );
87
88             // obtém os fluxos de entrada e saída
89             input = new Scanner( connection.getInputStream() );
90             output = new Formatter( connection.getOutputStream() );
91          } // fim do try
92          catch ( IOException ioException )
93          {
94             ioException.printStackTrace();
95          } // fim do catch
96
97          // cria e inicia a thread de trabalhador para esse cliente
98          ExecutorService worker = Executors.newFixedThreadPool( 1 );
99          worker.execute( this ); // executa o cliente
100      } // fim do método startClient
101
102      // thread de controle que permite atualização contínua da displayArea
103      public void run()
104      {
105         myMark = input.nextLine(); // obtém a marca do jogador (X ou O)
106
107         SwingUtilities.invokeLater(
108            new Runnable()
109            {
110               public void run()
111               {
```

```
112                // exibe a marca do jogador
113                idField.setText( "You are player \"" + myMark + "\"" );
114             } // fim do método run
115          } // fim da classe interna anônima
116       ); // fim da chamada para SwingUtilities.invokeLater
117
118       myTurn = ( myMark.equals( X_MARK ) ); // determina se a vez do cliente
119
120       // recebe as mensagens enviadas para o cliente e gera saída delas
121       while ( true )
122       {
123          if ( input.hasNextLine() )
124             processMessage(input.nextLine() );
125       } // fim do while
126    } // fim do método run
127
128    // processa as mensagens recebidas pelo cliente
129    private void processMessage( String message )
130    {
131       // ocorreu uma jogada válida
132       if ( message.equals( "Valid move." ) )
133       {
134          displayMessage( "Valid move, please wait.\n" );
135          setMark( currentSquare, myMark ); // configura a marca no quadrado
136       } // fim do if
137       else if ( message.equals( "Invalid move, try again" ) )
138       {
139          displayMessage( message + "\n" ); // exibe jogada inválida
140          myTurn = true; // ainda é a vez desse cliente
141       } // fim de else if
142       else if ( message.equals( "Opponent moved" ) )
143       {
144          int location = input.nextInt(); // obtém a posição da jogada
145          input.nextLine(); // pula uma nova linha depois da posição de int
146          int row = location / 3; // calcula a linha
147          int column = location % 3; // calcula a coluna
148
149          setMark(  board[ row ][ column ],
150             ( myMark.equals( X_MARK ) ? O_MARK : X_MARK ) ); // marca a jogada
151          displayMessage( "Opponent moved. Your turn.\n" );
152          myTurn = true; // agora é a vez desse cliente
153       } // fim de else if
154       else
155          displayMessage( message + "\n" ); // exibe a mensagem
156    } // fim do método processMessage
157
158    // manipula displayArea na thread de despacho de eventos
159    private void displayMessage( final String messageToDisplay )
160    {
161       SwingUtilities.invokeLater(
162          new Runnable()
163          {
164             public void run()
165             {
166                displayArea.append( messageToDisplay ); // atualiza a saída
167             } // fim do método run
168          }  // fim da classe inner
169       ); // fim da chamada para SwingUtilities.invokeLater
170    } // fim do método displayMessage
171
172    // método utilitário para configurar a marca sobre o tabuleiro na thread de despacho de eventos
173    private void setMark( final Square squareToMark, final String mark )
174    {
175       SwingUtilities.invokeLater(
176          new Runnable()
177          {
178             public void run()
```

```
179                {
180                   squareToMark.setMark( mark ); // configura a marca no quadrado
181                } // fim do método run
182             } // fim da classe interna anônima
183          ); // fim da chamada para SwingUtilities.invokeLater
184       } // fim do método setMark
185
186       // envia mensagem para o servidor indicando o quadrado clicado
187       public void sendClickedSquare( int location )
188       {
189          // se for minha vez
190          if ( myTurn )
191          {
192             output.format( "%d\n", location ); // envia a posição ao servidor
193             output.flush();
194             myTurn = false; // não é mais a minha vez
195          } // fim do if
196       } // fim do método sendClickedSquare
197
198       // configura o Squareatual
199       public void setCurrentSquare( Square square )
200       {
201          currentSquare = square; // configura o quadrado atual para o argumento
202       } // fim do método setCurrentSquare
203
204       // classe interna privada para os quadrados no tabuleiro
205       private class Square extends JPanel
206       {
207          private String mark; // marca a ser desenhada nesse quadrado
208          private int location; // posição do quadrado
209
210          public Square( String squareMark, int squareLocation )
211          {
212             mark = squareMark; // configura a marca para esse quadrado
213             location = squareLocation; // configura a posição desse quadrado
214
215             addMouseListener(
216                new MouseAdapter()
217                {
218                   public void mouseReleased( MouseEvent e )
219                   {
220                      setCurrentSquare( Square.this ); // configura o quadrado atual
221
222                      // envia a posição desse quadrado
223                      sendClickedSquare( getSquareLocation() );
224                   } // fim do método mouseReleased
225                } // fim da classe interna anônima
226             ); // fim da chamada para addMouseListener
227          } // fim do construtor Square
228
229          // retorna o tamanho preferido de Square
230          public Dimension getPreferredSize()
231          {
232             return new Dimension( 30, 30 ); // retorna o tamanho preferido
233          } // fim do método getPreferredSize
234
235          // retorna o tamanho mínimo de Square
236          public Dimension getMinimumSize()
237          {
238             return getPreferredSize(); // retorna o tamanho preferido
239          } // fim do método getMinimumSize
240
241          // configura a marca para Square
242          public void setMark( String newMark )
243          {
244             mark = newMark; // configura a marca do quadrado
245             repaint(); // repinta o quadrado
```

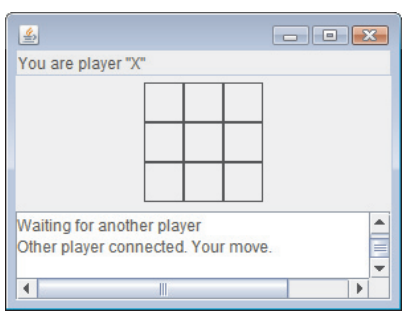
```
246         } // fim do método setMark
247
248         // retorna a posição de Square
249         public int getSquareLocation()
250         {
251            return location; // retorna a posição do quadrado
252         } // fim do método getSquareLocation
253
254         // desenha Square
255         public void paintComponent( Graphics g )
256         {
257            super.paintComponent( g );
258
259            g.drawRect( 0, 0, 29, 29 ); // desenha o quadrado
260            g.drawString( mark, 11, 20 ); // desenha a marca
261         } // fim do método paintComponent
262      } // fim da classe interna Square
263   } // fim da classe TicTacToeClient
```

Figura 27.15 | O lado do cliente de programa cliente/servidor Tic-Tac-Toe.

```
1   // Figura 27.16: TicTacToeClientTest.java
2   // Testa a classe para o cliente de Tic-Tac-Toe.
3   import javax.swing.JFrame;
4
5   public class TicTacToeClientTest
6   {
7      public static void main( String[] args )
8      {
9         TicTacToeClient application; // declara o aplicativo cliente
10
11        // se não houver nenhum argumento de linha de comando
12        if ( args.length == 0 )
13           application = new TicTacToeClient( "127.0.0.1" ); // host local
14        else
15           application = new TicTacToeClient( args[ 0 ] ); // utiliza argumentos
16
17        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
18     } // fim de main
19  } // fim da classe TicTacToeClientTest
```

Figura 27.16 | Testa a classe para o cliente de Tic-Tac-Toe.

Se a mensagem recebida for `"Valid move."`, as linhas 134–135 exibem a mensagem `"Valid move, please wait."` e chamam o método `setMark` (linhas 173–184) para configurar a marca do cliente no quadrado atual (aquele em que o usuário clicou), utilizando o método `SwingUtilities` `invokeLater` para assegurar que as atualizações das GUIs ocorram na thread de despacho de eventos. Se a mensagem recebida for `"Invalid move, try again."`, a linha 139 exibe a mensagem de modo que o usuário possa clicar em um quadrado diferente. Se a mensagem recebida for `"Opponent moved."`, a linha 144 lê um inteiro a partir do servidor indicando onde o oponente jogou e as linhas 149–150 colocam uma marca nesse quadrado do tabuleiro (novamente utilizando o método `SwingUtilities` `invokeLater` para assegurar que as atualizações das GUIs ocorram na thread de despacho de eventos). Se qualquer outra mensagem for recebida, a linha 155 simplesmente exibirá a mensagem.



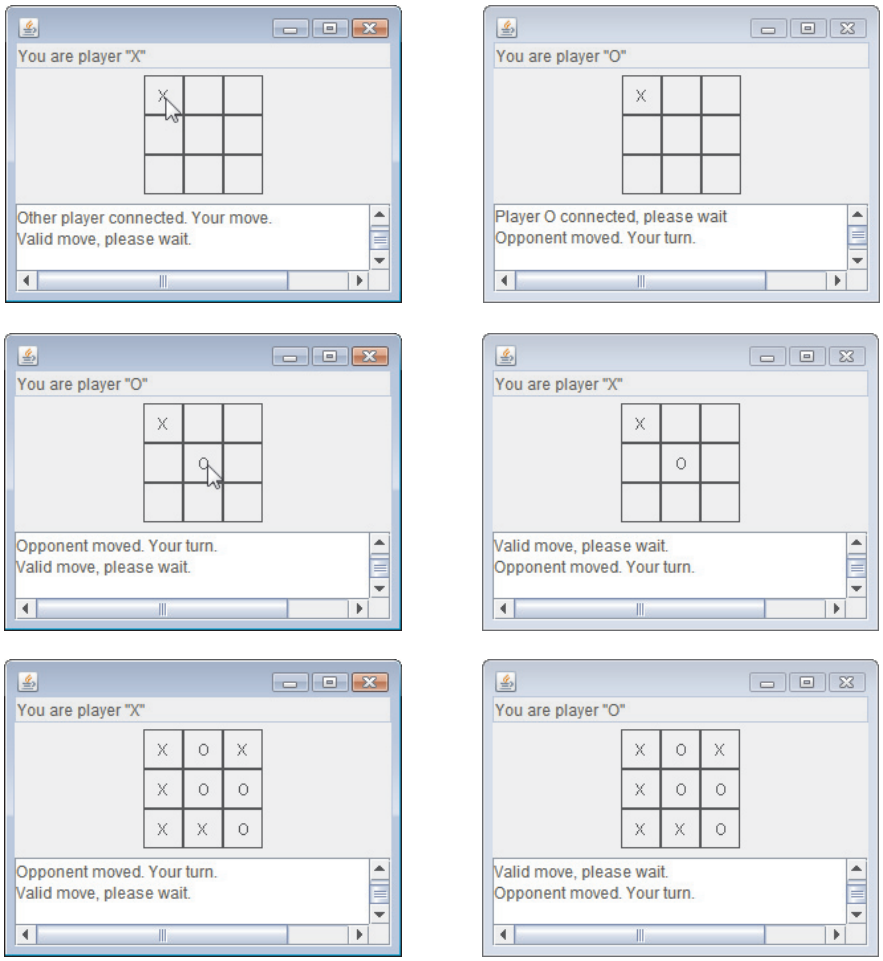


Figura 27.17 | Saídas de exemplo do programa cliente/servidor Tic-Tac-Toe.

27.9 [Bônus Web] Estudo de caso: Servidor e cliente `DeitelMessenger`

[*Nota*: Esse estudo de caso está disponível em inglês em `www.deitel.com/books/jhtp8/`.] Salas de chat são comuns na internet. Elas fornecem um local central em que os usuários podem conversar uns com os outros via mensagens curtas de texto. Cada participante pode ver todas as mensagens que os outros usuários postaram e cada usuário pode postar mensagens. Esse estudo de caso integra muitos dos recursos de rede e multithread do Java e da GUI do Swing que você aprendeu até agora para construir um sistema de chat on-line. Também introduzimos o **multicasting**, que permite a um aplicativo enviar `DatagramPackets` para grupos de clientes.

O estudo de caso `DeitelMessenger` é um aplicativo significativo que utiliza muitos recursos Java intermediários, como redes com `Sockets`, `DatagramPackets` e `MulticastSockets`, multithreading e GUI Swing. O estudo de caso também demonstra boas práticas de engenharia de software separando a interface da implementação, o que permite aos desenvolvedores dar suporte a diferentes protocolos de rede e fornecer diferentes interfaces com usuário. Depois de ler esse estudo de caso, você será capaz de construir aplicativos de rede mais significativos.

27.10 Conclusão

Neste capítulo, você aprendeu os princípios básicos da programação de rede em Java. Começamos com um applet e um aplicativo simples em que o Java cuidava das questões de rede para você. Aprendeu dois métodos diferentes de enviar dados por uma rede — rede baseada em fluxos utilizando TCP/IP e rede baseada em datagramas utilizando UDP. Mostramos como construir programas de chat cliente/servidor simples utilizando redes baseadas em fluxos e redes baseadas em datagramas. Vimos então um jogo cliente/servidor do tipo jogo da velha chamado Tic-Tac-Toe que permite a dois clientes jogar interagindo com um servidor baseado em múltiplas threads que mantém a lógica e o estado do jogo. No próximo capítulo, você aprenderá conceitos básicos sobre banco de dados, como interagir com os dados em um banco de dados utilizando o SQL e como utilizar o JDBC para permitir que aplicativos Java manipulem os dados no banco de dados.

Resumo

Seção 27.1 Introdução

- O Java fornece sockets de fluxo e sockets de datagrama. Com sockets de fluxo, um processo estabelece uma conexão com outro processo. Enquanto a conexão estiver no ar, os dados fluem entre os processos em fluxos contínuos. Dizemos que os sockets de fluxo fornecem um serviço orientado para conexão. O protocolo utilizado para transmissão é o popular TCP (Transmission Control Protocol).
- Com sockets de datagrama, são transmitidos pacotes individuais de informações. O UDP (User Datagram Protocol) é um serviço sem conexão que não garante que os pacotes não serão perdidos, duplicados ou que cheguem fora de sequência.

Seção 27.2 Manipulando URLs

- O protocolo HTTP (HyperText Transfer Protocol), que forma a base da Web, usa URIs (Uniform Resource Identifier) para localizar dados na Internet. Os URIs comuns representam arquivos ou diretórios e podem representar tarefas complexas como pesquisas de banco de dados e pesquisas de Internet. Um URI que representa um documento é chamado URL (Uniform Resource Locator).
- O método `applet` `getAppletContext` retorna um `AppletContext` que representa o navegador em que o applet está executando. O método `AppletContext` `showDocument` recebe um `URL` e o passa para o `AppletContext`, que exibe o recurso Web correspondente. Uma segunda versão de `showDocument` permite que um applet especifique o frame-alvo em que será exibido um recurso Web.

Seção 27.3 Lendo um arquivo em um servidor da Web

- O método `JEditorPane` `setPage` baixa o documento especificado pelo seu argumento e o exibe.
- Em geral, um documento XHTML contém hiperlinks que, quando clicados, levam o usuário para outros documentos na Web. Se um documento XHTML for exibido em uma `JEditorPane` e o usuário clicar em um hiperlink, o `JEditorPane` gera um hiperlink e notifica seu `HyperlinkListeners`.
- O método `HyperlinkEvent` `getEventType` determina o tipo de evento. `HyperlinkEvent` contém a classe aninhada `EventType`, que declara três tipos de evento de hiperlink: `ACTIVATED` (hiperlink clicado), `ENTERED` (mouse sobre um hiperlink) e `EXITED` (mouse saiu de cima de um hiperlink). O método `HyperlinkEvent` `getURL` obtém o `URL` representado pelo hiperlink.

Seção 27.4 Estabelecendo um servidor simples utilizando sockets de fluxo

- As conexões baseadas em fluxo são gerenciadas com os objetos `Socket`.
- Um objeto `ServerSocket` estabelece a porta onde um servidor espera conexões de clientes. O segundo argumento para o construtor `ServerSocket` é o número de conexões que pode esperar em uma fila para se conectar ao servidor. Se a fila de clientes estiver cheia, as conexões do cliente são recusadas. O método de `ServerSocket` `accept` espera indefinidamente (isto é, bloqueia) uma conexão de um cliente e retorna um objeto `Socket` quando uma conexão é estabelecida.
- Os métodos `getOutputStream` e `getInputStream` de `Socket` obtêm referências a um `OutputStream` e `InputStream` de `Socket`, respectivamente. O método `Socket` `close` termina uma conexão.

Seção 27.5 Estabelecendo um cliente simples utilizando sockets de fluxo

- Um nome de servidor e número de porta são especificados ao criar um objeto `Socket` para permitir que ele conecte um cliente ao servidor. Uma falha na tentativa de conexão lança uma `IOException`.
- O método `InetAddress` `getByName` retorna um objeto `InetAddress` que contém o endereço IP do computador especificado. O método `InetAddress` `getLocalHost` retorna um objeto `InetAddress` que contém o endereço IP do computador local que executa o programa.

Seção 27.7 Interação cliente/servidor sem conexão com datagramas

- A transmissão orientada para conexão é como o sistema de telefonia — você disca e recebe uma conexão para o telefone da pessoa com quem deseja se comunicar. A conexão é mantida até o fim da sua chamada telefônica, mesmo quando você não estiver conversando.
- A transmissão sem conexão com datagramas é semelhante ao envio de correspondências via serviço postal. Uma mensagem grande que não se cabe em um envelope pode ser dividida em partes separadas de mensagem que são colocadas em envelopes separados, numerados sequencialmente. Todas as cartas são então remetidas de uma vez. Elas podem chegar na ordem, fora da ordem ou simplesmente não chegar.
- Os objetos `DatagramPacket` armazenam pacotes de dados que devem ser enviados ou que são recebidos por um aplicativo. DatagramSockets envia e recebe `DatagramPackets`.
- O construtor `DatagramSocket` que não recebe nenhum argumento vincula o `DatagramSocket` a uma porta escolhida pelo computador que está executando o programa. Aquele que recebe um argumento inteiro de número de porta vincula o `DatagramSocket` à porta especificada. Se um construtor `Datagram-Socket` não conseguir vincular o `DatagramSocket` a uma porta, uma `Socket-Exception` ocorrerá. O método `DatagramSocket` `receive` bloqueia (espera) até que um pacote chegue e, então, armazena o pacote no seu argumento.
- O método `DatagramPacket` `getAddress` retorna um objeto `InetAddress` contendo informações sobre o computador do qual (ou para qual) o pacote foi enviado. O método `getPort` retorna um inteiro que especifica o número da porta por meio da qual o `DatagramPacket` host foi enviado ou recebido. O método `getLength` retorna um inteiro que representa o número de bytes dos dados em um `DatagramPacket`. O método `getData` retorna um array de bytes contendo os dados em um `DatagramPacket`.

- O construtor `DatagramPacket` para um pacote a ser enviado aceita quatro argumentos — o array de bytes a ser enviado, o número de bytes a ser enviado, o endereço de cliente para o qual o pacote será enviado e o número da porta onde o cliente está esperando receber os pacotes.
- O método de `DatagramSocket` `send` envia um `DatagramPacket` para fora da rede.
- Se ocorrer um erro ao receber ou enviar um `DatagramPacket`, uma `IOException` ocorrerá.

Terminologia

`127.0.0.1`, (`localhost`), endereço IP, 871
`accept`, método da classe `ServerSocket`, 866
`ACTIVATED`, constante da classe aninhada `EventType`, 865
`AppletContext`, interface, 860
`BindException`, classe, 871
`_blank`, frame-alvo, 863
bloquear, 866
chat cliente-servidor, 868
cliente, 859
`close`, método da classe `Socket`, 867
comprimento da fila, 866
comunicação baseada em pacotes, 859
comunicação baseada em socket, 859
conexão, 859
`DatagramPacket`, classe, 877
`DatagramSocket`, classe, 877
ecoar um pacote de volta para o cliente, 877
endereço de loopback, 871
`ENTERED`, constante da classe aninhada `EventType`, 865
`EventType`, classe aninhada de `HyperlinkEvent`, 865
`EXITED`, constante da classe aninhada `EventType`, 865
`flush`, método da classe `Formatter`, 887
`flush`, método da classe `ObjectOutputStream`, 871
fluxo, cabeçalho, 872
fluxo, comunicações baseadas em, 859
fluxos, 859
fluxo, socket, 859
frame-alvo, 863
`getAddress`, método da classe `DatagramPacket`, 879
`getAppletContext`, método da classe Applet, 863
`getByName`, método da classe `InetAddress`, 876
`getData`, método da classe `DatagramPacket`, 879
`getEventType`, método da classe `HyperlinkEvent`, 865
`getHostName`, método da classe `InetAddress`, 871
`getInetAddress`, método da classe `Socket`, 871
`getInputStream`, método da classe `Socket`, 866
`getLength`, método da classe `DatagramPacket`, 879
`getLocalHost`, método da classe `InetAddress`, 876
`getOutputStream`, método da classe `Socket`, 866
`getParameter`, método da classe `Applet`, 862
`getPort`, método da classe `DatagramPacket`, 879
`getURL`, método da classe `HyperlinkEvent`, 865
handshake, ponto, 866
hyperlink, 865
`HyperlinkEvent`, classe, 863
`HyperlinkListener`, interface, 865
`hyperlinkUpdate`, método da interface `HyperlinkListener`, 865
HyperText Transfer Protocol (HTTP), 860
`InetAddress`, classe, 871
`invokeLater`, método da classe `SwingUtilities`, 871
`java.net`, pacote, 859
`JEditorPane`, classe, 863
ligando o servidor a uma porta, 866
`localhost`, (`127.0.0.1`), endereço, 871
`MalformedURLException`, classe, 863
multicast, 860
número de porta, 866
pacote, 859
pacote de datagrama, 859
`param`, elemento, 862
parâmetro de applet, 862
porta, 866
`receive`, método da classe `DatagramSocket`, 879
registrar uma porta, 866
relacionamento cliente-servidor, 859
`_self`, frame-alvo, 863
`send`, método da classe `DatagramSocket`, 879
`ServerSocket`, classe, 866
serviço orientado para conexão, 859
serviço sem conexão, 859
servidor, 859
`setPage`, método da classe `JEditorPane`, 865
`showDocument`, método da interface `AppletContext`, 860
socket, 859
`Socket`, classe, 866
socket de datagrama, 859
`SocketException`, classe, 877
`SwingUtilities`, classe, 871
TCP (Transmission Control Protocol), 859
`_top`, frame-alvo, 863
transmissão sem conexão, 877
UDP (User Datagram Protocol), 859
Uniform Resource Identifier (URI), 860
`UnknownHostException`, classe, 867
URI (Uniform Resource Identifier), 860
URL (Uniform Resource Locator), 860
User Datagram Protocol (UDP), 859

Exercícios de autorrevisão

27.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) A exceção _____ ocorre quando um erro de entrada/saída ocorre ao fechar um socket.
b) A exceção _____ ocorre quando um nome de host indicado por um cliente não pode ser convertido em um endereço.
c) Se um construtor `DatagramSocket` não conseguir configurar um `DatagramSocket` adequadamente, uma exceção do tipo _____ ocorrerá.
d) Muitas classes para redes do Java estão contidas no pacote _____.
e) A classe _____ vincula o aplicativo a uma porta para transmissão de datagrama.
f) Um objeto da classe _____ contém um endereço IP.
g) Os dois tipos de sockets que discutimos neste capítulo são _____ e _____.
h) O acrônimo URL significa _____.
i) O acrônimo URI significa _____.
j) O protocolo-chave que forma a base da World Wide Web é _____.
k) O método `AppletContext` _____ recebe um objeto `URL` como um argumento e exibe em um navegador o recurso da World Wide Web associado com esse `URL`.

l) O método `getLocalHost` retorna um objeto _____ contendo o endereço IP local do computador em que o programa está executando.

m) O construtor `URL` determina se seu argumento de `String` é um URL válido. Se for, o objeto `URL` é inicializado com essa localização. Se não, uma exceção _____ ocorre.

27.2 Determine se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se *falso*, explique por quê.

a) O UDP é um protocolo orientado para conexão.

b) Com *sockets de fluxo* um processo estabelece uma *conexão* com outro processo.

c) Um servidor espera conexões de um cliente em uma porta.

d) A transmissão de pacote de datagrama em uma rede é confiável — garante-se que os pacotes chegarão na sequência correta.

Respostas dos exercícios de autorrevisão

27.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) sockets de fluxo, sockets de datagrama. h) Uniform Resource Locator. i) Uniform Resource Identifier. j) HTTP. k) `showDocument`. l) `InetAddress`. m) `MalformedURLException`.

27.2 a) Falso; o UDP é um protocolo sem conexão e o TCP é um protocolo orientado a conexões. b) Verdadeiro. c) Verdadeiro. d) Falso; os pacotes podem ser perdidos, chegar fora da ordem ou ser duplicados.

Exercícios

27.3 Faça uma distinção entre serviços de rede sem conexão e orientados para conexão.

27.4 Como um cliente determina o nome de host do computador cliente?

27.5 Sob que circunstâncias uma `SocketException` seria lançada?

27.6 Como um cliente pode obter uma linha de texto de um servidor?

27.7 Descreva como um cliente se conecta a um servidor.

27.8 Descreva como um servidor envia os dados para um cliente.

27.9 Descreva como preparar um servidor para receber uma solicitação de conexão baseada em fluxo a partir de um único cliente.

27.10 Como um servidor ouve conexões baseadas em fluxo de sockets em uma porta?

27.11 O que determina quantas solicitações de conexão de clientes podem esperar em uma fila para se conectar a um servidor?

27.12 Como descrito no texto, que razões podem fazer com que um servidor recuse uma solicitação de conexão de um cliente?

27.13 Utilize uma conexão de socket para permitir a um cliente especificar um nome de arquivo e fazer o servidor enviar o conteúdo do arquivo ou indicar que o arquivo não existe.

27.14 Modifique o Exercício 27.13 para permitir ao cliente modificar o conteúdo do arquivo e enviar o arquivo de volta ao servidor para armazenamento. O usuário pode editar o arquivo em uma `JTextArea`, então clique em um botão *salvar alterações* para enviar o arquivo de volta para o servidor.

27.15 Modifique o programa na Figura 27.2 para permitir aos usuários adicionar e remover seus próprios sites da lista.

27.16 Os servidores multiencadeados são bem populares hoje, especialmente por causa da utilização crescente de servidores multiprocessados. Modifique a aplicação de servidor simples apresentado na Seção 27.6 para ser um servidor com múltiplas threads. Então utilize vários aplicativos clientes e faça cada um deles se conectar ao servidor simultaneamente. Utilize um `ArrayList` para armazenar as threads de cliente. O `ArrayList` fornece vários métodos de uso neste exercício. O método `size` determina o número de elementos em um `ArrayList`. O método `get` retorna o elemento na localização especificada pelo seu argumento. O método `add` coloca seu argumento no fim do `ArrayList`. O método `remove` exclui seu argumento do `ArrayList`.

27.17 (***Jogo de damas***) No texto, apresentamos um programa de jogo-da-velha controlado por um servidor com múltiplas threads. Desenvolva um programa de damas modelado com base no programa Tic-Tac-Toe (jogo da velha). Os dois usuários devem fazer movimentos alternados. Seu programa deve mediar os movimentos dos jogadores, determinando de quem é a vez e permitindo apenas movimentos válidos. Os próprios jogadores determinarão quando o jogo acabou.

27.18 (***Jogo de xadrez***) Desenvolva um programa de jogo de xadrez modelado de acordo com o programa de damas no Exercício 27.17.

27.19 (***Jogo de cartas "vinte e um"***) Desenvolva um programa de jogo de cartas do tipo "vinte e um" em que o aplicativo servidor distribui as cartas para cada um dos clientes. O servidor deve distribuir as cartas adicionais (de acordo com as regras do jogo) para cada jogador quando solicitado.

27.20 (***Jogo de pôquer***) Desenvolva um jogo de cartas de pôquer em que o aplicativo servidor dá as cartas para cada um dos clientes. O servidor deve distribuir as cartas adicionais (de acordo com as regras do jogo) para cada jogador quando solicitado.

27.21 (***Modificações no programa Tic-Tac-Toe com múltiplos threads***) Os programas nas figuras 27.13 e 27.15 implementaram uma versão cliente/servidor com múltiplas threads do jogo-da-velha. Nosso objetivo ao desenvolver esse jogo foi demonstrar um servidor multiencadeado que

pode processar múltiplas conexões de clientes ao mesmo tempo. O servidor no exemplo é na realidade um mediador entre os dois applets clientes — ele se certifica de que cada movimento é válido e que cada cliente move-se na ordem adequada. O servidor não determina quem ganhou ou quem perdeu nem se houve um empate. Além disso, não há capacidade para permitir que um novo jogo seja jogado ou para terminar um jogo existente.

A seguir há uma lista das modificações sugeridas para as Figuras 27.13 e 27.15:

a) Modifique a classe `TicTacToeServer` para testar uma vitória, derrota ou empate depois de cada movimento. Envie uma mensagem para cada cliente indicando o resultado do jogo quando o jogo acabar.

b) Modifique a classe `TicTacToeClient` para exibir um botão que quando clicado permita ao cliente jogar outro jogo. O botão somente deve ser ativado quando um jogo terminar. Observe que tanto a classe `TicTacToeClient` como a classe `TicTacToeServer` devem ser modificadas para redefinir o tabuleiro e todas as informações de estado. Além disso, o outro `TicTacToeClient` deve ser notificado de que um novo jogo está para iniciar de modo que seu tabuleiro e seu estado possam ser reinicializados.

c) Modifique a classe `TicTacToeClient` para fornecer um botão que permita a um cliente terminar o programa a qualquer hora. Quando o usuário clicar no botão, o servidor e o outro cliente devem ser notificados. O servidor então deve esperar uma conexão do outro cliente para que um novo jogo possa começar.

d) Modifique a classe `TicTacToeClient` e a classe `TicTacToeServer` de modo que o vencedor de um jogo possa escolher o ser o X ou o O para o próximo jogo. Lembre-se: X sempre começa primeiro.

e) Se você se sentir ambicioso, permita que um cliente jogue contra o servidor enquanto o servidor espera uma conexão de outro cliente.

27.22 ***(Tic-Tac-Toe 3-D Multiencadeado)*** Modifique o programa cliente/servidor Tic-Tac-Toe multiencadeado para implementar uma versão tridimensional do jogo 4 por 4 por 4. Implemente o aplicativo servidor para mediar entre os dois clientes. Exiba o tabuleiro tridimensional como quatro tabuleiros contendo quatro linhas e quatro colunas cada um. Se você quiser ser ambicioso, experimente as seguintes modificações:

a) Desenhe o tabuleiro de uma maneira tridimensional.

b) Permita ao servidor testar se há um ganhador, um perdedor ou um empate. Cuidado! Há muitas possíveis maneiras de ganhar em um tabuleiro 4 por 4 por 4!

27.23 ***(Código Morse em rede)*** Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para utilização com o sistema de telégrafo. O código Morse atribui uma série de pontos e traços para cada letra do alfabeto, para cada dígito e alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas orientados para áudio, o ponto representa um som curto e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em sinais de bandeira. A separação entre palavras é indicada por um espaço, ou, simplesmente, a ausência de um ponto ou traço. Em um sistema orientado a som, um espaço é indicado por um tempo curto durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Figura 27.18

Caractere	Código	Caractere	Código	Caractere	Código
A	.-	N	-.	*Dígitos*	
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-	0	-----
L	.-..	Y	-.--		
M	--	Z	--..		

Figura 27.18 | Letras e dígitos no código Morse internacional.

Escreva um aplicativo cliente/servidor em que dois clientes possam enviar mensagens em Código Morse entre si por meio de um aplicativo servidor com múltiplas threads. O aplicativo cliente deve permitir que o usuário digite frases em linguagem natural em uma `JTextArea`. Quando o usuário envia a mensagem, o aplicativo cliente codifica o texto em Código Morse e envia a mensagem codificada, por meio do servidor, para o outro cliente. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse. Quando as mensagens são recebidas, elas devem ser decodificadas e exibidas como caracteres normais e como código Morse. O cliente deve ter um `JTextField` para digitar e uma `JText-Area` para exibir mensagens do outro cliente.

28

É um equívoco capital teorizar antes de ter os dados.
— **Arthur Conan Doyle**

Vai pois agora, escreve isso numa tábua perante eles, registra-o num [livro]; para que fique como testemunho para o tempo vindouro, para sempre.
— **A Bíblia Sagrada, Isaías 30:8**

Obtenha seus fatos primeiro e então você pode distorcê-los o quanto quiser.
— **Mark Twain**

Gosto de dois tipos de homens: americanos e estrangeiros.
— **Mae West**

Acesso a bancos de dados com o JDBC

Objetivos

Neste capítulo, você aprenderá:

- Conceitos de banco de dados relacional.
- Utilizar Structured Query Language (SQL) para recuperar dados de um banco de dados e manipular dados em um banco de dados.
- Utilizar o JDBC™ API do pacote `java.sql` para acessar bancos de dados.
- Utilizar a interface `RowSet` do pacote `javax.sql` para manipular bancos de dados.
- Utilizar a descoberta automática de driver JDBC do JDBC 4.0.
- Utilizar `PreparedStatements` para criar instruções de SQL precompiladas com parâmetros.
- Como o processamento de transação torna aplicativos de banco de dados mais robustos.

Sumário

28.1 Introdução
28.2 Bancos de dados relacionais
28.3 Visão geral de banco de dados relacional: O banco de dados `books`
28.4 SQL
- 28.4.1 Consulta `SELECT` básica
- 28.4.2 Cláusula `WHERE`
- 28.4.3 Cláusula `Order BY`
- 28.4.4 Mesclando dados a partir de múltiplas tabelas: `INNER JOIN`
- 28.4.5 Instrução `INSERT`
- 28.4.6 Instrução `UPDATE`
- 28.4.7 Instrução `DELETE`

28.5 Instruções para instalar o MySQL e o MySQL Conector/J
28.6 Instruções para configuração de uma conta de usuário MySQL
28.7 Criando banco de dados `books` no MySQL
28.8 Manipulando bancos de dados com o JDBC
- 28.8.1 Consultando e conectando-se a um banco de dados
- 28.8.2 Consultando o banco de dados `books`

28.9 Interface `RowSet`
28.10 Java DB/Apache Derby
28.11 `PreparedStatements`
28.12 Procedures armazenadas
28.13 Processamento de transações
28.14 Conclusão
28.15 Recursos da Web

Resumo | Terminologia | Exercício de autorrevisão | Respostas do exercício de autorrevisão | Exercícios

28.1 Introdução

Um **banco de dados** é uma coleção organizada de dados. Há muitas estratégias diferentes para organizar dados para facilitar acesso e manipulação. Um **sistema de gerenciamento de bancos de dados** (**Database Management System — DBMS**) fornece mecanismos para armazenar, organizar, recuperar e modificar dados para muitos usuários. Os sistemas de gerenciamento de bancos de dados permitem acesso e armazenamento de dados sem envolver a representação interna de dados.

Os sistemas de banco de dados atuais mais populares são os bancos de dados relacionais, nos quais os dados são armazenados sem se levar em conta sua estrutura física (Seção 28.2). Uma linguagem chamada **SQL** é a linguagem padrão internacional utilizada quase universalmente com bancos de dados relacionais para realizar **consultas** (isto é, solicitar informações que satisfazem determinados critérios) e manipular dados.

Alguns **sistemas de gerenciamento de banco de dados relacional** (**Relational Database Management Systems — RDBMSs**) populares são Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL e MySQL. O JDK vem agora com um RDBMS puro Java chamado Java DB — a versão da Sun para o Apache Derby. Neste capítulo, apresentamos exemplos utilizando MySQL e Java DB.

Os programas Java comunicam-se com bancos de dados e manipulam seus dados utilizando a **Java Database Connectivity** (**JDBC™**) **API**. Um **driver JDBC** permite aos aplicativos Java conectar-se a um banco de dados em um DBMS particular e permite a você manipular esse banco de dados utilizando o JDBC API.

Observação de engenharia de software 28.1

Usar a JDBC API permite que desenvolvedores mudem o DBMS subjacente sem modificar o código Java que acessa o banco de dados.

Os sistemas de gerenciamento de bancos de dados mais populares agora fornecem drivers JDBC. Também há muitos drivers JDBC independentes disponíveis. Neste capítulo, introduzimos o JDBC e o utilizamos para manipular bancos de dados MySQL e Java DB. As técnicas demonstradas aqui também podem ser utilizadas para manipular outros bancos de dados que têm drivers JDBC. Verifique a documentação do DBMS para determinar se o seu DBMS vem com um driver JDBC. Se não, fornecedores independentes comercializam drivers JDBC para muitos DBMSs.

Observação de engenharia de software 28.2

A maioria dos fornecedores de banco de dados importantes fornece seus próprios drivers de banco de dados de JDBC e muitos fornecedores independentes também fornecem drivers JDBC. Para obter informações adicionais visite `devapp.sun.com/product/jdbc/drivers`.

Para obter informações adicionais sobre o JDBC, visite

```
java.sun.com/javase/technologies/database/index.jsp
```

que contém informações sobre o JDBC incluindo a especificação JDBC, FAQs, um centro de recursos de aprendizagem e downloads de software para pesquisa de drivers JDBC para DBMS, e

```
developers.sun.com/product/jdbc/drivers/
```

que fornece um sistema de pesquisa para ajudá-lo a localizar drivers adequados para o seu DBMS.

28.2 Bancos de dados relacionais

Um **banco de dados relacional** é uma representação lógica de dados que permite que os dados sejam acessados sem considerar sua estrutura física. Um banco de dados relacional armazena dados em **tabelas**. A Figura 28.1 ilustra uma tabela de exemplo que pode ser utilizada em um sistema pessoal. O nome da tabela é `Employee` e seu principal propósito é armazenar os atributos de um empregado. As tabelas são compostas de **linhas** e as linhas são compostas de **colunas** nas quais os valores são armazenados. Essa tabela consiste em seis linhas. A coluna `Number` de cada linha é **chave primária** da tabela — uma coluna (ou grupo de colunas) com um valor único que não pode ser duplicado em outras linhas. Isso garante que cada linha possa ser identificada por sua chave primária. Bons exemplos de colunas de chave primária são o número do CPF, o número de ID de um empregado e o número serial de um produto em um sistema de inventário, uma vez que é garantido que os valores em cada uma dessas colunas são únicos. As linhas na Figura 28.1 são exibidas em ordem por chave primária. Nesse caso, as linhas são listadas em ordem crescente, mas também poderíamos utilizar a ordem decrescente.

Number	Name	Department	Salary	Location
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Linha — Chave primária — Coluna

Figura 28.1 | Dados de exemplo da tabela `Employee`.

Não é garantido que as linhas nas tabelas serão armazenadas em alguma ordem particular. Como demonstraremos em um exemplo mais adiante, os programas podem especificar critérios de ordenação ao solicitar dados de um banco de dados.

Cada coluna representa um atributo de dados diferente. Normalmente, as linhas são únicas (pela chave primária) dentro de uma tabela, mas os valores de coluna particulares podem ser duplicados entre as linhas. Por exemplo, três linhas diferentes na coluna `Department` da tabela `Employee` contêm o número 413.

Frequentemente, diferentes usuários de um banco de dados estão interessados em diferentes dados e diferentes relacionamentos entre os dados. A maioria dos usuários exige apenas os subconjuntos das linhas e colunas. Para obter esses subconjuntos, utilize consultas para especificar quais dados selecionar de uma tabela. Você utiliza a SQL para definir consultas complexas. Por exemplo, poderia selecionar dados da tabela `Employee` para criar um resultado que mostra onde cada departamento está localizado, apresentando os dados classificados em ordem crescente por número de departamento. Esse resultado é mostrado na Figura 28.2. As consultas SQL são discutidas na Seção 28.4.

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

Figura 28.2 | Resultado de selecionar dados distintos de `Department` e `Location` a partir da tabela `Employee`.

28.3 Visão geral de um banco de dados relacional: o banco de dados books

Agora fornecemos uma visão geral de bancos de dados relacionais no contexto de um banco de dados `books` de exemplo que criamos para este capítulo. Antes de discutirmos SQL, apresentamos as tabelas do banco de dados `books`. Utilizamos esse banco de dados para introduzir vários conceitos de banco de dados, incluindo como utilizar SQL para obter informações do banco de dados e manipular os dados. Fornecemos um script para criar o banco de dados. Você pode localizar o script no diretório de exemplos deste capítulo. A Seção 28.7 explica como utilizar esse script.

O banco de dados consiste em três tabelas: `Authors`, `AuthorISBN` e `Titles`. A tabela `Authors` (descrita na Figura 28.3) consiste em três colunas que mantêm o número de ID único, nome e sobrenome de cada autor. A Figura 28.4 contém os dados da tabela `Authors` do banco de dados `books`.

Coluna	Descrição
`AuthorID`	Número de ID do autor no banco de dados. No banco de dados `books`, essa coluna inteira é definida como **autoincrementada** — para cada linha inserida nessa tabela, o valor `AuthorID` é aumentado automaticamente por 1 para assegurar que cada linha tenha um único `AuthorID`. Essa coluna representa a chave primária da tabela.
`FirstName`	Nome do autor (uma string).
`LastName`	Sobrenome do autor (uma string).

Figura 28.3 | Tabela `Authors` do banco de dados `books`.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes

Figura 28.4 | Dados de exemplo da tabela `authors`.

A tabela `AuthorISBN` (descrita na Figura 28.5) consiste em duas colunas que mantêm cada ISBN e o número de ID do autor correspondente. Essa tabela associa autores com seus livros. Ambas as colunas são chaves estrangeiras que representam o relacionamento entre as tabelas `Authors` e `Titles` — uma linha na tabela `Authors` poder ser associada com muitas linhas na tabela `Titles` e vice-versa. As colunas combinadas da tabela `AuthorISBN` representam a chave primária da tabela — assim, cada linha nessa tabela deve ser uma combinação única de um `AuthorID` e um `ISBN`. A Figura 28.6 contém os dados da tabela `Author-ISBN` do banco de dados `books`. [*Nota*: Para economizar espaço, dividimos o conteúdo dessa tabela em duas colunas, cada uma contendo as colunas `AuthorID` e `ISBN`]. A coluna chamada `AuthorID` é uma **chave estrangeira** — uma coluna nessa tabela que coincide com a coluna de chave primária em outra tabela (isto é, `AuthorID` na tabela `Authors`). As chaves estrangeiras são especificadas ao criar uma tabela. A chave estrangeira ajuda a manter a **Regra de Integridade Referencial** — cada valor de chave estrangeira deve aparecer como outro valor de chave primária da tabela. Isso permite que o DBMS determine se o valor `AuthorID` de um livro particular é válido. As chaves estrangeiras também permitem que os dados relacionados em múltiplas tabelas sejam selecionados a partir dessas tabelas para fins analíticos — isso é conhecido como fazer uma **join**, ou **junção**, dos dados.

Coluna	Descrição
`AuthorID`	O número de ID do autor, uma chave estrangeira para a tabela `Authors`.
`ISBN`	O ISBN de um livro, uma chave estrangeira para a tabela `Titles`.

Figura 28.5 | Tabela `AuthorISBN` do banco de dados `books`.

AuthorID	ISBN	AuthorID	ISBN
1	0131869000	2	0131450913
2	0131869000	1	0131828274
1	0132222205	2	0131828274
2	0132222205	3	0131450913
1	0131450913	4	0131828274

Figura 28.6 | Dados de exemplo da tabela `AuthorISBN` de `books`.

A tabela `Titles` descrita na Figura 28.7 consiste em quatro colunas que representam o ISBN, o título, o número de edição e o ano dos direitos autorais. A tabela está na Figura 28.8.

Coluna	Descrição
`ISBN`	ISBN do livro (uma string). A chave primária da tabela. O ISBN é a abreviação de "International Standard Book Number" — um sistema de numeração padronizado que os editores utilizam para dar a todos os livros um número de identificação único.
`Title`	Título do livro (uma string).
`EditionNumber`	Número de edição do livro (um inteiro).
`Copyright`	Ano de direitos autorais do livro (uma string).

Figura 28.7 | Tabela `Titles` do banco de dados `books`.

ISBN	Title	EditionNumber	Copyright
0131869000	Visual Basic 2005 How to Program	3	2006
0131525239	Visual C# 2005 How to Program	2	2006
0132222205	Java How to Program	7	2007
0131857576	C++ How to Program	5	2005
0132404168	C How to Program	5	2007
0131450913	Internet & World Wide Web How to Program	3	2004
0131828274	Sistemas operacionais	3	2004

Figura 28.8 | Dados de exemplo da tabela `Titles` do banco de dados `books`.

Há um relacionamento de um para muitos entre uma chave primária e uma chave estrangeira correspondente (por exemplo, um editor pode publicar muitos livros). Uma chave estrangeira pode aparecer muitas vezes na própria tabela, mas pode aparecer apenas uma vez (como a chave primária) em outra tabela. A Figura 28.9 é um **diagrama de relacionamento de entidade** (***Entity-Relationship* — ER**) do banco de dados `books`. Esse diagrama mostra as tabelas de banco de dados e as relações entre elas. O primeiro compartimento em cada quadro contém o nome da tabela e os demais compartimentos contêm as colunas da tabela. Os nomes em itálico são chaves primárias. Uma chave primária da tabela identifica unicamente cada linha na tabela. Cada linha deve ter um valor de chave primária, e esse valor deve ser único na tabela. Isso é conhecido como **Regra de Integridade de Entidade**. Novamente, para a tabela `AuthorISBN`, a chave primária é a combinação de ambas as colunas.

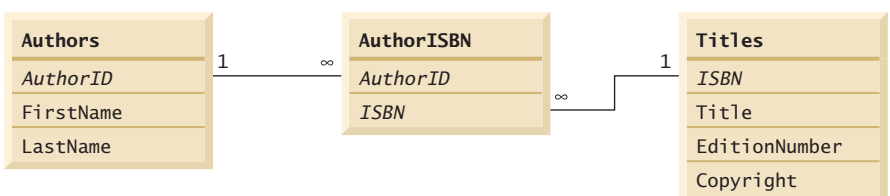


Figura 28.9 | Relações de tabela no banco de dados `books`.

Erro comum de programação 28.1

Não fornecer um valor para cada coluna em uma chave primária quebra a Regra de Integridade de Entidade e faz com que o DBMS informe um erro.

Erro comum de programação 28.2

Fornecer o mesmo valor da chave primária em múltiplas linhas faz com que a DBMS informe um erro.

As linhas que conectam as tabelas (Figura 28.9) representam os relacionamentos entre as tabelas. Considere a linha entre as tabelas `AuthorISBN` e `Authors`. Na extremidade `Authors` da linha há um `1`, e na extremidade `AuthorISBN` há um símbolo de infinito (∞), indicando um **relacionamento de um para muitos** em que cada editor na tabela `Authors` pode ter um número arbitrário de livros na tabela `AuthorISBN`. A linha de relação vincula a coluna `AuthorID` em `Authors` (isto é, sua chave primária) à coluna `AuthorID` em `AuthorISBN` (isto é, a sua chave estrangeira). A coluna `AuthorID` na tabela `AuthorISBN` é uma chave estrangeira.

Erro comum de programação 28.3

Fornecer um valor de chave estrangeira que não aparece como um valor de chave primária em outra tabela quebra a Regra de Integridade Referencial e faz com que o DBMS informe um erro.

A linha entre `Titles` e `AuthorISBN` ilustra outro relacionamento um para muitos; um título pode ser escrito por qualquer número de autores. De fato, o único propósito da tabela `AuthorISBN` é fornecer um relacionamento de muitos para muitos entre `Authors` e `Titles` — um autor pode escrever muitos livros e um livro pode ter muitos autores.

28.4 SQL

Agora fornecemos uma visão geral de SQL no contexto de nosso banco de dados `books`. Você será capaz de utilizar a SQL discutida aqui nos exemplos posteriores do capítulo e nos exemplos dos capítulos 30–31.

As várias subseções a seguir discutem as palavras-chave de SQL listadas na Figura 28.10 no contexto de consultas e instruções SQL. Outras palavras-chave de SQL estão além do escopo deste texto. Para aprender outras palavras-chave, consulte o guia de referência de SQL fornecido pelo fornecedor do RDBMS que você está utilizando. [*Nota*: Para obter mais informações sobre SQL, consulte os recursos Web na Seção 28.15.]

SQL, palavras-chave	Descrição
`SELECT`	Recupera dados de uma ou mais tabelas.
`FROM`	Tabelas envolvidas na consulta. Requeridas em cada `SELECT`.
`WHERE`	Critérios de seleção que determinam as linhas a ser recuperadas, excluídas ou atualizadas. Opcional em uma consulta ou uma instrução de SQL.
`GROUP BY`	Critérios para agrupar linhas. Opcional em uma consulta `SELECT`.
`ORDER BY`	Critérios para ordenar linhas. Opcional em uma consulta `SELECT`.
`INNER JOIN`	Mescla linhas de múltiplas tabelas.
`INSERT`	Insere linhas em uma tabela especificada.
`UPDATE`	Atualiza linhas em uma tabela especificada.
`DELETE`	Exclui linhas de uma tabela especificada.

Figura 28.10 | As palavras-chave de consulta de SQL.

28.4.1 Consulta SELECT básica

Vamos considerar várias consultas de SQL que extraem informações do banco de dados `books`. Uma consulta de SQL "seleciona" linhas e colunas de uma ou mais tabelas em um banco de dados. Essas seleções são realizadas por consultas com a palavra-chave **`SELECT`**. O formato básico de uma consulta `SELECT` é

```
SELECT * FROM nomeDaTabela
```

em que o **asterisco** (`*`) indica que todas as colunas da tabela *nomeDaTabela* devem ser recuperadas. Por exemplo, para recuperar todos os dados na tabela `Authors`, utilize

```
SELECT * FROM Authors
```

A maioria dos programas não exige todos os dados em uma tabela. Para recuperar somente colunas específicas de uma tabela, substitua o asterisco (`*`) por uma lista dos nomes de coluna separados por vírgulas. Por exemplo, para recuperar somente as colunas `AuthorID` e `LastName` de todas as linhas na tabela `authors`, utilize a consulta

```
SELECT AuthorID, LastName FROM Authors
```

Essa consulta retorna os dados listados na Figura 28.11.

AuthorID	LastName
1	Deitel
2	Deitel
3	Goldberg
4	Choffnes

Figura 28.11 | Dados `AuthorID` e `LastName` de exemplo da tabela `Authors`.

Observação de engenharia de software 28.3

Para a maioria das consultas, o asterisco (*) não deve ser utilizado para especificar nomes de coluna. Em geral, os programadores processam resultados sabendo antecipadamente a ordem das colunas no resultado — por exemplo, selecionar `AuthorID` e `LastName` da tabela `Authors` assegura que as colunas aparecerão no resultado com `AuthorID` como a primeira coluna e `LastName` como a segunda coluna. Em geral, os programas processam colunas de resultados especificando o número de coluna no resultado (iniciando do número 1 da primeira coluna). Selecionar colunas por nome também evita retornar colunas desnecessárias e ainda protege contra modificações na ordem real das colunas na(s) tabela(s) retornando colunas na ordem exata especificada.

Erro comum de programação 28.4

Se um programador assume que as colunas são sempre retornadas na mesma ordem de uma consulta que utiliza o asterisco (*), o programa pode processar o resultado incorretamente. Se a ordem de coluna na(s) tabela(s) mudar ou se colunas adicionais forem adicionadas posteriormente, a ordem das colunas no resultado mudaria de maneira correspondente.

28.4.2 Cláusula `WHERE`

Na maioria dos casos, é necessário localizar linhas em um banco de dados que satisfaçam certos **critérios de seleção**. Somente as linhas que satisfazem os critérios de seleção (formalmente chamadas **predicados**) são selecionadas. O SQL utiliza a **cláusula `WHERE`** opcional em uma consulta para especificar os critérios de seleção para a consulta. A forma básica de uma consulta com critérios de seleção é

```
SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela WHERE critérios
```

Por exemplo, para selecionar as colunas `Title`, `EditionNumber` e `Copyright` da tabela `Titles` para a qual os dados de `Copyright` são maiores que `2005`, utilize a consulta

```
SELECT Title, EditionNumber, Copyright
   FROM Titles
   WHERE Copyright > '2005'
```

Observe que as strings na SQL são delimitadas por aspas simples (') em vez de aspas duplas ("").

A Figura 28.12 mostra o resultado da consulta precedente. Os critérios da cláusula `WHERE` podem conter os operadores <, >, <=, >=, =, <> e `LIKE`. O operador **`LIKE`** é utilizado para **correspondência de padrão** com caracteres curingas **porcentagem** (%) e **sublinhado** (_). A correspondência de padrão permite à SQL procurar strings que correspondem a um dado padrão.

Title	EditionNumber	Copyright
Visual C# 2005 How to Program	2	2006
Visual Basic 2005 How to Program	3	2006
Java How to Program	7	2007
C How to Program	5	2007

Figura 28.12 | Amostragem de títulos com direitos autorais posteriores a 2005 da tabela `Titles`.

Um padrão que contém um caractere porcentagem (%) procura strings que tenham zero ou mais caracteres na posição do caractere porcentagem no padrão. Por exemplo, a seguinte consulta localiza as linhas de todos os autores cujo sobrenome começa com a letra `D`:

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE 'D%'
```

A consulta anterior seleciona as duas linhas mostradas na Figura 28.13, porque dois dos quatro autores em nosso banco de dados têm um sobrenome que inicia com a letra `D` (seguida por zero ou mais caracteres). O `%` no padrão `LIKE` da cláusula `WHERE` indica que qualquer número de caracteres pode aparecer depois da letra `D` na coluna `LastName`. Observe que a string do padrão é envolvida entre caracteres de aspas simples.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel

Figura 28.13 | Autores cujo sobrenome começa com `D` da tabela `authors`.

Dica de portabilidade 28.1

Consulte a documentação do sistema de banco de dados para determinar se a SQL diferencia letras maiúsculas de minúsculas no sistema e a sintaxe de palavras-chave de SQL (isto é, todas devem estar em maiúsculas, todas em minúsculas ou algumas combinações das duas?).

Dica de portabilidade 28.2

Leia a documentação do sistema de banco de dados atentamente para determinar se ele suporta o operador `LIKE` conforme discutido aqui. O SQL que discutimos é suportado pela maioria dos RDBMSs, mas é sempre uma boa ideia verificar os recursos SQL que são suportados pelo seu RDBMS.

Um caractere sublinhado (_) na string do padrão indica um único curinga nessa posição no padrão. Por exemplo, a seguinte consulta localiza as linhas de todos os autores cujos sobrenomes começam com qualquer caractere (especificado por _) seguido pela letra `o`, seguida por qualquer número de caracteres adicionais (especificado por `%`):

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE '_o%'
```

A consulta precedente produz a linha mostrada na Figura 28.14 porque apenas um autor em nosso banco de dados tem um sobrenome que contém a letra `o` como sua segunda letra.

AuthorID	FirstName	LastName
3	Andrew	Goldberg

Figura 28.14 | O único autor da tabela `authors` cujo sobrenome contém `o` como a segunda letra.

28.4.3 Cláusula Order BY

As linhas no resultado de uma consulta podem ser classificadas em ordem crescente ou decrescente utilizando a cláusula **`ORDER BY`**. opcional O formato básico de uma consulta com uma cláusula `ORDER BY` é

```
SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela   ORDER BY coluna ASC
SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela   ORDER BY coluna DESC
```

onde `ASC` especifica a ordem crescente (do mais baixo para o mais alto), `DESC` especifica ordem decrescente (do mais alto para o mais baixo) e *coluna* especifica a coluna em que a classificação é baseada. Por exemplo, para obter a lista de autores em ordem crescente por sobrenome (Figura 28.15), utilize a consulta

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName ASC
```

AuthorID	FirstName	LastName
4	David	Choffnes
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg

Figura 28.15 | Dados de exemplo da tabela `Authors` em ordem crescente por `LastName`.

Observe que a ordem de classificação padrão é crescente, então `ASC` é opcional. Para obter a mesma lista de autores em ordem decrescente por sobrenome (Figura 28.16), utilize a consulta

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
3	Andrew	Goldberg
1	Harvey	Deitel
2	Paul	Deitel
4	David	Choffnes

Figura 28.16 | Dados de exemplo da tabela `Authors` em ordem decrescente por `LastName`.

As múltiplas colunas podem ser utilizadas para classificação com uma cláusula `ORDER BY` da forma

```
ORDER BY coluna1 ordemDeClassificação, coluna2 ordemDeClassificação, ...
```

onde *ordemDeClassificação* é tanto `ASC` como `DESC`. Observe que a *ordemDeClassificação* não tem de ser idêntica para cada coluna. A consulta

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName, FirstName
```

classifica todas as linhas em ordem crescente por sobrenome, e depois por nome. Se quaisquer linhas tiverem o mesmo valor de sobrenome, elas serão retornadas classificadas por nome (Figura 28.17).

AuthorID	FirstName	LastName
4	David	Choffnes
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg

Figura 28.17 | Dados de exemplo de `Authors` em ordem crescente por `LastName` e `FirstName`.

As cláusulas `WHERE` e `ORDER BY` podem ser combinadas em uma consulta.

```
SELECT ISBN, Title, EditionNumber, Copyright
   FROM Titles
   WHERE Title LIKE '%How to Program'
   ORDER BY Title ASC
```

que retorna o `ISBN`, o `Title`, o `EditionNumber` e o `Copyright` de cada livro na tabela `Titles` que tem um `Title` terminando com `"How to Program"` e os classifica em ordem crescente por `Title`. Os resultados de consulta são mostrados na Figura 28.18.

ISBN	Title	Edition-Number	Copy-right
0132404168	C How to Program	5	2007
0131857576	C++ How to Program	5	2005
0131450913	Internet and World Wide Web How to Program	3	2004
0132222205	Java How to Program	7	2007
0131869000	Visual Basic 2005 How to Program	3	2006
013152539	Visual C# 2005 How to Program	2	2006

Figura 28.18 | Amostragem de livros da tabela `Titles` cujos títulos acabam com `How to Program` em ordem crescente por `Title`.

28.4.4 Mesclando dados a partir de múltiplas tabelas: INNER JOIN

Os projetistas de banco de dados costumam dividir os dados relacionados em tabelas separadas para assegurar que um banco de dados não armazene dados de maneira redundante. Por exemplo, o banco de dados `books` tem tabelas `Authors` e `Titles`. Utilizamos uma tabela `AuthorISBN` para armazenar os dados de relacionamento entre autores e seus títulos correspondentes. Se não separássemos essas informações em tabelas individuais, precisaríamos incluir as informações de autor com cada entrada na tabela `Titles`. Isso resultaria no armazenamento de informações de autor duplicadas no caso dos autores que escreveram múltiplos livros. Frequentemente é necessário mesclar dados de múltiplas tabelas em um único resultado. Referido como join, ou junção, de tabelas, esse procedimento é especificado por um operador **`INNER JOIN`** na consulta. Uma `INNER JOIN` mescla linhas de duas tabelas correspondendo valores em colunas que são comuns às tabelas. A forma básica de uma `INNER JOIN` é:

```
SELECT nomeDaColuna1, nomeDaColuna2, ...
FROM tabela1
INNER JOIN tabela2
   ON tabela1.nomeDaColuna = tabela2.nomeDaColuna
```

A **cláusula `ON`** da `INNER JOIN` especifica as colunas de cada tabela que são comparadas para determinar que linhas são mescladas. Por exemplo, a consulta a seguir produz uma lista de autores acompanhados pelos ISBNs dos livros escritos por cada autor:

```
SELECT FirstName, LastName, ISBN
FROM Authors
INNER JOIN AuthorISBN
   ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

A consulta mescla dados das colunas `FirstName` e `LastName` de tabela `Authors` com a coluna `ISBN` da tabela `AuthorISBN`, classificando o resultado em ordem crescente por `LastName` e `FirstName`. Observe o uso da sintaxe *nomeDaTabela*`.`*nomeDaColuna* na cláusula `ON`. Essa sintaxe (chamada de **nome qualificado**) especifica as colunas de cada tabela que devem ser comparadas para unir as tabelas. A sintaxe "*NomeDaTabela*`.`" é requerida se as colunas tiverem o mesmo nome em ambas as tabelas. A mesma sintaxe pode ser utilizada em qualquer consulta para distinguir colunas em diferentes tabelas que tenham o mesmo nome. Em alguns sistemas, os nomes de tabela qualificados com o nome de banco de dados podem ser utilizados para realizar consultas entre vários bancos de dados. Como sempre, a consulta pode conter uma cláusula `ORDER BY`. A Figura 28.19 mostra os resultados da consulta anterior, ordenada por `LastName` e `FirstName`. [*Nota*: Para economizar espaço, dividimos o resultado da consulta em duas colunas, cada uma contendo as colunas `FirstName`, `LastName` e `ISBN`].

FirstName	LastName	ISBN	FirstName	LastName	ISBN
David	Choffnes	0131828274	Paul	Deitel	0131869000
Harvey	Deitel	0131869000	Paul	Deitel	0131525239
Harvey	Deitel	0131525239	Paul	Deitel	0132222205
Harvey	Deitel	0132222205	Paul	Deitel	0131857576
Harvey	Deitel	0131857576	Paul	Deitel	0132404168
Harvey	Deitel	0132404168	Paul	Deitel	0131450913
Harvey	Deitel	0131450913	Paul	Deitel	0131828274
Harvey	Deitel	0131828274	Andrew	Goldberg	0131450913

Figura 28.19 | Amostragem de autores e ISBNs dos livros que eles escreveram em ordem crescente por `LastName` e `FirstName`.

Observação de engenharia de software

Se uma instrução de SQL incluir colunas de múltiplas tabelas que tenham o mesmo nome, a instrução deve preceder esses nomes de coluna com seus nomes de tabela e um ponto (por exemplo, `Authors.AuthorID`).

Erro comum de programação 28.5

Em uma consulta, a falha em qualificar nomes de colunas que tenham o mesmo nome em duas ou mais tabelas é um erro.

28.4.5 Instrução INSERT

A instrução **`INSERT`** insere uma linha em uma tabela. A forma básica dessa instrução é

```
INSERT INTO nomeDaTabela ( nomeDaColuna1, nomeDaColuna2, ..., nomeDaColunaN )
   VALUES ( valor1, valor2, ..., valorN )
```

onde *nomeDaTabela* é a tabela na qual inserir a linha. O *nomeDaTabela* é seguido por uma lista de nomes de coluna separados por vírgulas entre parênteses (essa lista não é necessária se a operação `INSERT` especificar um valor para cada coluna da tabela na ordem correta). A lista de nomes de coluna é seguida pela palavra-chave de SQL **`VALUES`** e uma lista separada por vírgulas de valores entre parênteses. Os valores especificados aqui devem corresponder às colunas especificadas depois do nome de tabela tanto pela ordem como pelo tipo (por exemplo, se *nomeDaColuna1* deve ser a coluna `FirstName`, então o *valor1* deve ser uma string entre aspas simples que representa o nome). Sempre liste explicitamente as colunas ao inserir linhas. Se a ordem de coluna da tabela mudar ou uma nova coluna for adicionada, usar somente `VALUES` pode causar um erro. A instrução `INSERT`

```
INSERT INTO Authors ( FirstName, LastName )
   VALUES ( 'Sue', 'Smith' )
```

insere uma linha na tabela `Authors`. A instrução indica que os valores são fornecidos para as colunas `FirstName` e `LastName`. Os valores correspondentes são `'Sue'` e `'Smith'`. Não especificamos um `AuthorID` nesse exemplo porque `AuthorID` é uma coluna autoincementada na tabela `Authors`. Para cada linha adicionada a essa tabela, o DBMS atribui um único valor `AuthorID` que é o próximo valor na sequência autoincementada (isto é, 1, 2, 3 e assim por diante). Nesse caso, Sue Smith receberia o `AuthorID` número 5. A Figura 28.20 mostra a tabela `Authors` depois da operação `INSERT`. [*Nota*: Nem todo sistema de gerenciamento de bancos de dados suporta colunas autoincementadas. Verifique a documentação do seu DBMS para alternativas às colunas de autoincemento].

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Sue	Smith

Figura 28.20 | Os dados de exemplo da tabela `Authors` depois de uma operação `INSERT`.

Erro comum de programação 28.6

É um erro especificar um valor para uma coluna autoincementada.

Erro comum de programação 28.7

O SQL delimita strings com aspas simples (`'`). Uma string que contém uma aspa simples (por exemplo, O'Malley) deve ter duas aspas simples na posição em que a aspa simples aparece (por exemplo, `'O''Malley'`). A primeira atua como um caractere de escape da segunda. Não "escapar" caracteres aspas simples em uma string que seja parte de uma instrução de SQL é um erro de sintaxe de SQL.

28.4.6 Instrução UPDATE

Uma instrução **`UPDATE`** modifica os dados em uma tabela. Sua forma básica é

```
UPDATE nomeDaTabela
   SET nomeDaColuna1= valor1, nomeDaColuna2= valor2, ..., nomeDaColunaN= valorN
   WHERE critérios
```

onde *nomeDaTabela* é a tabela a atualizar. O *nomeDaTabela* é seguido por uma palavra-chave **SET** e uma lista separada por vírgulas de pares nome-valor de coluna na forma *nomeDaColuna* = *valor*. A cláusula `WHERE` opcional fornece critérios que determinam quais linhas atualizar. Apesar de não ser necessária, a cláusula `WHERE` é geralmente utilizada, a menos que uma alteração seja feita em cada linha. A instrução `UPDATE`

```
UPDATE Authors
   SET LastName = 'Jones'
   WHERE LastName = 'Smith' AND FirstName = 'Sue'
```

atualiza uma linha na tabela `Authors`. A instrução indica que `LastName` receberá o valor `Jones` para a linha em que `LastName` é igual a `Smith` e `FirstName` é igual a `Sue`. [*Nota*: Se houver múltiplas linhas com o nome "Sue" e o sobrenome "Smith," essa instrução modificará todas essas linhas para ter o sobrenome "Jones"]. Se soubéssemos o `AuthorID` antes da operação `UPDATE` (possivelmente porque pesquisamos por ela anteriormente), a cláusula `WHERE` poderia ser simplificada da seguinte maneira:

```
WHERE AuthorID = 5
```

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes
5	Sue	Jones

Figura 28.21 | Os dados de exemplo da tabela `Authors` depois de uma operação `UPDATE`.

28.4.7 Instrução DELET

Uma instrução **DELETE** de SQL remove as linhas de uma tabela. Sua forma básica é

```
DELETE FROM nomeDaTabela   WHERE critérios
```

onde *nomeDaTabela* é a tabela a partir da qual excluir. A cláusula `WHERE` opcional especifica os critérios utilizados para determinar quais linhas excluir. Se essa cláusula for omitida, todas as linhas da tabela serão excluídas. A instrução `DELETE`

```
DELETE FROM Authors
   WHERE LastName = 'Jones' AND FirstName = 'Sue'
```

exclui a linha para Sue Jones na tabela `Authors`. Se soubermos o `AuthorID` antes de a operação `DELETE` ocorrer, a cláusula `WHERE` pode ser simplificada assim:

```
WHERE AuthorID = 5
```

A Figura 28.22 mostra a tabela `Authors` depois de a operação `DELETE` ter acontecido.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Andrew	Goldberg
4	David	Choffnes

Figura 28.22 | Os dados de exemplo da tabela `Authors` depois de uma operação `INSERT`.

28.5 Instruções para instalar o MySQL e o MySQL Conector/J

MySQL 5.0 Community Edition é um sistema de gerenciamento de bancos de dados de código-fonte aberto que executa em muitas plataformas, incluindo Windows, Solaris, Linux e Macintosh. Informações completas sobre o MySQL estão disponíveis em `www.mysql.com`. Os exemplos das seções 28.8–28.9 manipulam bancos de dados MySQL.

Instale o MySQL

Para instalar o MySQL Community Edition:

1. Para aprender os requisitos de instalação da sua plataforma, visite o site `dev.mysql.com/doc/refman/5.0/en/installing-cs.html`.
2. Visite `dev.mysql.com/downloads/mysql/5.0.html` e baixe o instalador da sua plataforma. Para os exemplos de MySQL neste capítulo, você só precisa do pacote Windows Essentials no Microsoft Windows, ou do pacote Standard na maioria das outras plataformas. [*Nota*: Para essas instruções, assumimos que você esteja executando o Microsoft Windows. As instruções de instalação completas de outras plataformas estão disponíveis em `dev.mysql.com/doc/refman/5.0/en/installing.html`.]
3. Dê um clique duplo em `mysql-essential-5.0.67-win32.msi` para iniciar o instalador. [*Nota*: Esse nome pode diferir de acordo com a versão atual do MySQL 5.0.] Clique em Next>.
4. Escolha Typical para o Setup Type e clique em Next >. Então clique em Install.

Quando a instalação estiver completa, clique em Next> duas vezes e, depois em Finish para iniciar a configuração do servidor. Para configurar o servidor:

1. Clique em Next >, então selecione Standard Configuration e clique em Next > novamente.
2. Você tem a opção de instalar MySQL como um serviço Windows, o que permite que o servidor MySQL comece a executar automaticamente toda vez que o sistema iniciar. Para nossos exemplos, isso é desnecessário, portanto, desmarque Install as a Windows Service e marque Include Bin Directory in Windows PATH. Isso permitirá utilizar os comandos MySQL no Windows Command Prompt.
3. Clique em Next > e, então, clique em Execute para realizar a configuração de servidor.
4. Clique em Finish para fechar o assistente.

Instale o MySQL Connector/J

Para usar o MySQL com o JDBC, você também precisa instalar o **MySQL Connector/J** (o J significa Java) — um driver JDBC que permite que programas usem o JDBC para interagir com MySQL. O MySQL Connector/J pode ser baixado de

```
dev.mysql.com/downloads/connector/j/5.1.html
```

A documentação do Conector/J encontra-se em `dev.mysql.com/doc/connector/j/en/connector-j.html`. Quando escrevíamos este livro, a versão estável atual do MySQL Connector/J era a versão 5.1.7. Para instalar o MySQL Connector/J:

1. O arquivo para download é `mysql-connector-java-5.1.7.tar.gz`.
2. Abra `mysql-connector-java-5.1.7.tar.gz` com um extrator de arquivos, como o WinZip (`www.winzip.com`). Extraia seu conteúdo para a unidade `C:\`. Isso criará um diretório chamado `mysql-connector-java-5.1.7`. O subdiretório `docs` dessa pasta contém a documentação do MySQL Connector/J (`connector-j.pdf`). Você também pode visualizá-lo on-line em `dev.mysql.com/doc/connector/j/en/connector-j.html`.

28.6 Instruções para configuração de uma conta de usuário MySQL

Para que os exemplos no livro executem corretamente, você precisa configurar uma conta de usuário que permita aos usuários criar, excluir e modificar um banco de dados. Depois que o MySQL estiver instalado, siga os passos abaixo para configurar uma conta de usuário (esses passos assumem que MySQL está instalado no diretório de instalação padrão):

1. Abra um prompt de comando e inicie o servidor do bancos de dados executando o comando `mysqld-nt.exe`. Observe que esse comando não tem saída — ele simplesmente inicia o servidor MySQL. Não feche essa janela — isso encerra o servidor.
2. Em seguida, você iniciará o monitor MySQL para poder configurar uma conta de usuário, abrir outro prompt de comando e executar o comando

```
mysql -h localhost -u root
```

A opção `-h` indica o host (isto é, computador) no qual o servidor MySQL está executando — nesse caso o seu computador local (`localhost`). A opção `-u` indica a conta de usuário que será utilizada para fazer login no servidor — `root` é a conta de usuário padrão criada durante a instalação para permitir que você configure o servidor. Assim que você fizer o login, verá um prompt `mysql>` no qual pode digitar comandos para interagir com o servidor MySQL.

3. No prompt `mysql>`, digite

```
USE mysql;
```

para selecionar o banco de dados predefinido chamado `mysql`, que armazena informações sobre o servidor, como contas de usuário e seus privilégios para interagir com o servidor. Observe que cada comando deve terminar com um ponto-e-vírgula. Para confirmar o comando, MySQL emite a mensagem "`Database changed`."

4. Em seguida, você adicionará a conta de usuário `deitel` ao banco de dados interno `mysql`. O banco de dados `mysql` contém uma tabela chamada `user` com colunas que representam o nome do usuário, a senha e vários privilégios. Para criar a conta de usuário `deitel` com a senha `deitel`, execute os seguintes comandos do prompt `mysql>`:

```
create user 'deitel'@'localhost' identified by 'deitel';
grant select, insert, update, delete, create, drop, references,
   execute on *.* to 'deitel'@'localhost';
```

Isso cria o usuário `deitel` com os privilégios necessários para criar os bancos de dados utilizados neste capítulo e manipulá-los.

5. Digite o comando

```
exit;
```

para encerrar o monitor MySQL.

28.7 Criando banco de dados books no MySQL

Para cada banco de dados MySQL que discutimos, fornecemos um script SQL em um arquivo `.sql` que configura o banco de dados e suas tabelas. Você pode executar esses scripts no monitor MySQL. No diretório de exemplos deste capítulo, você localizará o script `books.sql` para criar o banco de dados `books`. Para os próximos passos, supomos que o servidor MySQL (`mysqld-nt.exe`) ainda esteja rodando. Para executar o script `books.sql`:

1. Abra um prompt de comando e utilize o comando `cd` para mudar os diretórios para o local que contenha o script `books.sql`.
2. Inicie o monitor MySQL digitando

```
mysql -h localhost -u deitel -p
```

A opção `-p` solicita a senha para a conta de usuário `deitel`. Quando solicitado, insira a senha `deitel`.

3. Execute o script digitando

```
source books.sql;
```

Isso cria um novo diretório chamado `books` no diretório `data` do servidor — localizado por padrão no Windows em `C:\Program Files\MySQL\MySQL Server 5.0\data`. Esse novo diretório contém o banco de dados `books`.

4. Digite o comando

```
exit;
```

para terminar o monitor MySQL. Agora você está pronto para avançar para o primeiro exemplo de JDBC.

28.8 Manipulando bancos de dados com o JDBC

Nesta seção, apresentamos dois exemplos. O primeiro exemplo introduz como se conectar a um banco de dados e consultá-lo. O segundo exemplo demonstra como exibir o resultado da consulta em uma `JTable`.

28.8.1 Consultando e conectando-se a um banco de dados

O exemplo da Figura 28.23 realiza uma consulta simples no banco de dados `books` que recupera a tabela `Authors` inteira e exibe os dados. O programa ilustra a conexão com o banco de dados, consultando o banco de dados e processando o resultado. A seguinte discussão apresenta os aspectos-chave de JDBC do programa. [*Nota*: As seções 28.5–28.7 demonstram como iniciar o servidor MySQL, configurar uma conta de usuário e criar o banco de dados `books`. Esses passos *devem* ser realizados antes de executar o programa da Figura 28.23.]

```
1  // Figura 28.23: DisplayAuthors.java
2  // Exibindo o conteúdo da tabela authors.
3  import java.sql.Connection;
```

```
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
8  import java.sql.SQLException;
9
10 public class DisplayAuthors
11 {
12    // URL do banco de dados
13    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
14
15    // carrega o aplicativo
16    public static void main( String args[] )
17    {
18       Connection connection = null; // gerencia a conexão
19       Statement statement = null; // instrução de consulta
20       ResultSet resultSet = null; // gerencia resultados
21
22       // conecta-se ao banco de dados books e o consulta
23       try
24       {
25          // estabelece uma conexão com o banco de dados
26          connection = DriverManager.getConnection(
27             DATABASE_URL, "deitel", "deitel" );
28
29          // cria Statement para consultar banco de dados
30          statement = connection.createStatement();
31
32          // consulta o banco de dados
33          resultSet = statement.executeQuery(
34             "SELECT AuthorID, FirstName, LastName FROM Authors" );
35
36          // processa os resultados da consulta
37          ResultSetMetaData metaData = resultSet.getMetaData();
38          int numberOfColumns = metaData.getColumnCount();
39          System.out.println( "Authors Table of Books Database:\n" );
40
41          for ( int i = 1; i <= numberOfColumns; i++ )
42             System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
43          System.out.println();
44
45          while (resultSet.next() )
46          {
47             for ( int i = 1; i <= numberOfColumns; i++ )
48                System.out.printf( "%-8s\t", resultSet.getObject( i ) );
49             System.out.println();
50          } // fim do while
51       }  // fim do try
52       catch ( SQLException sqlException )
53       {
54          sqlException.printStackTrace();
55       } // fim do catch
56       finally // assegura que o resultSet, a instrução e a conexão estão fechados
57       {
58          try
59          {
60             resultSet.close();
61             statement.close();
62             connection.close();
63          } // fim do try
64          catch ( Exception exception )
65          {
66             exception.printStackTrace();
67          } // fim do catch
68       } // fim de finally
69    } // fim de main
70 } // fim da classe DisplayAuthors
```

```
Authors Table of Books Database:
AuthorID        FirstName       LastName
1               Harvey          Deitel
2               Paul            Deitel
3               Andrew          Goldberg
4               David           Choffnes
```

Figura 28.23 | Exibindo o conteúdo da tabela authors.

As linhas 3–8 importam as interfaces JDBC e as classes do pacote `java.sql` utilizadas nesse programa. A linha 13 declara uma constante de string para o URL de banco de dados. Isso identifica o nome do banco de dados ao qual se conectar, bem como informações sobre o protocolo utilizado pelo driver JDBC (discutido em breve). O método `main` (linhas 16–69) conecta-se ao banco de dados de livros, o consulta, exibe o resultado dessa consulta e fecha a conexão.

Nas versões anteriores do Java, os programas precisavam carregar um driver de banco de dados apropriado antes de se conectar a um banco de dados. O JDBC 4.0, parte do Java SE 6, suporta a **descoberta de driver automática** — você não precisa mais carregar o driver de banco de dados com antecedência. Para assegurar que o programa pode localizar a classe de driver do banco de dados, você deve incluir a localização da classe no classpath do programa ao executá-lo. Para MySQL, você inclui o arquivo `mysql-connector-java-5.1.7-bin.jar` (no diretório `C:\mysql-connector-java-5.1.7`) no classpath do seu programa, como em:

```
java -classpath
   .;c:\mysql-connector-java-5.1.7\mysql-connector-java-5.1.7-bin.jar
   DisplayAuthors
```

Observe o ponto (`.`) no início das informações sobre o classpath. Se ele estiver ausente, a JVM não procurará classes no diretório atual e, assim, não localizará o arquivo da classe `DisplayAuthors`. Você também pode copiar o arquivo `mysql-connector-java-5.1.7-bin.jar` para a pasta `\jre\lib\ext` do seu JDK. Depois de fazer isso, você poderia executar o aplicativo simplesmente utilizando o comando `java DisplayAuthors`.

```
java DisplayAuthors
```

As linhas 26–27 da Figura 28.23 criam um objeto **`Connection`** (pacote `java.sql`) referenciado por `connection`. Um objeto que implementa a interface `Connection` gerencia a conexão entre o programa Java e o banco de dados. Os objetos `Connection` permitem aos programas criar instruções de SQL que acessem bancos de dados. O programa inicializa `connection` com o resultado de uma chamada para o método `static` **`getConnection`** da classe **`DriverManager`** (pacote `java.sql`), que tenta conectar-se ao banco de dados especificado por seu URL. O método `getConnection` aceita três argumentos — uma `String` que especifica o URL de banco de dados, uma `String` que especifica o nome de usuário e uma `String` que especifica a senha. O nome de usuário e a senha são configurados na Seção 28.6. Se tiver utilizado o nome de usuário e senha diferentes, você precisa substituir o nome de usuário (segundo argumento) e a senha (terceiro argumento) passados para o método `getConnection` na linha 27. O URL localiza o banco de dados (possivelmente em uma rede ou no sistema de arquivos local do computador). O URL `jdbc:mysql://localhost/books` especifica o protocolo de comunicação (`jdbc`), o **subprotocolo** de comunicação (`mysql`) e a localização do banco de dados (`//localhost/books`, em que `localhost` é o nome do host de servidor de MySQL e `books` é o nome de banco de dados). O subprotocolo `mysql` indica que o programa utiliza um subprotocolo específico ao MySQL para conectar-se ao banco de dados MySQL. Se `DriverManager` não se conectar ao banco de dados, o método `getConnection` lança uma **`SQLException`** (pacote `java.sql`). A Figura 28.24 lista os nomes de driver JDBC e formatos de URL de banco de dados de vários RDBMSs populares.

RDBMS	Formato de URL de banco de dados
MySQL	`jdbc:mysql://`*nomeDoHost*`:`*númeroDaPorta*`/`*nomeDoBancoDeDados*
ORACLE	`jdbc:oracle:thin:@`*nomeDoHost*`:`*númeroDaPorta*`:`*nomeDoBancoDeDados*
DB2	`jdbc:db2:`*nomeDoHost*`:`*númeroDaPorta*`/`*nomeDoBancoDeDados*
Java DB/Apache Derby	`jdbc:derby:`*nomeBancoDeDados* (incorporado) `jdbc:derby://`*nomeDoHost*`:` *númeroDaPorta* `/`*nomeDoBancoDeDados* (rede)
Microsoft SQL Server	`jdbc:sqlserver://` *nomeDoHost*`:` *númeroDaPorta*`;databaseName=` *nomeDoBancoDeDados*
Sybase	`jdbc:sybase:Tds:`*nomeDoHost*`:`*númeroDaPorta*`/`*nomeDoBancoDeDados*

Figura 28.24 | Formatos de URL do banco de dados JDBC populares.

Observação de engenharia de software 28.5

A maioria dos sistemas de gerenciamento de bancos de dados requer que o usuário efetue logon antes de acessar o conteúdo de banco de dados. O método `DriverManager` `getConnection` é sobrecarregado com versões que permitem ao programa fornecer o nome de usuário e a senha para ganhar acesso.

A linha 30 invoca o método `Connection` **`createStatement`** para obter um objeto que implementa interface `Statement` (pacote `java.sql`). O programa utiliza o objeto **`Statement`** para enviar instruções de SQL ao banco de dados.

As linhas 33–34 utilizam o método **`executeQuery`** do objeto `Statement` para submeter uma consulta que seleciona todas as informações de autor de tabela `Authors`. Esse método retorna um objeto que implementa a interface **`ResultSet`** e contém os resultados da consulta. Os métodos `ResultSet` permitem que o programa manipule o resultado da consulta.

As linhas 37–50 processam o `ResultSet`. A linha 37 obtém os **metadados** do `ResultSet` como um objeto **`ResultSetMetaData`** (pacote `java.sql`). Os metadados descrevem o conteúdo do `ResultSet`. Os programas podem utilizar os metadados programaticamente a fim de obter informações sobre nomes de coluna e tipos do `ResultSet`. A linha 38 usa o método `ResultSetMetaData` **`getColumnCount`** para recuperar o número de colunas no `ResultSet`. As linhas 41–42 exibem os nomes de coluna.

Observação de engenharia de software 28.6

Os metadados permitem aos programas processar o conteúdo de `ResultSet` dinamicamente quando informações detalhadas sobre o `ResultSet` não forem conhecidas antecipadamente.

As linhas 45–50 exibem os dados em cada linha do `ResultSet`. Em primeiro lugar, o programa posiciona o cursor `ResultSet` (que aponta para a linha que é processada) à primeira linha no `ResultSet` com o método **`next`** (linha 45). O método `next` retorna o valor `boolean` `true` se for capaz de se posicionar na próxima linha; caso contrário, o método retorna `false`.

Erro comum de programação 28.8

Inicialmente, um cursor `ResultSet` é posicionado antes da primeira linha. Uma `SQLException` ocorre se você tentar acessar o conteúdo de um `ResultSet` antes de posicionar o cursor `ResultSet` na primeira linha com o método `next`.

Se houver linhas no `ResultSet`, as linhas 47–48 extraem e exibem o conteúdo de cada coluna na linha atual. Quando um `ResultSet` é processado, cada coluna pode ser extraída como um tipo Java específico. De fato, o método `ResultSetMetaData` **`getColumnType`** retorna um inteiro constante da classe **`Types`** (pacote `java.sql`) que indica o tipo de uma coluna especificada. Os programas podem utilizar esses valores em uma instrução `switch` para invocar os métodos `ResultSet` que retornam os valores de coluna como tipos Java apropriados. Se o tipo de uma coluna for `Types.INTEGER`, o método `ResultSet` **`getInt`** retorna o valor de coluna como um `int`. Os métodos `ResultSet` *get* em geral recebem como um argumento um *número de coluna* (como um `int`) ou um *nome de coluna* (como uma `String`) indicando que valor da coluna obter. Visite

```
java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/GettingStartedTOC.fm.html
```

para obter mapeamentos detalhados de tipos de dados de SQL para tipos Java e determinar o método `ResultSet` apropriado para chamar cada tipo de dados de SQL.

Dica de desempenho 28.1

Se uma consulta especificar as colunas exatas a selecionar do banco de dados, o `ResultSet` conterá as colunas na ordem especificada. Nesse caso, utilizar o número de coluna para obter o valor da coluna é mais eficiente que utilizar o nome de coluna. O número de coluna fornece acesso direto à coluna especificada. Utilizar o nome de coluna requer uma pesquisa linear dos nomes de coluna para localizar a coluna apropriada.

Dica de prevenção de erro 28.1

Utilizar nomes de coluna para obter valores de um `ResultSet` produz um código que é menos propenso a erros do que obter valores por número de coluna — você não precisa se lembrar da ordem de coluna. Além disso, se a ordem de coluna mudar, o código não precisa mudar.

Para simplificar, esse exemplo trata cada valor como um `Object`. O programa recupera todo valor de coluna com o método `ResultSet` **`getObject`** (linha 48) e imprime a representação `String` do `Object`. Observe que, diferentemente de índices de array, que iniciam em 0, números de coluna `ResultSet` iniciam em 1. O bloco `finally` (linhas 56–68) fecha `ResultSet` (linha 60) e o banco de dados `Statement` (linha 61). [*Nota*: As linhas 60–62 lançarão `NullPointerExceptions` se os objetos `ResultSet`, `Statement` ou `Connection` não forem

criados corretamente. Para o código utilizado na indústria, você deve verificar as variáveis que se referem a esses objetos para ver se eles serão `null` antes de você chamar `close`.]

Erro comum de programação 28.9

Especificar número de coluna 0 ao obter valores de um `ResultSet` produz uma `SQLException`.

Erro comum de programação 28.10

Uma `SQLException` ocorre se você tentar manipular um `ResultSet` depois de fechar o `Statement` que a criou. O `ResultSet` é descartado quando a `Statement` correspondente for fechada.

Observação de engenharia de software

Todo objeto `Statement` pode abrir apenas um objeto `ResultSet` por vez. Quando um `Statement` retorna um novo `ResultSet`, `Statement` fecha o `ResultSet` anterior. Para utilizar múltiplos `ResultSets` paralelamente, objetos `Statement` separados devem retornar `ResultSets`.

28.8.2 Consultando o banco de dados books

O próximo exemplo (figuras 28.25 e 28.28) permite ao usuário inserir qualquer consulta no programa. O exemplo exibe o resultado de uma consulta em uma **`JTable`**, usando um objeto **`TableModel`** para fornecer os dados de `ResultSet` para a `JTable`. Um `JTable` é um componente GUI Swing que pode ser vinculado a um banco de dados para exibir os resultados de uma consulta. A classe `ResultSetTableModel` (Figura 28.25) realiza a conexão com o banco de dados via uma `TableModel` e mantém o `ResultSet`. A classe `DisplayQueryResults` (Figura 28.28) cria a GUI e especifica uma instância da classe `ResultSetTableModel` para fornecer dados a `JTable`.

Classe `ResultSetTableModel`

A classe `ResultSetTableModel` (Figura 28.25) estende a classe **`AbstractTableModel`** (pacote `javax.swing.table`), que implementa a interface `TableModel`. `ResultSetTableModel` sobrescreve os métodos `TableModel` **`getColumnClass`**, **`getColumnCount`**, **`getColumnName`**, **`getRowCount`** e **`getValueAt`**. As implementações padrão dos métodos `TableModel` `isCellEditable` e `setValueAt` (fornecidas por `AbstractTableModel`) não são sobrescritas, porque esse exemplo não suporta a edição de células `JTable`. As implementações padrão dos métodos `TableModel` **`addTableModelListener`** e **`removeTableModelListener`** (fornecidos por `AbstractTableModel`) não são sobrescritas, porque as implementações desses métodos em `AbstractTableModel` adicionam e removem adequadamente os ouvintes (*listeners*) de evento.

```
1  // Figura 28.25: ResultSetTableModel.java
2  // Um TableModel que fornece dados ResultSet a uma JTable.
3  import java.sql.Connection;
4  import java.sql.Statement;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.ResultSetMetaData;
8  import java.sql.SQLException;
9  import javax.swing.table.AbstractTableModel;
10
11 // Linhas e colunas do ResultSet são contadas a partir de 1 e linhas e
12 // colunas JTable são contadas a partir de 0. Ao processar
13 // linhas ou colunas de ResultSet para utilização em uma JTable, é
14 // necessário adicionar 1 ao número de linha ou coluna para manipular
15 // a coluna apropriada de ResultSet (isto é, coluna 0 de JTable é a
16 // coluna de ResultSet 1 e a linha de JTable 0 é a linha de ResultSet 1).
17 public class ResultSetTableModel extends AbstractTableModel
18 {
19    private Connection connection;
20    private Statement statement;
21    private ResultSet resultSet;
22    private ResultSetMetaData metaData;
23    private int numberOfRows;
24
25    // monitora o status da conexão de banco de dados
26    private boolean connectedToDatabase = false;
27
```

```
28       // construtor inicializa resultSet e obtém seu objeto de metadados;
29       // determina o número de linhas
30       public ResultSetTableModel( String url, String username,
31           String password, String query ) throws SQLException
32       {
33          // conecta-se ao banco de dados
34          connection = DriverManager.getConnection( url, username, password );
35
36          // cria Statement para consultar o banco de dados
37          statement = connection.createStatement(
38             ResultSet.TYPE_SCROLL_INSENSITIVE,
39             ResultSet.CONCUR_READ_ONLY );
40
41          // atualiza status de conexão de banco de dados
42          connectedToDatabase = true;
43
44          // configura consulta e a executa
45          setQuery( query );
46       } // fim do construtor ResultSetTableModel
47
48       // obtém a classe que representa o tipo de coluna
49       public Class getColumnClass( int column ) throws IllegalStateException
50       {
51          // assegura que o banco de dados conexão está disponível
52          if ( !connectedToDatabase )
53             throw new IllegalStateException( "Not Connected to Database" );
54
55          // determina a classe Java de coluna
56          try
57          {
58             String className = metaData.getColumnClassName( column + 1 );
59
60             // retorna objeto Class que representa className
61             return Class.forName( className );
62          } // fim do try
63          catch ( Exception exception )
64          {
65             exception.printStackTrace();
66          } // fim do catch
67
68          return Object.class; // se ocorrerem os problemas acima, assume tipo Object
69       } // fim do método getColumnClass
70
71       // obtém o número de colunas em ResultSet
72       public int getColumnCount() throws IllegalStateException
73       {
74          // assegura que o banco de dados conexão está disponível
75          if ( !connectedToDatabase )
76             throw new IllegalStateException( "Not Connected to Database" );
77
78          // determina número de colunas
79          try
80          {
81             return metaData.getColumnCount();
82          } // fim do try
83          catch ( SQLException sqlException )
84          {
85             sqlException.printStackTrace();
86          } // fim do catch
87
88          return 0; // se ocorrerem os problemas acima, retorna 0 para o número de colunas
89       } // fim do método getColumnCount
90
91       // obtém o nome de uma coluna particular em ResultSet
92       public String getColumnName( int column ) throws IllegalStateException
93       {
94          // assegura que o banco de dados conexão está disponível
95          if ( !connectedToDatabase )
96             throw new IllegalStateException( "Not Connected to Database" );
```

```
97
98          // determina o nome de coluna
99          try
100         {
101            return metaData.getColumnName( column + 1 );
102         } // fim do try
103         catch ( SQLException sqlException )
104         {
105            sqlException.printStackTrace();
106         } // fim do catch
107
108         return ""; // se ocorrerem problemas, retorna string vazia para nome de coluna
109      } // fim do método getColumnName
110
111      // retorna o número de linhas em ResultSet
112      public int getRowCount() throws IllegalStateException
113      {
114         // assegura que o banco de dados conexão está disponível
115         if ( !connectedToDatabase )
116            throw new IllegalStateException( "Not Connected to Database" );
117
118         return numberOfRows;
119      } // fim do método getRowCount
120
121      // obtém o valor na linha e na coluna especificadas
122      public Object getValueAt( int row, int column )
123         throws IllegalStateException
124      {
125         // assegura que o banco de dados conexão está disponível
126         if ( !connectedToDatabase )
127            throw new IllegalStateException( "Not Connected to Database" );
128
129         // obtém o valor na linha e na coluna especificadas do ResultSet
130         try
131         {
132            resultSet.absolute( row + 1 );
133            return resultSet.getObject( column + 1 );
134         } // fim do try
135         catch ( SQLException sqlException )
136         {
137            sqlException.printStackTrace();
138         } // fim do catch
139
140         return ""; // se ocorrerem problemas, retorna objeto string vazio
141      } // fim do método getValueAt
142
143      // configura a nova string de consulta de banco de dados
144      public void setQuery( String query )
145         throws SQLException, IllegalStateException
146      {
147         // assegura que o banco de dados conexão está disponível
148         if ( !connectedToDatabase )
149            throw new IllegalStateException( "Not Connected to Database" );
150
151         // especifica a consulta e a executa
152         resultSet = statement.executeQuery( query );
153
154         // obtém os metadados para o ResultSet
155         metaData = resultSet.getMetaData();
156
157         // determina o número de linhas em ResultSet
158         resultSet.last(); // move para a última linha
159         numberOfRows = resultSet.getRow(); // obtém o número de linha
160
161         // notifica a JTable de que modelo foi alterado
162         fireTableStructureChanged();
163      } // fim do método setQuery
164
165      // fecha Statement e Connection
```

```
166       public void disconnectFromDatabase()
167       {
168          if ( connectedToDatabase )
169          {
170             // fecha Statement e Connection
171             try
172             {
173                resultSet.close();
174                statement.close();
175                connection.close();
176             } // fim do try
177             catch ( SQLException sqlException )
178             {
179                sqlException.printStackTrace();
180             } // fim do catch
181             finally  // atualiza status de conexão de banco de dados
182             {
183                connectedToDatabase = false;
184             } // fim de finally
185          } // fim do if
186       } // fim do método disconnectFromDatabase
187    }  // fim da classe ResultSetTableModel
```

Figura 28.25 | Um `TableModel` que fornece dados `ResultSet` para um `JTable`.

O construtor `ResultSetTableModel` (linhas 30–46) aceita cinco argumentos `String` — o nome de classe driver, o URL do banco de dados, o nome de usuário, a senha e a consulta padrão a ser executada. O construtor lança qualquer exceção que ocorra no seu corpo de volta para o aplicativo que criou o objeto `ResultSetTableModel`, de modo que o aplicativo pode determinar como tratar a exceção (por exemplo, informar um erro e terminar o aplicativo). A linha 34 estabelece uma conexão com o banco de dados. As linhas 37–39 invocam o método `Connection` `createStatement` para criar um objeto `Statement`. Esse exemplo usa uma versão de método `createStatement` que aceita dois argumentos — o tipo de conjunto de resultados (result set) e a concorrência do conjunto de resultados. O **tipo de conjunto de resultados** (Figura 28.26) especifica se o cursor `ResultSet` é capaz de rolar em ambas as direções ou apenas encaminhar e se o `ResultSet` é sensível a alterações feitas nos dados subjacentes.

Constante `ResultSet`	Descrição
`TYPE_FORWARD_ONLY`	Especifica que o cursor de um `ResultSet` pode mover apenas para frente (isto é, da primeira para a última linha no `ResultSet`).
`TYPE_SCROLL_INSENSITIVE`	Especifica que cursor de um `ResultSet` pode rolar em qualquer direção e que as alterações feitas no `ResultSet` durante o processamento do `ResultSet` não são refletidas no `ResultSet` a menos que o programa consulte novamente o banco de dados.
`TYPE_SCROLL_SENSITIVE`	Especifica que cursor de um `ResultSet` pode rolar em qualquer direção e que as alterações feitas no `ResultSet` durante o processamento do `ResultSet` são refletidas imediatamente no `ResultSet`.

Figura 28.26 | Constantes `ResultSet` para especificar o tipo `ResultSet`.

Dica de portabilidade 28.3

Alguns drivers JDBC não suportam `ResultSet`s roláveis. Nesses casos, o driver em geral retorna um `ResultSet` em que o cursor só pode mover para frente. Para informações adicionais, consulte sua documentação de driver de banco de dados.

Erro comum de programação 28.11

Tentar mover o cursor para trás por um `ResultSet` quando o driver de banco de dados não suportar rolagem para trás causa uma `SQLFeatureNotSupportedException`.

`ResultSet`s que são sensíveis a alterações refletem essas modificações imediatamente depois de elas serem feitas com os métodos da interface `ResultSet`. Se um `ResultSet` não for sensível a alterações, a consulta que produziu o `ResultSet` deve ser novamente executada para refletir qualquer alteração feita. A **concorrência do conjunto de resultados** (Figura 28.27) especifica se o `ResultSet` pode ser atualizado com os métodos de atualização do `ResultSet`.

Constante `ResultSet` `static` de concorrência	Descrição
`CONCUR_READ_ONLY`	Especifica que um `ResultSet` não pode ser atualizado (isto é, alterações no conteúdo do `ResultSet` não podem ser refletidas no banco de dados com os métodos de atualização do `ResultSet`).
`CONCUR_UPDATABLE`	Especifica que um `ResultSet` pode ser atualizado (isto é, as alterações no conteúdo do `ResultSet` podem ser refletidas no banco de dados com os métodos *update* do `ResultSet`).

Figura 28.27 | Constantes `ResultSet` para especificar propriedades de resultado.

Dica de portabilidade 28.4

Alguns drivers JDBC não suportam `ResultSets` atualizáveis. Nesses casos, o driver em geral retorna um `ResultSet` de leitura. Para informações adicionais, consulte sua documentação de driver de banco de dados.

Erro comum de programação 28.12

Tentar atualizar um `ResultSet` quando o driver de banco de dados não suporta `ResultSets` atualizáveis causa `SQLFeatureNotSupportedExceptions`.

Esse exemplo utiliza um `ResultSet` que é rolável, insensível às alterações e apenas de leitura. A linha 45 invoca o método `ResultSetTableModel` `setQuery` (linhas 144–163) para realizar a consulta padrão.

O método `getColumnClass` (linhas 49–69) retorna um objeto `Class` que representa a superclasse de todos os objetos em uma coluna particular. A `JTable` utiliza essas informações para configurar o renderizador de célula e editor de célula padrão para essa coluna na `JTable`. A linha 58 utiliza o método `ResultSetMetaData` **`getColumnClassName`** para obter o nome de classe completamente qualificado para a coluna especificada. A linha 61 carrega a classe e retorna o correspondente objeto `Class`. Se ocorrer uma exceção, `catch` nas linhas 63–66 imprime um rastreamento de pilha e a linha 68 retorna `Object.class` — a instância `Class` que representa a classe `Object` — como o tipo padrão. [*Nota*: A linha 58 utiliza o argumento `column + 1`. Como os arrays, os números de linha e coluna da `JTable` são contados a partir de 0. Mas os números de linha e coluna no `ResultSet` são contados a partir de 1. Portanto, ao processar as linhas ou colunas `ResultSet` para uso em uma `JTable`, é necessário adicionar 1 ao número de linha ou coluna para manipular a linha ou coluna `ResultSet` apropriada].

O método `getColumnCount` (linhas 72–89) retorna o número de colunas no `ResultSet` subjacente do modelo. A linha 81 utiliza o método `ResultSetMetaData` **`getColumnCount`** para obter o número de colunas no `ResultSet`. Se ocorrer uma exceção, o `catch` nas linhas 83–86 imprime um rastreamento de pilha e a linha 88 retorna 0 como o número padrão de colunas.

O método `getColumnName` (linhas 92–109) retorna o nome da coluna no `ResultSet` subjacente do modelo. A linha 101 utiliza o método `ResultSetMetaData` `getColumnName` para obter o nome de coluna do `ResultSet`. Se ocorrer uma exceção, `catch` nas linhas 103–106 imprime um rastreamento de pilha e a linha 108 retorna a string vazia como o nome padrão de coluna.

O método `getRowCount` (linhas 112–119) retorna o número de linhas na `ResultSet` subjacente do modelo. Quando o método `setQuery` (linhas 144–163) realizar uma consulta, ele armazena o número de linhas na variável `numberOfRows`.

O método `getValueAt` (linhas 122–141) retorna o `Object` em uma linha e coluna particular do `ResultSet` subjacente do modelo. A linha 132 utiliza o método `ResultSet` **`absolute`** para posicionar o cursor `ResultSet` em uma linha específica. A linha 133 utiliza o método `ResultSet` `getObject` para obter `Object` em uma coluna específica da linha atual. Se ocorrer uma exceção, o `catch` nas linhas 135–138 imprime um rastreamento de pilha e a linha 140 retorna uma string vazia como o valor padrão.

O método `setQuery` (linhas 144–163) executa a consulta que ele recebe como um argumento para obter um novo `ResultSet` (linha 152). A linha 155 obtém o `ResultSetMetaData` para o novo `ResultSet`. A linha 158 utiliza o método `ResultSet` **`last`** para posicionar o cursor `ResultSet` na última linha no `ResultSet` . [*Nota*: Isso pode ser lento se a tabela contiver muitas linhas.] A linha 159 usa o método `ResultSet` **`getRow`** para obter o número de linha para a linha atual no `ResultSet`. A linha 162 invoca o método **`fireTableStructureChanged`** (herdado da classe `AbstractTableModel`) para notificar qualquer `JTable` que utiliza esse objeto `ResultSetTableModel` como seu modelo que a estrutura do modelo alterou. Isso faz com que a `JTable` preencha novamente suas linhas e colunas com os novos dados de `ResultSet`. O método `setQuery` lança qualquer exceção que ocorra no seu corpo de volta ao aplicativo que invocou o `setQuery`.

O método `disconnectFromDatabase` (linhas 166–186) implementa um método de terminação correto da classe `ResultSetTableModel`. Um projetista de classe deve fornecer um método `public` que os clientes da classe devem invocar explicitamente para liberar recursos que um objeto utilizou. Nesse caso, o método `disconnectFromDatabase` fecha a instrução e a conexão de banco de dados (linhas 173–175), que são consideradas recursos limitados. Os clientes da classe `ResultSetTableModel` devem sempre invocar esse método quando a instância dessa classe não for mais necessária. Antes de liberar recursos, a linha 168 verifica se a conexão já foi terminada. Se não, o método avança. Observe que os outros métodos na classe `ResultSetTableModel` lançam, cada um, uma `IllegalStateException` se

`connectedToDatabase` for `false`. O método `disconnectFromDatabase` configura `connectedToDatabase` como `false` (linha 183) para assegurar que os clientes não utilizam uma instância de `ResultSetTableModel` depois que a instância já foi terminada. `IllegalStateException` é uma exceção das bibliotecas Java que é adequada para indicar essa condição de erro.

Classe `DisplayQueryResults`

A classe `DisplayQueryResults` (Figura 28.28) implementa a GUI do aplicativo e interage com o `ResultSetTableModel` via um objeto `JTable`. Esse aplicativo também demonstra as capacidades de classificação e filtragem de `JTable` introduzidas no Java SE 6.

```
1  // Figura 28.28: DisplayQueryResults.java
2  // Exibe o conteúdo da tabela Authors no banco de dados de livros
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.WindowAdapter;
7  import java.awt.event.WindowEvent;
8  import java.sql.SQLException;
9  import java.util.regex.PatternSyntaxException;
10 import javax.swing.JFrame;
11 import javax.swing.JTextArea;
12 import javax.swing.JScrollPane;
13 import javax.swing.ScrollPaneConstants;
14 import javax.swing.JTable;
15 import javax.swing.JOptionPane;
16 import javax.swing.JButton;
17 import javax.swing.Box;
18 import javax.swing.JLabel;
19 import javax.swing.JTextField;
20 import javax.swing.RowFilter;
21 import javax.swing.table.TableRowSorter;
22 import javax.swing.table.TableModel;
23
24 public class DisplayQueryResults extends JFrame
25 {
26    // URL de banco de dados, nome de usuário e senha
27    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
28    static final String USERNAME = "deitel";
29    static final String PASSWORD = "deitel";
30
31    // a consulta padrão recupera todos os dados da tabela Authors
32    static final String DEFAULT_QUERY = "SELECT * FROM Authors";
33
34    private ResultSetTableModel tableModel;
35    private JTextArea queryArea;
36
37    // cria o ResultSetTableModel e GUI
38    public DisplayQueryResults()
39    {
40       super( "Displaying Query Results" );
41
42       // cria o ResultSetTableModel e exibe a tabela de banco de dados
43       try
44       {
45          // cria o TableModel para resultados da consulta SELECT * FROM authors
46          tableModel = new ResultSetTableModel( DATABASE_URL,
47             USERNAME, PASSWORD, DEFAULT_QUERY );
48
49          // configura JTextArea em que o usuário digita consultas
50          queryArea = new JTextArea( DEFAULT_QUERY, 3, 100 );
51          queryArea.setWrapStyleWord( true );
52          queryArea.setLineWrap( true );
53
54          JScrollPane scrollPane = new JScrollPane( queryArea,
55             ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
56             ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
57
58          // configura o JButton para enviar consulta
59          JButton submitButton = new JButton( "Submit Query" );
```

```
60
61          // cria o Box para gerenciar o posicionamento da queryArea e do
62          // submitButton na GUI
63          Box boxNorth = Box.createHorizontalBox();
64          boxNorth.add( scrollPane );
65          boxNorth.add( submitButton );
66
67          // cria JTable com base no tableModel
68          JTable resultTable = new JTable( tableModel );
69
70          JLabel filterLabel = new JLabel( "Filter:" );
71          final JTextField filterText = new JTextField();
72          JButton filterButton = new JButton( "Apply Filter" );
73          Box boxSouth = Box.createHorizontalBox();
74
75          boxSouth.add( filterLabel );
76          boxSouth.add( filterText );
77          boxSouth.add( filterButton );
78
79          // posiciona os componentes GUI no painel de conteúdo
80          add( boxNorth, BorderLayout.NORTH );
81          add( new JScrollPane( resultTable ), BorderLayout.CENTER );
82          add( boxSouth, BorderLayout.SOUTH );
83
84          // cria evento ouvinte para submitButton
85          submitButton.addActionListener(
86
87             new ActionListener()
88             {
89                // passa consulta para modelo de tabela
90                public void actionPerformed( ActionEvent event )
91                {
92                   // realiza uma nova consulta
93                   try
94                   {
95                      tableModel.setQuery( queryArea.getText() );
96                   } // fim do try
97                   catch ( SQLException sqlException )
98                   {
99                      JOptionPane.showMessageDialog( null,
100                        sqlException.getMessage(), "Database error",
101                        JOptionPane.ERROR_MESSAGE );
102
103                     // tenta recuperar a partir da consulta de usuário inválida
104                     // executando consulta padrão
105                     try
106                     {
107                        tableModel.setQuery( DEFAULT_QUERY );
108                        queryArea.setText( DEFAULT_QUERY );
109                     } // fim do try
110                     catch ( SQLException sqlException2 )
111                     {
112                        JOptionPane.showMessageDialog( null,
113                           sqlException2.getMessage(), "Database error",
114                           JOptionPane.ERROR_MESSAGE );
115
116                        // assegura que a conexão de banco de dados está fechada
117                        tableModel.disconnectFromDatabase();
118
119                        System.exit( 1 ); // termina o aplicativo
120                     } // fim do catch
121                  } // fim do catch externo
122               } // fim do método actionPerformed
123            }  // fim da classe inner
124         ); // fim da chamada para addActionListener
125
126         final TableRowSorter< TableModel > sorter =
127            new TableRowSorter< TableModel >( tableModel );
128         resultTable.setRowSorter( sorter );
```

```
         setSize( 500, 250 ); // configura o tamanho da janela
         setVisible( true ); // exibe a janela

         // cria o ouvinte para filterButton
         filterButton.addActionListener(
            new ActionListener()
            {
               // passa o texto do filtro para o ouvinte
               public void actionPerformed( ActionEvent e )
               {
                  String text = filterText.getText();

                  if ( text.length() == 0 )
                     sorter.setRowFilter( null );
                  else
                  {
                     try
                     {
                        sorter.setRowFilter(
                           RowFilter.regexFilter( text ) );
                     } // fim do try
                     catch ( PatternSyntaxException pse )
                     {
                        JOptionPane.showMessageDialog( null,
                           "Bad regex pattern", "Bad regex pattern",
                           JOptionPane.ERROR_MESSAGE );
                     } // fim do catch
                  } // fim de else
               } // fim do método actionPerfomed
            } // fim da classe interna anônima
         ); // fim da chamada para addActionListener
      } // fim do try
      catch ( SQLException sqlException )
      {
         JOptionPane.showMessageDialog( null, sqlException.getMessage(),
            "Database error", JOptionPane.ERROR_MESSAGE );

         // assegura que a conexão de banco de dados está fechada
         tableModel.disconnectFromDatabase();

         System.exit( 1 ); // termina o aplicativo
      } // fim do catch

      // dispõe da janela quando o usuário fecha o aplicativo (isso sobrescreve
      // o padrão de HIDE_ON_CLOSE)
      setDefaultCloseOperation( DISPOSE_ON_CLOSE );

      // assegura que a conexão de banco de dados é fechada quando usuário fecha o aplicativo
      addWindowListener(

         new WindowAdapter()
         {
            // desconecta-se do banco de dados e sai quando a janela for fechada
            public void windowClosed( WindowEvent event )
            {
               tableModel.disconnectFromDatabase();
               System.exit( 0 );
            } // fim do método windowClosed
         } // fim da classe WindowAdapter interna
      ); // fim da chamada a addWindowListener
   } // fim do construtor DisplayQueryResults

   // executa o aplicativo
   public static void main( String args[] )
   {
      new DisplayQueryResults();
   } // fim de main
} // fim da classe DisplayQueryResults
```

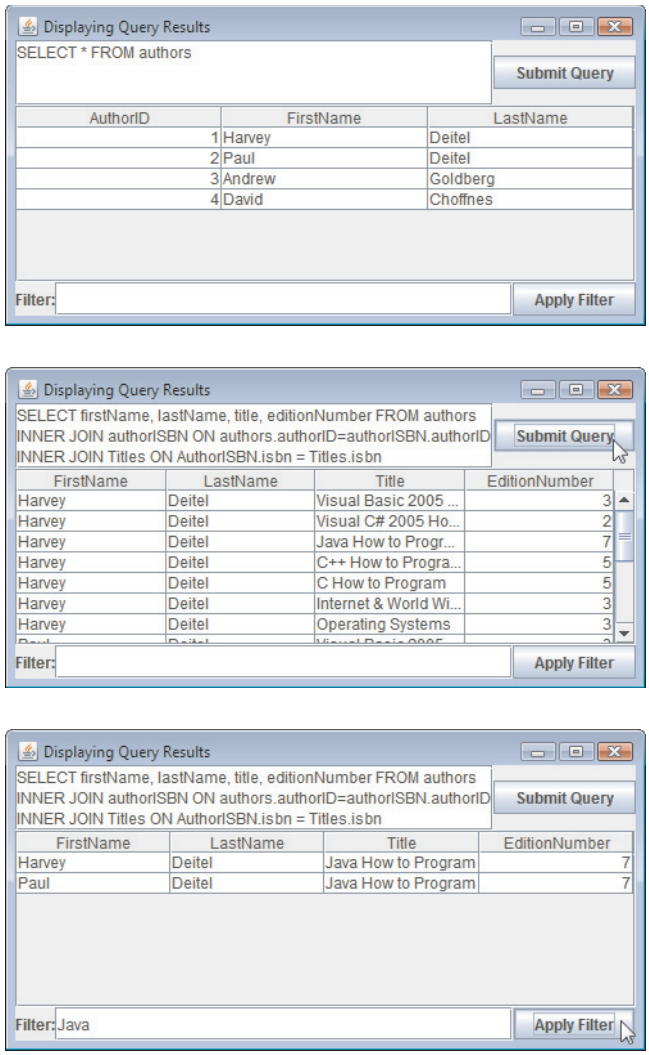


Figura 28.28 | Exibindo o conteúdo do banco de dados `books`.

As linhas 27–29 e 32 declaram o URL, o nome de usuário, a senha e a consulta padrão que são passados ao construtor `ResultSetTableModel` para fazer a conexão inicial com o banco de dados e realizar a consulta padrão. O construtor `DisplayQueryResults` (linhas 38–189) cria um objeto `ResultSetTableModel` e a GUI do aplicativo. A linha 68 cria o objeto `JTable` e passa um objeto `ResultSetTable-Model` para o construtor `JTable`, que então registra a `JTable` como um ouvinte para `TableModelEvents` gerados pelo `ResultSetTable-Model`.

Observe que as variáveis locais `filterText` (linha 71) e `sorter` (linhas 126–127) são declaradas `final`. Essas duas são utilizadas a partir de um handler de evento que é implementado como uma classe interna anônima (linhas 134–158). Qualquer variável local que for utilizada em uma classe interna anônima *deve* ser declarada final; caso contrário, ocorre um erro de compilação.

As linhas 85–124 registram um handler de evento para `submitButton` em que o usuário clica para submeter uma consulta para o banco de dados. Quando o usuário clica no botão, o método `actionPerformed` (linhas 90–122) invoca o método `setQuery` da classe `ResultSetTableModel` para executar a nova consulta (linha 95). Se a consulta do usuário falhar (por exemplo, por causa de um erro de sintaxe na entrada do usuário), as linhas 107–108 executam a consulta padrão. Se a consulta padrão também falhar, haverá um erro mais grave, então a linha 117 assegura que a conexão de banco de dados é fechada e a linha 119 fecha o programa. As capturas de tela na Figura 28.28 mostram os resultados de duas consultas. A primeira captura de tela mostra a consulta padrão que recupera todos os dados da tabela `Authors` do banco de dados `books`. A segunda captura de tela mostra uma consulta que seleciona o nome e o sobrenome de cada autor a partir da tabela `Authors` e combina essas informações com o título e número de edição da tabela `Titles`. Tente inserir suas próprias consultas na área de texto e clique no botão Submit Query para executar a consulta.

No Java SE 6, `JTables` agora permitem a usuários classificarem linhas pelos dados em uma coluna específica. As linhas 126–127 utilizam a classe **`TableRowSorter`** (do pacote **`javax.swing.table`**) para criar um objeto que utiliza nosso `ResultSetTableModel` para classificar linhas na `JTable` que exibe resultados de consulta. Quando o usuário clicar no título de determinada coluna `JTable`, o `TableRowSorter` interage com o `TableModel` subjacente para reordenar as linhas com base nos dados dessa coluna. A linha 128 utiliza o método `JTable` **`setRowSorter`** para especificar o `TableRowSorter` para `resultTable`.

`JTables` podem mostrar agora subconjuntos dos dados do `TableModel` subjacente. Isso é conhecido como filtragem dos dados. As linhas 133–159 registram um handler de evento para o `filterButton` no qual o usuário clica para filtrar os dados. No método `actionPerformed` (linhas 137–157), a linha 139 obtém o texto de filtro. Se o usuário não tiver especificado o texto de filtro, a linha 142 utilizará o método `JTable` **`setRowFilter`** para remover algum filtro anterior configurando o filtro como `null`. Caso contrário, as linhas 147–148 utilizam `setRowFilter` para especificar um **`RowFilter`** (do pacote `javax.swing`) com base na entrada do usuário. A classe `RowFilter` fornece vários métodos para criar filtros. O método `static` **`regexFilter`** recebe uma `String` contendo um padrão de expressão regular como seu argumento e um conjunto opcional de índices que especifica quais colunas filtrar. Se nenhum índice for especificado, então todas as colunas são pesquisadas. Nesse exemplo, o padrão de expressão regular é o texto que o usuário digitou. Assim que o filtro for configurado, o dado exibido no `JTable` é atualizado com base no `TableModel` filtrado.

As linhas 177–188 registram um **`WindowListener`** para o evento **`windowClosed`**, que ocorre quando o usuário fecha a janela. Como `WindowListeners` podem tratar vários eventos de janela, estendemos a classe **`WindowAdapter`** e sobrescrevemos apenas o handler de evento `windowClosed`.

28.9 Interface RowSet

Nos exemplos anteriores, você aprendeu a consultar um banco de dados estabelecendo explicitamente uma `Connection` com o banco de dados, preparando uma `Statement` para consultar o banco de dados e executar a consulta. Nesta seção, demonstramos a interface **`RowSet`**, que configura a conexão de banco de dados e prepara automaticamente as instruções de consulta. A interface `RowSet` fornece vários métodos *set* que permitem especificar as propriedades necessárias para estabelecer uma conexão (como o URL do banco de dados, o nome de usuário e a senha do banco de dados) e criar uma `Statement` (como uma consulta). `RowSet` também fornece vários métodos *get* que retornam essas propriedades.

Há dois tipos de objetos `RowSet` — conectado e desconectado. Um objeto **`RowSet` conectado** conecta-se ao banco de dados uma vez e permanece conectado enquanto o objeto estiver em uso. Um objeto **`RowSet` desconectado** conecta-se ao banco de dados, executa uma consulta para recuperar os dados do banco de dados e depois fecha a conexão. Um programa pode alterar os dados em um `RowSet` desconectado enquanto ele estiver desconectado. Dados modificados podem ser atualizados no banco de dados depois que um `RowSet` desconectado restabelecer a conexão com o banco de dados.

O pacote **`javax.sql.rowset`** contém duas subinterfaces de `RowSet` — `JdbcRowSet` e `CachedRowSet`. **`JdbcRowSet`**, um `RowSet` conectado, atua como um empacotador em torno de um objeto `ResultSet` e permite aos programadores percorrer e atualizar as linhas no `ResultSet`. Lembre-se de que, por padrão, um objeto `ResultSet` é não rolável e apenas de leitura — você deve configurar explicitamente a constante do tipo result set como `TYPE_SCROLL_INSENSITIVE` e configurar a constante de concorrência do result set como `CONCUR_UPDATABLE` para fazer um objeto `ResultSet` rolável e atualizável. Um objeto `JdbcRowSet` é rolável e atualizável por padrão. **`CachedRowSet`**, um `RowSet` desconectado, armazena os dados em cache de um `ResultSet` na memória e desconecta-se do banco de dados. Como `JdbcRowSet`, um objeto `CachedRowSet` é rolável e atualizável por padrão. Um objeto `CachedRowSet` também é serializável, então ele pode ser passado entre aplicativos Java por uma rede, como a internet. Entretanto, `CachedRowSet` tem uma limitação — a quantidade de dados que pode ser armazenada na memória é limitada. O pacote `javax.sql.rowset` contém três outras subinterfaces de `RowSet`. Para obter detalhes dessas interfaces, visite `java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html`.

Dica de portabilidade 28.5

Um `RowSet` pode fornecer a capacidade de rolagem para drivers que não suportam `ResultSets` roláveis.

A Figura 28.29 reimplementa o exemplo da Figura 28.23 utilizando um `RowSet`. Em vez de estabelecer a conexão e criar um `Statement` explicitamente, a Figura 28.29 utiliza um objeto `JdbcRowSet` para criar uma `Connection` e um `Statement` automaticamente.

```
1  // Figura 28.29: JdbcRowSetTest.java
2  // Exibindo o conteúdo da tabela authors com JdbcRowSet.
3  import java.sql.ResultSetMetaData;
4  import java.sql.SQLException;
5  import javax.sql.rowset.JdbcRowSet;
6  import com.sun.rowset.JdbcRowSetImpl; // Implementação JdbcRowSet da Sun
7
8  public class JdbcRowSetTest
9  {
10    // nome do driver JDBC e URL do banco de dados
11    static final String DATABASE_URL = "jdbc:mysql://localhost/books";
12    static final String USERNAME = "deitel";
13    static final String PASSWORD = "deitel";
14
15    // construtor conecta-se ao banco de dados, consulta banco de dados, processa
16    // os resultados e os exibe na janela
17    public JdbcRowSetTest()
```

```
18      {
19         // conecta-se ao banco de dados books e o consulta
20         try
21         {
22            // especifica propriedades de JdbcRowSet
23            JdbcRowSet rowSet = new JdbcRowSetImpl();
24            rowSet.setUrl( DATABASE_URL ); // configura o URL de banco de dados
25            rowSet.setUsername( USERNAME ); // configura o nome de usuário
26            rowSet.setPassword( PASSWORD ); // configura a senha
27            rowSet.setCommand( "SELECT * FROM Authors" ); // configura a consulta
28            rowSet.execute(); // executa a consulta
29
30            // processa resultados da consulta
31            ResultSetMetaData metaData = rowSet.getMetaData();
32            int numberOfColumns = metaData.getColumnCount();
33            System.out.println( "Authors Table of Books Database:\n" );
34
35            // exibe o cabeçalho rowset
36            for ( int i = 1; i <= numberOfColumns; i++ )
37               System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
38            System.out.println();
39
40            // exibe cada linha
41            while (rowSet.next() )
42            {
43               for ( int i = 1; i <= numberOfColumns; i++ )
44                  System.out.printf( "%-8s\t", rowSet.getObject( i ) );
45               System.out.println();
46            } // fim do while
47
48            // fecha o ResultSet, Statement e Connection subjacentes
49            rowSet.close();
50         } // fim do try
51         catch ( SQLException sqlException )
52         {
53            sqlException.printStackTrace();
54            System.exit( 1 );
55         } // fim do catch
56      } // fim do construtor DisplayAuthors
57
58      // carrega o aplicativo
59      public static void main( String args[] )
60      {
61         JdbcRowSetTest application = new JdbcRowSetTest();
62      } // fim de main
63   } // fim da classe JdbcRowSetTest
```

```
Authors Table of Books Database:

AuthorID        FirstName       LastName
1               Harvey          Deitel
2               Paul            Deitel
3               Andrew          Goldberg
4               David           Choffnes
```

Figura 28.29 | Exibindo a tabela `Authors` com `JdbcRowSet`.

O pacote **com.sun.rowset** fornece implementações de referência da Sun para as interfaces no pacote `javax.sql.rowset`. A linha 23 usa a implementação de referência da Sun para a interface `JdbcRowSet` — **`JdbcRowSetImpl`** — a fim de criar um objeto `JdbcRowSet`. Utilizamos a classe `JdbcRowSetImpl` aqui para demonstrar a capacidade da interface `JdbcRowSet`. Alguns bancos de dados podem fornecer suas próprias implementações de `RowSet`.

As linhas 24–26 configuram as propriedades `RowSet` que o `DriverManager` utiliza para estabelecer uma conexão de banco de dados. A linha 24 invoca o método `JdbcRowSet` **`setUrl`** para especificar o URL de banco de dados. A linha 25 invoca o método `JdbcRowSet` **`setUsername`** para especificar o nome de usuário. A linha 26 invoca o método `JdbcRowSet` **`setPassword`** para especificar a senha. A linha 27 invoca o método `JdbcRowSet` **`setCommand`** para especificar a consulta SQL que será utilizada para preencher o `RowSet`. A linha 28 invoca o método `JdbcRowSet` **`execute`** para executar a consulta de SQL. O método `execute` realiza quatro ações — estabelece

uma `Connection`, prepara a consulta `Statement`, executa a consulta e armazena o `ResultSet` retornado pela consulta. `Connection`, `Statement` e `ResultSet` são encapsulados no objeto `JdbcRowSet`.

O código restante é quase idêntico ao da Figura 28.23, exceto que a linha 31 obtém um objeto `ResultSetMetaData` do `JdbcRowSet`, a linha 41 utiliza o método `next` de `JdbcRowSet` para obter a próxima linha do resultado e a linha 44 utiliza o método `next` de `JdbcRowSet` para obter a próxima linha do resultado e a linha 47 utiliza o método `getObject` do `JdbcRowSet` para obter o valor de uma coluna. A linha 49 invoca o método `JdbcRowSet` **`close`**, que fecha o `ResultSet`, a `Statement` e a `Connection` encapsulados do `RowSet`. Em um `CachedRowSet`, invocar `close` também libera recursos mantidos por esse `RowSet`. Observe que a saída desse aplicativo é exatamente a mesma saída da Figura 28.23.

28.10 Java DB/Apache Derby

No JDK 6, a Sun Microsystems agora fornece o código-fonte aberto, **Java DB** de banco de dados Java puro (a versão com a marca Sun do Apache Derby) com o JDK. Na Seção 28.11, utilizamos o Java DB para demonstrar `PreparedStatements`. Antes de poder executar o aplicativo na próxima seção, você deve configurar o banco de dados `AddressBook` no Java DB. A Seção 28.11 utiliza a versão incorporada do Java DB. Há também uma versão de rede que executa de modo semelhante ao DBMS MySQL introduzido anteriormente no capítulo. Para o propósito dos passos a seguir, supomos que você esteja executando o Microsoft Windows com o Java instalado em sua localização padrão. [*Nota*: Usuários do Mac OS X talvez precisem baixar o Apache Derby e instalá-lo (`db.apache.org/derby/`) se o Java DB não estiver instalado por padrão. Usuários do Linux devem visitar `developers.sun.com/javadb/reference/docs/` e ler "Getting Started with Java DB" para obter instruções de configuração e instalação.]

1. O Java DB vem com vários arquivos em lote para configurá-lo e executá-lo. Antes de executar esses arquivos em lote a partir de um prompt de comando, você deve configurar a variável de ambiente `JAVA_HOME` para referir-se ao diretório de instalação `C:\Program Files\Java\jdk1.6.0_11` do JDK. [*Nota*: Talvez você precise mudar isso de acordo com a versão do JDK instalada no seu computador.]
2. Abra o arquivo de lote `setEmbeddedCP.bat` (localizado em `C:\Program Files\Sun\JavaDB\bin`) em um editor de textos, como o Notepad. Localize a linha

```
@rem set DERBY_INSTALL=
```

e mude-a para

```
@set DERBY_INSTALL=C:\Program Files\Sun\JavaDB
```

Salve suas alterações e feche esse arquivo.

3. Abra um prompt de comando e mude para o diretório `C:\Program Files\Sun\JavaDB\bin`. Então, digite `setEmbeddedCP.bat` e pressione *Enter* para configurar as variáveis de ambiente requeridas pelo Java DB.
4. Um banco de dados Java DB incorporado deve residir na mesma localização do aplicativo que manipula o banco de dados. Por essa razão, mude para o diretório que contém o código de figuras 28.30–28.32. Esse diretório contém um script SQL `address.sql` que constrói o banco de dados `AddressBook`.
5. Execute o comando

```
"C:\Program Files\Sun\JavaDB\bin\ij"
```

para iniciar a ferramenta de linha de comando para interagir com o Java DB. As aspas duplas são necessárias porque o caminho contém um espaço. Isso exibirá o prompt `ij>`.

6. No prompt `ij>` digite

```
connect 'jdbc:derby:AddressBook;create=true;user=deitel;
password=deitel';
```

a fim de criar o banco de dados `AddressBook` no diretório atual. Esse comando também cria o usuário `deitel` com a senha `deitel` para acessar o banco de dados.

7. Para criar a tabela de banco de dados e inserir dados de exemplo no banco de dados, digite

```
run 'address.sql';
```

8. Para terminar a ferramenta de linha de comando Java DB, digite

```
exit;
```

Você está pronto agora para executar o aplicativo `AddressBook` da Seção 28.11. Observe que o MySQL (ou qualquer outro banco de dados que suporte JDBC `PreparedStatements`) também pode ser utilizado na Seção 28.11.

28.11 PreparedStatements

A interface **PreparedStatement** permite criar instruções SQL compiladas que executam mais eficientemente do que os objetos Statement. PreparedStatements também podem especificar parâmetros, tornando-os mais flexíveis do que as Statements. Os programas podem executar a mesma consulta repetidamente com diferentes valores de parâmetro. Por exemplo, no banco de dados books, você poderia querer localizar todos os títulos de livro de um autor com um sobrenome e nome específico e poderia querer executar essa consulta para vários autores. Com uma PreparedStatement, essa consulta é definida como mostrado a seguir:

```
PreparedStatement authorBooks = connection.prepareStatement(
   "SELECT LastName, FirstName, Title " +
   "FROM Authors INNER JOIN AuthorISBN " +
      "ON Authors.AuthorID=AuthorISBN.AuthorID " +
   "INNER JOIN Titles " +
      "ON AuthorISBN.ISBN=Titles.ISBN " +
   "WHERE LastName = ? AND FirstName = ?" );
```

Os dois pontos de interrogação (?) na linha última da instrução SQL anterior são um espaço reservado para valores que serão passados como parte da consulta ao banco de dados. Antes de executar uma PreparedStatement, o programa deve especificar os valores de parâmetro utilizando os métodos *set* da interface PreparedStatement.

Para a consulta anterior, ambos os parâmetros são strings que podem ser configuradas com o método PreparedStatement **setString** como mostrado a seguir:

```
authorBooks.setString( 1, "Deitel" );
authorBooks.setString( 2, "Paul" );
```

O primeiro argumento do método setString representa o número do parâmetro sendo configurado, e o segundo argumento é o valor desse parâmetro. Os números de parâmetro são contados a partir de 1, iniciando com o primeiro ponto de interrogação (?). Quando o programa executa a PreparedStatement anterior com os valores de parâmetro mostrados aqui, a SQL passada para o banco de dados é

```
SELECT LastName, FirstName, Title
FROM Authors INNER JOIN AuthorISBN
   ON Authors.AuthorID=AuthorISBN.AuthorID
INNER JOIN Titles
   ON AuthorISBN.ISBN=Titles.ISBN
WHERE LastName = 'Deitel' AND FirstName = 'Paul'
```

O método setString escapa automaticamente valores de parâmetro String conforme necessário. Por exemplo, se o sobrenome for O'Brien, a instrução

```
authorBooks.setString( 1, "O'Brien" );
```

escapa o caractere ' em O'Brien substituindo-o por dois caracteres de aspa simples.

Dica de desempenho 28.2

PreparedStatements são mais eficientes do que Statements ao executar instruções SQL múltiplas vezes e com diferentes valores de parâmetro.

Dica de prevenção de erro 28.2

Utilize PreparedStatements com parâmetros de consultas que recebem valores String como argumentos para assegurar que as Strings sejam colocadas corretamente entre aspas na instrução SQL.

A interface PreparedStatement fornece métodos *set* para cada tipo SQL suportado. É importante utilizar o método *set* correto para o tipo SQL do parâmetro no banco de dados — SQLExceptions ocorrem quando um programa tenta converter um valor de parâmetro em um tipo incorreto. Para uma lista completa dos métodos *set* da interface PreparedStatement, consulte java.sun.com/javase/6/docs/api/java/sql/PreparedStatement.html.

Aplicativo AddressBook que usa PreparedStatements

Agora apresentamos um aplicativo de catálogo de endereços que permite navegar por entradas existentes, adicionar novas entradas e pesquisar por entradas com um sobrenome específico. O nosso banco de dados Java DB AddressBook contém uma tabela Addresses com as colunas addressID, FirstName, LastName, Email e PhoneNumber. A coluna addressID é a chamada coluna de identidade. Essa é a maneira padrão SQL para representar uma coluna autoincrementada. O script SQL que fornecemos para esse banco de dados utiliza a

palavra-chave **IDENTITY** de SQL para marcar a coluna `addressID` como uma coluna de identidade. Para obter mais informações sobre como utilizar a palavra-chave `IDENTITY` e criar bancos de dados, consulte o guia do desenvolvedor em Java DB em `developers.sun.com/docs/javadb/10.4.2.0/devguide/derbydev.pdf`.

Nosso aplicativo de catálogo de endereços consiste em três classes — `Person` (Figura 28.30), `PersonQueries` (Figura 28.31) e `AddressBookDisplay` (Figura 28.32). `Person` é uma classe simples que representa uma pessoa no catálogo de endereços. A classe contém campos para o ID de endereço, o nome, o sobrenome, o endereço de e-mail e o número de telefone, bem como os métodos *set* e *get* para manipular esses campos.

```
1  // Figura 28.30: Person.java
2  // Classe Person que representa uma entrada em um catálogo de endereços.
3  public class Person
4  {
5     private int addressID;
6     private String firstName;
7     private String lastName;
8     private String email;
9     private String phoneNumber;
10
11    // construtor sem argumento
12    public Person()
13    {
14    } // fim do construtor sem argumentos da classe Stack
15
16    // construtor
17    public Person( int id, String first, String last,
18       String emailAddress, String phone )
19    {
20       setAddressID( id );
21       setFirstName( first );
22       setLastName( last );
23       setEmail( emailAddress );
24       setPhoneNumber( phone );
25    } // fim do construtor CommissionEmployee de cinco argumentos
26
27    // configura o IDdeEndereço
28    public void setAddressID( int id )
29    {
30       addressID = id;
31    } // fim do método setAddressID
32
33    // retorna o IDdeEndereço
34    public int getAddressID()
35    {
36       return addressID;
37    } // fim do método getAddressID
38
39    // estabelece o nome
40    public void setFirstName( String first )
41    {
42       firstName = first;
43    } // fim do método setFirstName
44
45    // retorna o nome
46    public String getFirstName()
47    {
48       return firstName;
49    } // fim do método getFirstName
50
51    // configura o sobrenome
52    public void setLastName( String last )
53    {
54       lastName = last;
55    } // fim do método setLastName
56
57    // retorna o sobrenome
58    public String getLastName()
59    {
```

```
60          return lastName;
61       } // fim do método getLastName
62
63       // configura o endereço de e-mail
64       public void setEmail( String emailAddress )
65       {
66          email = emailAddress;
67       } // fim do método setEmail
68
69       // retorna o endereço de e-mail
70       public String getEmail()
71       {
72          return email;
73       } // fim do método getEmail
74
75       // configura o número de telefone
76       public void setPhoneNumber( String phone )
77       {
78          phoneNumber = phone;
79       } // fim do método setPhoneNumber
80
81       // retorna o número de telefone
82       public String getPhoneNumber()
83       {
84          return phoneNumber;
85       } // fim do método getPhoneNumber
86    } // fim da classe Person
```

Figura 28.30 | A classe `Person` que representa uma entrada em um `AddressBook`.

Classe PersonQueries

A classe `PersonQueries` (Figura 28.31) gerencia a conexão de banco de dados do aplicativo de catálogo de endereços e cria as `PreparedStatements` que o aplicativo utiliza para interagir com o banco de dados. As linhas 18–20 declaram três variáveis `PreparedStatement`. O construtor (linhas 23–49) conecta-se ao banco de dados nas linhas 27–28.

As linhas 31–32 invocam o método `Connection` **`prepareStatement`** para criar a `PreparedStatement` chamada `selectAllPeople` que seleciona todas as linhas na tabela `Addresses`. As linhas 35–36 criam a `PreparedStatement` chamada `selectPeopleByLastName` com um parâmetro. Essa instrução seleciona todas as linhas na tabela `Addresses` que correspondem a determinado sobrenome. Observe o caractere `?` que é utilizado para especificar o parâmetro de sobrenome. As linhas 39–42 criam a `PreparedStatement` chamada `insertNewPerson` com quatro parâmetros que representam o nome, o sobrenome, o endereço de e-mail e o número de telefone de uma nova entrada. Novamente, observe os caracteres `?` utilizados para representar esses parâmetros.

```
1  // Figura 28.31: PersonQueries.java
2  // PreparedStatements utilizadas pelo aplicativo AddressBook.
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8  import java.util.List;
9  import java.util.ArrayList;
10
11 public class PersonQueries
12 {
13    private static final String URL = "jdbc:derby:AddressBook";
14    private static final String USERNAME = "deitel";
15    private static final String PASSWORD = "deitel";
16
17    private Connection connection = null; // gerencia a conexão
18    private PreparedStatement selectAllPeople = null;
19    private PreparedStatement selectPeopleByLastName = null;
20    private PreparedStatement insertNewPerson = null;
21
22    // construtor
23    public PersonQueries()
```

```
24       {
25          try
26          {
27             connection =
28                DriverManager.getConnection( URL, USERNAME, PASSWORD );
29
30             // cria a consulta que seleciona todas as entradas no AddressBook
31             selectAllPeople =
32                connection.prepareStatement( "SELECT * FROM Addresses" );
33
34             // cria a consulta que seleciona entradas com um sobrenome específico
35             selectPeopleByLastName = connection.prepareStatement(
36                "SELECT * FROM Addresses WHERE LastName = ?" );
37
38             // cria a inserção que adiciona uma nova entrada no banco de dados
39             insertNewPerson = connection.prepareStatement(
40                "INSERT INTO Addresses " +
41                "( FirstName, LastName, Email, PhoneNumber ) " +
42                "VALUES ( ?, ?, ?, ? )" );
43          } // fim do try
44          catch ( SQLException sqlException )
45          {
46             sqlException.printStackTrace();
47             System.exit( 1 );
48          } // fim do catch
49       } // fim do construtor
50
51       // seleciona todos os endereços no banco de dados
52       public List< Person > getAllPeople()
53       {
54          List< Person > results = null;
55          ResultSet resultSet = null;
56
57          try
58          {
59             // executeQuery retorna ResultSet que contém entradas correspondentes
60             resultSet = selectAllPeople.executeQuery();
61             results = new ArrayList< Person >();
62
63             while ( resultSet.next() )
64             {
65                results.add( new Person(
66                   resultSet.getInt( "addressID" ),
67                   resultSet.getString( "FirstName" ),
68                   resultSet.getString( "LastName" ),
69                   resultSet.getString( "Email" ),
70                   resultSet.getString( "PhoneNumber" ) ) );
71             } // fim do while
72          } // fim do try
73          catch ( SQLException sqlException )
74          {
75             sqlException.printStackTrace();
76          } // fim do catch
77          finally
78          {
79             try
80             {
81                resultSet.close();
82             } // fim do try
83             catch ( SQLException sqlException )
84             {
85                sqlException.printStackTrace();
86                close();
87             } // fim do catch
88          } // fim de finally
89
90          return results;
91       } // fim do método getAllPeople
92
```

```
93       // seleciona a pessoa pelo sobrenome
94       public List< Person > getPeopleByLastName( String name )
95       {
96          List< Person > results = null;
97          ResultSet resultSet = null;
98
99          try
100         {
101            selectPeopleByLastName.setString( 1, name ); // especifica o sobrenome
102
103            // executeQuery retorna ResultSet que contém entradas correspondentes
104            resultSet = selectPeopleByLastName.executeQuery();
105
106            results = new ArrayList< Person >();
107
108            while ( resultSet.next() )
109            {
110               results.add( new Person( resultSet.getInt( "addressID" ),
111                  resultSet.getString( "FirstName" ),
112                  resultSet.getString( "LastName" ),
113                  resultSet.getString( "Email" ),
114                  resultSet.getString( "PhoneNumber" ) ) );
115            } // fim do while
116         } // fim do try
117         catch ( SQLException sqlException )
118         {
119            sqlException.printStackTrace();
120         } // fim do catch
121         finally
122         {
123            try
124            {
125               resultSet.close();
126            } // fim do try
127            catch ( SQLException sqlException )
128            {
129               sqlException.printStackTrace();
130               close();
131            } // fim do catch
132         } // fim de finally
133
134         return results;
135      } // fim do método getPeopleByName
136
137      // adiciona uma entrada
138      public int addPerson(
139         String fname, String lname, String email, String num )
140      {
141         int result = 0;
142
143         // configura parâmetros e então executa insertNewPerson
144         try
145         {
146            insertNewPerson.setString( 1, fname );
147            insertNewPerson.setString( 2, lname );
148            insertNewPerson.setString( 3, email );
149            insertNewPerson.setString( 4, num );
150
151            // insere a nova entrada; retorna o número de linhas atualizadas
152            result = insertNewPerson.executeUpdate();
153         } // fim do try
154         catch ( SQLException sqlException )
155         {
156            sqlException.printStackTrace();
157            close();
158         } // fim do catch
159
160         return result;
161      } // fim do método addPerson
```

```
162
163       // fecha a conexão de banco de dados
164       public void close()
165       {
166          try
167          {
168             connection.close();
169          } // fim do try
170          catch ( SQLException sqlException )
171          {
172             sqlException.printStackTrace();
173          } // fim do catch
174       } // fim do método close
175    } // fim da classe PersonQueries
```

Figura 28.31 | `PreparedStatements` utilizadas pelo aplicativo de catálogo de endereços.

O método `getAllPeople` (linhas 52–91) executa o método `selectAllPeople` de `PreparedStatement` (linha 60) chamando o método **`executeQuery`**, que retorna um `ResultSet` contendo as linhas que correspondem à consulta (nesse caso, todas as linhas na tabela `Addresses`). As linhas 61–71 colocam os resultados da consulta em um ArrayList de objetos Person, que é retornado ao chamado na linha 90. O método `getPeopleByLastName` (linhas 94–135) utiliza o método `PreparedStatement` de `setString` para configurar o parâmetro para `selectPeopleByLastName` (linha 101). Então, a linha 104 executa a consulta e as linhas 106–115 colocam os resultados de consulta A linha 134 retorna a `ArrayList` ao chamador.

O método `addPerson` (linhas 138–161) utiliza o método `PreparedStatement` `setString` (linhas 146–149) para configurar os parâmetros para o `insertNewPerson` `PreparedStatement`. A linha 152 utiliza o método `PreparedStatement` **`executeUpdate`** para inserir o novo registro. Esse método retorna um número inteiro que indica o número de linhas que foram atualizadas (ou inseridas) no banco de dados. O método `close` (linhas 164–174) simplesmente fecha a conexão de banco de dados.

Classe `AddressBookDisplay`

O aplicativo `AddressBookDisplay` (Figura 28.32) utiliza um objeto `PersonQueries` para interagir com o banco de dados. A linha 59 cria o objeto `PersonQueries`. Quando o usuário pressionar o `JButton` **Browse All Entries**, o handler `browseButtonActionPerformed` (linhas 309–335) será chamado. A linha 313 chama o método `getAllPeople` no objeto `PersonQueries` para obter todas as entradas no banco de dados. O usuário então pode rolar pelas entradas utilizando os `JButtons` **Previous** e **Next**. Quando o usuário pressionar o `JButton` **Find**, o handler `queryButtonActionPerformed` (linhas 265–287) será chamado. As linhas 267–268 chamam o método `getPeopleBy-LastName` no objeto `PersonQueries` para obter as entradas no banco de dados que correspondem ao sobrenome especificado. Se houver várias entradas, o usuário pode então rolar por elas utilizando os `JButtons` **Previous** e **Next**.

```
1  // Figura 28.32: AddressBookDisplay.java
2  // Um catálogo de endereços simples
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.awt.event.WindowAdapter;
6  import java.awt.event.WindowEvent;
7  import java.awt.FlowLayout;
8  import java.awt.GridLayout;
9  import java.util.List;
10 import javax.swing.JButton;
11 import javax.swing.Box;
12 import javax.swing.JFrame;
13 import javax.swing.JLabel;
14 import javax.swing.JPanel;
15 import javax.swing.JTextField;
16 import javax.swing.WindowConstants;
17 import javax.swing.BoxLayout;
18 import javax.swing.BorderFactory;
19 import javax.swing.JOptionPane;
20
21 public class AddressBookDisplay extends JFrame
22 {
23    private Person currentEntry;
24    private PersonQueries personQueries;
25    private List< Person > results;
26    private int numberOfEntries = 0;
27    private int currentEntryIndex;
```

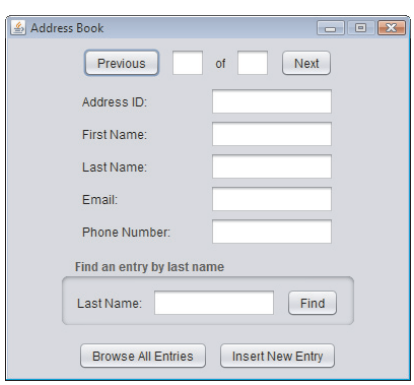
```
28
29       private JButton browseButton;
30       private JLabel emailLabel;
31       private JTextField emailTextField;
32       private JLabel firstNameLabel;
33       private JTextField firstNameTextField;
34       private JLabel idLabel;
35       private JTextField idTextField;
36       private JTextField indexTextField;
37       private JLabel lastNameLabel;
38       private JTextField lastNameTextField;
39       private JTextField maxTextField;
40       private JButton nextButton;
41       private JLabel ofLabel;
42       private JLabel phoneLabel;
43       private JTextField phoneTextField;
44       private JButton previousButton;
45       private JButton queryButton;
46       private JLabel queryLabel;
47       private JPanel queryPanel;
48       private JPanel navigatePanel;
49       private JPanel displayPanel;
50       private JTextField queryTextField;
51       private JButton insertButton;
52
53       // construtor sem argumento
54       public AddressBookDisplay()
55       {
56          super( "Address Book" );
57
58          // estabelece a conexão de banco de dados e configura PreparedStatements
59          personQueries = new PersonQueries();
60
61          // cria a GUI
62          navigatePanel = new JPanel();
63          previousButton = new JButton();
64          indexTextField = new JTextField( 2 );
65          ofLabel = new JLabel();
66          maxTextField = new JTextField( 2 );
67          nextButton = new JButton();
68          displayPanel = new JPanel();
69          idLabel = new JLabel();
70          idTextField = new JTextField( 10 );
71          firstNameLabel = new JLabel();
72          firstNameTextField = new JTextField( 10 );
73          lastNameLabel = new JLabel();
74          lastNameTextField = new JTextField( 10 );
75          emailLabel = new JLabel();
76          emailTextField = new JTextField( 10 );
77          phoneLabel = new JLabel();
78          phoneTextField = new JTextField( 10 );
79          queryPanel = new JPanel();
80          queryLabel = new JLabel();
81          queryTextField = new JTextField( 10 );
82          queryButton = new JButton();
83          browseButton = new JButton();
84          insertButton = new JButton();
85
86          setLayout( new FlowLayout( FlowLayout.CENTER, 10, 10 ) );
87          setSize( 400, 300 );
88          setResizable( false );
89
90          navigatePanel.setLayout(
91             new BoxLayout( navigatePanel, BoxLayout.X_AXIS ) );
92
93          previousButton.setText( "Previous" );
94          previousButton.setEnabled( false );
95          previousButton.addActionListener(
96             new ActionListener()
97             {
98                public void actionPerformed( ActionEvent evt )
```

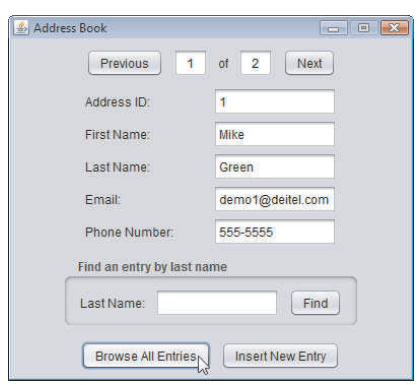
```
            {
               previousButtonActionPerformed( evt );
            } // fim do método actionPerformed
         } // fim da classe interna anônima
      ); // fim da chamada para addActionListener

      navigatePanel.add( previousButton );
      navigatePanel.add( Box.createHorizontalStrut( 10 ) );

      indexTextField.setHorizontalAlignment(
         JTextField.CENTER );
      indexTextField.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
               indexTextFieldActionPerformed( evt );
            } // fim do método actionPerformed
         } // fim da classe interna anônima
      ); // fim da chamada para addActionListener

      navigatePanel.add( indexTextField );
      navigatePanel.add( Box.createHorizontalStrut( 10 ) );

      ofLabel.setText( "of" );
      navigatePanel.add( ofLabel );
      navigatePanel.add( Box.createHorizontalStrut( 10 ) );

      maxTextField.setHorizontalAlignment(
         JTextField.CENTER );
      maxTextField.setEditable( false );
      navigatePanel.add( maxTextField );
      navigatePanel.add( Box.createHorizontalStrut( 10 ) );

      nextButton.setText( "Next" );
      nextButton.setEnabled( false );
      nextButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent evt )
            {
               nextButtonActionPerformed( evt );
            } // fim do método actionPerformed
         } // fim da classe interna anônima
      ); // fim da chamada para addActionListener

      navigatePanel.add( nextButton );
      add( navigatePanel );

      displayPanel.setLayout( new GridLayout( 5, 2, 4, 4 ) );

      idLabel.setText( "Address ID:" );
      displayPanel.add( idLabel );

      idTextField.setEditable( false );
      displayPanel.add( idTextField );

      firstNameLabel.setText( "First Name:" );
      displayPanel.add( firstNameLabel );
      displayPanel.add( firstNameTextField );

      lastNameLabel.setText( "Last Name:" );
      displayPanel.add( lastNameLabel );
      displayPanel.add( lastNameTextField );

      emailLabel.setText( "Email:" );
      displayPanel.add( emailLabel );
      displayPanel.add( emailTextField );

```

```
168          phoneLabel.setText( "Phone Number:" );
169          displayPanel.add( phoneLabel );
170          displayPanel.add( phoneTextField );
171          add( displayPanel );
172
173          queryPanel.setLayout(
174             new BoxLayout( queryPanel, BoxLayout.X_AXIS) );
175
176          queryPanel.setBorder( BorderFactory.createTitledBorder(
177             "Find an entry by last name" ) );
178          queryLabel.setText( "Last Name:" );
179          queryPanel.add( Box.createHorizontalStrut( 5 ) );
180          queryPanel.add( queryLabel );
181          queryPanel.add( Box.createHorizontalStrut( 10 ) );
182          queryPanel.add( queryTextField );
183          queryPanel.add( Box.createHorizontalStrut( 10 ) );
184
185          queryButton.setText( "Find" );
186          queryButton.addActionListener(
187             new ActionListener()
188             {
189                public void actionPerformed( ActionEvent evt )
190                {
191                   queryButtonActionPerformed( evt );
192                } // fim do método actionPerformed
193             } // fim da classe interna anônima
194          ); // fim da chamada para addActionListener
195
196          queryPanel.add( queryButton );
197          queryPanel.add( Box.createHorizontalStrut( 5 ) );
198          add( queryPanel );
199
200          browseButton.setText( "Browse All Entries" );
201          browseButton.addActionListener(
202             new ActionListener()
203             {
204                public void actionPerformed( ActionEvent evt )
205                {
206                   browseButtonActionPerformed( evt );
207                } // fim do método actionPerformed
208             } // fim da classe interna anônima
209          ); // fim da chamada para addActionListener
210
211          add( browseButton );
212
213          insertButton.setText( "Insert New Entry" );
214          insertButton.addActionListener(
215             new ActionListener()
216             {
217                public void actionPerformed( ActionEvent evt )
218                {
219                   insertButtonActionPerformed( evt );
220                } // fim do método actionPerformed
221             } // fim da classe interna anônima
222          ); // fim da chamada para addActionListener
223
224          add( insertButton );
225
226          addWindowListener(
227             new WindowAdapter()
228             {
229                public void windowClosing( WindowEvent evt )
230                {
231                   personQueries.close(); // fecha a conexão
232                   System.exit( 0 );
233                } // fim do método windowClosing
234             } // fim da classe interna anônima
235          ); // fim da chamada para addWindowListener
236
```

```
237          setVisible( true );
238       } // fim do construtor sem argumentos
239
240       // trata a chamada quando previousButton é clicado
241       private void previousButtonActionPerformed( ActionEvent evt )
242       {
243          currentEntryIndex--;
244
245          if ( currentEntryIndex < 0 )
246             currentEntryIndex = numberOfEntries - 1;
247
248          indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
249          indexTextFieldActionPerformed( evt );
250       } // fim do método previousButtonActionPerformed
251
252       // trata a chamada quando nextButton é clicado
253       private void nextButtonActionPerformed( ActionEvent evt )
254       {
255          currentEntryIndex++;
256
257          if ( currentEntryIndex >= numberOfEntries )
258             currentEntryIndex = 0;
259
260          indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
261          indexTextFieldActionPerformed( evt );
262       } // fim do método nextButtonActionPerformed
263
264       // trata a chamada quando queryButton é clicado
265       private void queryButtonActionPerformed( ActionEvent evt )
266       {
267          results =
268             personQueries.getPeopleByLastName( queryTextField.getText() );
269          numberOfEntries = results.size();
270
271          if ( numberOfEntries != 0 )
272          {
273             currentEntryIndex = 0;
274             currentEntry = results.get( currentEntryIndex );
275             idTextField.setText( "" + currentEntry.getAddressID() );
276             firstNameTextField.setText( currentEntry.getFirstName() );
277             lastNameTextField.setText( currentEntry.getLastName() );
278             emailTextField.setText( currentEntry.getEmail() );
279             phoneTextField.setText( currentEntry.getPhoneNumber() );
280             maxTextField.setText( "" + numberOfEntries );
281             indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
282             nextButton.setEnabled( true );
283             previousButton.setEnabled( true );
284          } // fim do if
285          else
286             browseButtonActionPerformed( evt );
287       } // fim do método queryButtonActionPerformed
288
289       // trata a chamada quando um novo valor é inserido em indexTextField
290       private void indexTextFieldActionPerformed( ActionEvent evt )
291       {
292          currentEntryIndex =
293             ( Integer.parseInt( indexTextField.getText() ) - 1 );
294
295          if ( numberOfEntries != 0 && currentEntryIndex < numberOfEntries )
296          {
297             currentEntry = results.get( currentEntryIndex );
298             idTextField.setText("" + currentEntry.getAddressID() );
299             firstNameTextField.setText( currentEntry.getFirstName() );
300             lastNameTextField.setText( currentEntry.getLastName() );
301             emailTextField.setText( currentEntry.getEmail() );
302             phoneTextField.setText( currentEntry.getPhoneNumber() );
303             maxTextField.setText( "" + numberOfEntries );
304             indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
305          } // fim do if
```

```
306       } // fim do método indexTextFieldActionPerformed
307
308       // trata a chamada quando browseButton é clicado
309       private void browseButtonActionPerformed( ActionEvent evt )
310       {
311          try
312          {
313             results = personQueries.getAllPeople();
314             numberOfEntries = results.size();
315
316             if ( numberOfEntries != 0 )
317             {
318                currentEntryIndex = 0;
319                currentEntry = results.get( currentEntryIndex );
320                idTextField.setText( "" + currentEntry.getAddressID() );
321                firstNameTextField.setText( currentEntry.getFirstName() );
322                lastNameTextField.setText( currentEntry.getLastName() );
323                emailTextField.setText( currentEntry.getEmail() );
324                phoneTextField.setText( currentEntry.getPhoneNumber() );
325                maxTextField.setText( "" + numberOfEntries );
326                indexTextField.setText( "" + ( currentEntryIndex + 1 ) );
327                nextButton.setEnabled( true );
328                previousButton.setEnabled( true );
329             } // fim do if
330          } // fim do try
331          catch ( Exception e )
332          {
333             e.printStackTrace();
334          } // fim do catch
335       } // fim do método browseButtonActionPerformed
336
337       // trata a chamada quando insertButton é clicado
338       private void insertButtonActionPerformed( ActionEvent evt )
339       {
340          int result = personQueries.addPerson( firstNameTextField.getText(),
341             lastNameTextField.getText(), emailTextField.getText(),
342             phoneTextField.getText() );
343
344          if ( result == 1 )
345             JOptionPane.showMessageDialog( this, "Person added!",
346                "Person added", JOptionPane.PLAIN_MESSAGE );
347          else
348             JOptionPane.showMessageDialog( this, "Person not added!",
349                "Error", JOptionPane.PLAIN_MESSAGE );
350
351          browseButtonActionPerformed( evt );
352       } // fim do método insertButtonActionPerformed
353
354       // método main
355       public static void main( String args[] )
356       {
357          new AddressBookDisplay();
358       } // fim do método main
359    } // fim da classe AddressBookDisplay
```





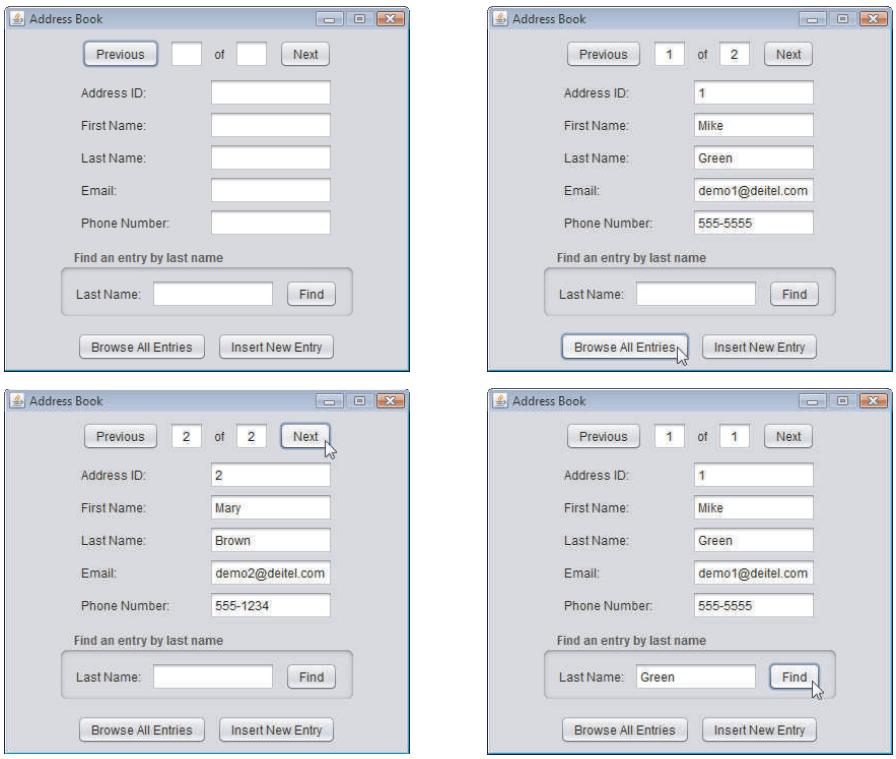


Figura 28.32 | Um catálogo de endereços simples.

Para adicionar uma nova entrada no banco de dados `AddressBook`, o usuário pode inserir o nome, o sobrenome, o e-mail e o número de telefone (o `AddressID` se autoincrementará) nos `JTextFields` e pressionar o `JButton` **Insert New Entry**. Quando o usuário pressiona **Insert New Entry**, o handler `insertButtonActionPerformed` (linhas 338–352) é chamado. As linhas 340–342 chamam o método `addPerson` no objeto `PersonQueries` para adicionar uma nova entrada ao banco de dados. A linha 351 chama `browseButtonActionPerformed` para obter o conjunto atualizado de pessoas no catálogo de endereços e atualizar a GUI apropriadamente.

O usuário então pode examinar as diferentes entradas pressionando o `JButton` **Previous** ou o `JButton` **Next**, que resulta em chamadas aos métodos `previousButtonActionPerformed` (linhas 241–250) ou `nextButtonActionPerformed` (linhas 253–262), respectivamente. Alternativamente, o usuário pode inserir um número no `indexTextField` e pressionar *Enter* para examinar determinada entrada. Isso resulta em uma chamada ao método `indexTextFieldActionPerformed` (linhas 290–306) para exibir o registro especificado.

28.12 Procedures armazenadas

Muitos sistemas de gerenciamento de bancos de dados podem armazenar instruções de SQL individuais ou conjuntos de instruções de SQL em um banco de dados, de modo que os programas que acessam esse banco de dados possam invocá-las. Essas coleções de SQL nomeadas são chamadas de **procedures armazenadas**. O JDBC permite que os programas invoquem procedures armazenadas utilizando objetos que implementam a interface `Callable-Statement`. `Callable-Statements` podem receber argumentos especificados com os métodos herdados da interface `PreparedStatement`. Além disso, `CallableStatements` podem especificar **parâmetros de saída** nos quais uma procedure armazenada pode colocar valores de retorno. A interface `CallableStatement` inclui métodos para especificar quais parâmetros em uma procedure armazenada são parâmetros de saída. A interface também inclui métodos para obter os valores de parâmetros de saída retornados de uma procedure armazenada. Para aprender mais sobre `CallableStatements`, visite

```
java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/
callablestatement.html#999652
```

Dica de portabilidade 28.6

Embora a sintaxe para criar procedures armazenadas seja diferente entre sistemas de gerenciamento de bancos de dados, a interface `CallableStatement` fornece uma interface uniforme para especificar os parâmetros de entrada e saída para procedures armazenadas e para invocar procedures armazenadas.

Dica de portabilidade 28.7

De acordo com a documentação da Java API para interface `CallableStatement`*, para a máxima portabilidade entre sistemas de banco de dados, os programas devem processar as contagens de atualização ou os* `ResultSet`*s retornados de um* `CallableStatement` *antes de obter os valores de qualquer parâmetro de saída.*

28.13 Processamento de transações

Muitos aplicativos do banco de dados impõem garantias de que uma série de inserções, atualizações e exclusões no banco de dados executem corretamente antes de os aplicativos continuarem o processamento da próxima operação de banco de dados. Por exemplo, quando você transfere dinheiro eletronicamente entre contas bancárias, vários fatores determinam se a transação é bem-sucedida. Você começa especificando a conta de origem e a quantia que deseja transferir dessa conta para uma conta de destino. Depois, especifica a conta de destino. O banco verifica a conta de origem para determinar se os fundos são suficientes para completar a transferência. Nesse caso, o banco retira a quantia especificada e, se tudo correr bem o deposita na conta de destino para concluir a transferência. O que acontece se a transferência falha depois de o banco retirar o dinheiro da conta de origem? Em um sistema bancário real, o banco deposita novamente o dinheiro na conta de origem. Como você se sentiria se o dinheiro fosse retirado da sua conta de origem e o banco *não* o depositasse na conta de destino?

O **processamento de transação** permite que um programa que interage com um banco de dados trate uma operação de banco de dados (ou conjunto de operações) como uma operação única. Essa operação também é conhecida como **operação atômica** ou **transação**. No fim de uma transação, pode-se tomar a decisão de **confirmar a transação** ou **reverter a transação**. Confirmar a transação finaliza a(s) operação(ões) do banco de dados; todas as inserções, atualizações e exclusões realizadas como parte da transação não podem ser revertidas sem realizar uma nova operação de banco de dados. A reversão da transação deixa o banco de dados no seu estado anterior à operação de banco de dados. Isso é útil quando parte de uma transação não consegue ser completada corretamente. Na nossa discussão sobre transferência da conta bancária, a transação seria revertida se o depósito não pudesse ser feito na conta de destino.

O Java fornece processamento de transações por meio de métodos da interface `Connection`. O método **`setAutoCommit`** especifica se cada instrução SQL é confirmada depois de ser completada (um argumento `true`) ou se várias instruções SQL devem ser agrupadas como uma transação (um argumento `false`). Se o argumento para `setAutoCommit` for `false`, o programa deve seguir a última instrução SQL na transação com uma chamada ao método `Connection` **`commit`** (para confirmar as modificações no banco de dados) ou método `Connection` **`rollback`** (para retornar o banco de dados ao estado anterior da transação). A interface `Connection` também fornece o método **`getAutoCommit`** para determinar o estado autocommit para `Connection`.

28.14 Conclusão

Neste capítulo, você aprendeu os conceitos básicos de banco de dados, a interagir com dados em um banco de dados utilizando SQL e a utilizar JDBC para permitir que os aplicativos Java manipulem dados de banco de dados. Aprendeu sobre os comandos SQL `SELECT`, `INSERT`, `UPDATE` e `DELETE`, bem como cláusulas, como `WHERE`, `ORDER BY` e `INNER JOIN`. Conheceu os passos explícitos para obter uma `Connection` com o banco de dados, para criar uma `Statement` para interagir com os dados do banco de dados, executar a instrução e processar os resultados. Então utilizou um `RowSet` para simplificar o processo de conexão com um banco de dados e criar instruções. Utilizou `PreparedStatements` para criar instruções SQL precompiladas. Também aprendeu a criar e configurar bancos de dados tanto no Java DB como no MySQL utilizando scripts SQL predefinidos. Também fornecemos visões gerais sobre as `CallableStatements` e o processamento de transação. No próximo capítulo, você aprenderá sobre o desenvolvimento de aplicativos Web com o JavaServer Faces.

28.15 Recursos da Web

`java.sun.com/javase/technologies/database/index.jsp`
Sun Microsystems, Java SE database technologies home page.

`java.sun.com/docs/books/tutorial/jdbc/index.html`
Trilha JDBC do *The Java Tutorial*.

`developers.sun.com/product/jdbc/drivers`
O sistema de pesquisa da Sun Microsystems para localizar drivers JDBC.

`www.sql.org`
Esse portal de SQL fornece links para muitos recursos, incluindo sintaxe de SQL, dicas, tutoriais, livros, revistas, grupos de discussão, empresas com serviços de SQL, consultores de SQL e softwares gratuitos.

`www.datadirect.com/developer/jdbc/topics/perfoptjdbc/index.ssp`
Artigo que discute como projetar um bom aplicativo JDBC.

`java.sun.com/javase/6/docs/technotes/guides/jdbc/index.html`
A documentação do JDBC API da Sun Microsystems.

`www.mysql.com`
Esse site é a home page do banco de dados MySQL. Você pode fazer download das últimas versões de MySQL e MySQL Connector/J e acessar sua documentação on-line.

`dev.mysql.com/doc/refman/5.0/en/index.html`
Manual de referência de MySQL.

`java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html`
Visão geral da interface `RowSet` e suas subinterfaces. Esse site também discute as implementações de referência dessas interfaces da Sun e sua utilização.

`java.sun.com/developer/Books/JDBCTutorial/chapter5.html`
O Capítulo 5 (Tutorial `RowSet`) do livro *The JDBC 2.0 API Tutorial and Reference, Second Edition*.

Resumo

Seção 28.1 Introdução

- Um banco de dados é uma coleção integrada de dados. Um sistema de gerenciamento de bancos de dados (*Database Management System* — DBMS) fornece mecanismos para armazenar, organizar, recuperar e modificar dados para muitos usuários.
- Os sistemas de gerenciamento de bancos de dados mais populares de hoje são sistemas de bancos de dados relacionais.
- SQL é a linguagem padrão internacional utilizada para consultar e manipular dados relacionais.
- Os programas conectam-se aos bancos de dados relacionais e interagem com estes por meio de uma interface — o software que facilita comunicações entre um sistema de gerenciamento de bancos de dados e um programa.
- Um driver JDBC permite que os aplicativos Java se conectem a um banco de dados em um DBMS particular e permite aos programadores recuperar e manipular dados de banco de dados.

Seção 28.2 Bancos de dados relacionais

- Um banco de dados relacional armazena dados em tabelas. As tabelas são compostas de linhas, e as linhas são compostas de colunas nas quais os valores são armazenados.
- Uma chave primária fornece um valor único que não pode ser duplicado em outras linhas da mesma tabela.
- Cada coluna de uma tabela representa um atributo diferente.
- A chave primária pode ser composta de mais de uma coluna.
- Cada coluna em uma chave primária deve ter um valor, e o valor da chave primária deve ser único. Isso é conhecido como Regra de Integridade de Entidade.
- Um relacionamento de um para muitos entre tabelas indica que uma linha em uma tabela pode ter muitas linhas relacionadas em uma tabela separada.
- Uma chave estrangeira é uma coluna em uma tabela que corresponde à coluna de chave primária em outra tabela. Isso é conhecido como Regra de Integridade de Entidade.
- As chaves estrangeiras permitem que as informações de múltiplas tabelas sejam unidas. Há um relacionamento um para muitos entre uma chave primária e sua chave estrangeira correspondente.

Seção 28.4.1 Consulta `SELECT` básica

- A forma básica de uma consulta é

  ```
  SELECT * FROM nomeDaTabela
  ```

  onde o asterisco (`*`) indica que todas as colunas de *nomeDaTabela* devem ser selecionadas e *nomeDaTabela* especifica a tabela no banco de dados a partir da qual as linhas serão recuperadas.
- Para recuperar colunas específicas, substitua o `*` por uma lista separada por vírgulas de nomes de coluna.

Seção 28.4.2 Cláusula `WHERE`

- A cláusula `WHERE` opcional em uma consulta especifica os critérios de seleção da consulta. A forma básica de uma consulta com critérios de seleção é

  ```
  SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela WHERE critérios
  ```
- A cláusula `WHERE` pode conter os operadores <, >, <=, >=, =, <> e `LIKE`. O operador `LIKE` é utilizado para correspondência de string de padrão com os curingas porcentagem (`%`) e sublinhado (`_`).

- Um caractere de porcentagem (%) em um padrão indica que uma string que corresponde ao padrão tem zero ou mais caracteres na posição do caractere de porcentagem no padrão.
- Um sublinhado (_) na string de padrão indica um único caractere nessa posição do padrão.

Seção 28.4.3 Cláusula ORDER BY

- O resultado de uma consulta pode ser classificado com a cláusula ORDER BY. A forma mais simples de uma cláusula ORDER BY é

```
SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela ORDER BY coluna ASC
SELECT nomeDaColuna1, nomeDaColuna2, ... FROM nomeDaTabela ORDER BY coluna DESC
```

  onde ASC especifica a ordem crescente, DESC especifica a ordem decrescente e *coluna* especifica a coluna em que a classificação é baseada. A ordem de classificação padrão é crescente, então ASC é opcional.
- Múltiplas colunas podem ser utilizadas para fins de ordenação com uma cláusula ORDER BY da forma

```
ORDER BY coluna1 ordemDeClassificação, coluna2 ordemDeClassificação, ...
```

- As cláusulas WHERE e ORDER BY podem ser combinadas em uma consulta. Se utilizada, ORDER BY deve ser a última cláusula na consulta.

25.4.4 Mesclando dados a partir de múltiplas tabelas: INNER JOIN

- Uma INNER JOIN mescla linhas de duas tabelas correspondendo valores em colunas que são comuns às tabelas. A forma básica do operador INNER JOIN é:

```
SELECT nomeDaColuna1, nomeDaColuna2, ...
FROM tabela1
INNER JOIN tabela2
   ON tabela1.nomeDaColuna = tabela2.nomeDaColuna
```

  A cláusula ON especifica as colunas de cada tabela que são comparadas para determinar as linhas que são unidas. Se uma instrução de SQL utiliza colunas com o mesmo nome de múltiplas tabelas, os nomes de coluna devem ser completamente qualificados prefixando-os com seus nomes de tabela e um ponto (.).

Seção 28.4.5 Instrução INSERT

- Uma instrução INSERT insere uma nova linha em uma tabela. A forma básica dessa instrução é

```
INSERT INTO nomeDaTabela ( nomeDaColuna1, nomeDaColuna2, ..., nomeDaColunaN )
   VALUES ( valor1, valor2, ..., valorN )
```

  onde *nomeDaTabela* é a tabela na qual inserir a linha. O *nomeDaTabela* é seguido por uma lista separada por vírgulas de nomes de coluna entre parênteses. A lista de nomes de coluna é seguida pela palavra-chave de SQL VALUES e uma lista separada por vírgulas de valores entre parênteses.
- SQL utiliza aspa simples (') para delimitar strings. Para especificar uma string que contém uma aspa simples em SQL, escape a aspa simples com outra aspa simples (isto é, '').

Seção 28.4.6 Instrução UPDATE

- Uma instrução UPDATE modifica os dados em uma tabela. A forma básica de uma instrução UPDATE é

```
UPDATE nomeDaTabela
   SET nomeDaColuna1 = valor1, nomeDaColuna2 = valor2, ..., nomeDaColunaN = valorN
   WHERE critérios
```

  onde *nomeDaTabela* é a tabela em que atualizar os dados. *O nomeDaTabela* é seguido pela palavra-chave SET e uma lista separada por vírgulas de pares *nomeDaColuna* = *valor*. Os *critérios* da cláusula WHERE opcional determinam que linhas atualizar.

Seção 28.4.7 Instrução DELETE

- Uma instrução DELETE remove as linhas de uma tabela. A forma mais simples de uma instrução DELETE é

```
DELETE FROM nomeDaTabela WHERE critérios
```

  onde *nomeDaTabela* é a tabela a partir da qual excluir uma linha (ou linhas). Os *critérios* WHERE opcionais determinam que linhas excluir. Se essa cláusula for omitida, todas as linhas da tabela serão excluídas.

25.8.1 Consultando e conectando-se a um banco de dados

- O pacote java.sql contém as classes e interfaces para acessar bancos de dados relacionais em Java.
- Um objeto que implementa a interface Connection gerencia a conexão entre um programa Java e um banco de dados. Os objetos Connection permitem aos programas criar instruções de SQL que acessam dados.
- O método DriverManager getConnection tenta conectar-se a um banco de dados em um URL que especifica o protocolo para a comunicação, o subprotocolo da comunicação e o nome de banco de dados.
- O método Connection createStatement cria um objeto Statement, que pode ser utilizado para enviar instruções SQL para o banco de dados.
- O método Statement executeQuery executa uma consulta e retorna um objeto que implementa a interface ResultSet que contém o resultado de consulta. Os métodos ResultSet permitem a um programa manipular resultados de consulta.

- Um objeto `ResultSetMetaData` descreve o conteúdo de um `ResultSet`. Os programas podem utilizar metadados programaticamente para obter informações sobre os nomes de coluna e tipos de `ResultSet`.
- O método `ResultSetMetaData getColumnCount` recupera o número de colunas `ResultSet`.
- O método `ResultSet next` posiciona o cursor `ResultSet` na próxima linha e retorna `true` se a linha existir; caso contrário, retorna `false`. Esse método deve ser chamado para começar o processamento de um `ResultSet` porque o cursor está inicialmente posicionado antes da primeira linha.
- É possível extrair cada coluna `ResultSet` como um tipo Java específico. O método `ResultSetMetaData getColumnType` retorna uma constante `Types` (pacote `java.sql`) para indicar o tipo da coluna.
- Os métodos `ResultSet` *get* em geral recebem como um argumento um número de coluna (como um `int`) ou um nome de coluna (como uma `String`) indicando que valor da coluna obter.
- Os números de linha e coluna de `ResultSet` iniciam em 1.
- Todo objeto `Statement` pode abrir apenas um objeto `ResultSet` por vez. Quando um `Statement` retorna um novo `ResultSet`, `Statement` fecha o `ResultSet` anterior.
- O método `Connection createStatement` tem uma versão sobrecarregada que recebe o tipo de resultado e a concorrência de resultado. O tipo de resultado especifica se o cursor de `ResultSet` é capaz de rolar em ambas as direções ou apenas avançar e se o `ResultSet` é sensível ou não às alterações. A concorrência de resultado especifica se o `ResultSet` pode ser atualizado com os métodos de atualização do `ResultSet`.
- Alguns drivers JDBC não suportam `ResultSet`s roláveis ou atualizáveis.

Seção 28.8.2 Consultando o banco de dados books

- O método `TableModel getColumnClass` retorna um objeto `Class` que representa a superclasse de todos os objetos em determinada coluna. A `JTable` utiliza essas informações para configurar o renderizador de célula e editor de célula padrão para essa coluna na `JTable`.
- O método `ResultSetMetaData getColumnClassName` obtém o nome de classe totalmente qualificado de uma coluna.
- O método `TableModel getColumnCount` retorna o número de colunas no `ResultSet` subjacente.
- O método `TableModel getNomeDaColuna` retorna o nome de coluna no `ResultSet` subjacente.
- O método `ResultSetMetaData getNomeDaColuna` obtém o nome de coluna do `ResultSet`.
- O método `TableModel getRowCount` retorna o número de linhas no modelo `ResultSet`.
- O método `TableModel getValueAt` retorna o `Object` em uma linha e coluna particulares do `ResultSet` subjacente do modelo.
- O método `ResultSet absolute` posiciona o cursor `ResultSet` em uma linha específica.
- O método `AbstractTableModel fireTableStructureChanged` notifica qualquer `JTable` utilizando um objeto `TableModel` particular como seu modelo que os dados no modelo alterou.

Seção 28.9 Interface RowSet

- A interface `RowSet` configura a conexão de banco de dados e executa a consulta automaticamente.
- Um `RowSet` conectado permanece conectado ao banco de dados enquanto o objeto estiver em uso. Um `RowSet` desconectado conecta, executa uma consulta e, então, fecha a conexão.
- `JdbcRowSet`, um `RowSet` conectado, empacota um objeto `ResultSet` e permite que você role e atualize as linhas. Ao contrário de um objeto `ResultSet`, um objeto `JdbcRowSet` é rolável e atualizável por padrão.
- `CachedRowSet`, um `RowSet` desconectado, armazena em cache dados de um `ResultSet` na memória. Um `CachedRowSet` é rolável e atualizável. Um `CachedRowSet` também é serializável.

Seção 28.10 Java DB/Apache Derby

- No JDK 6, a Sun Microsystems distribui o código-fonte aberto, o banco de dados "puro Java" chamado Java DB (a versão com a marca Sun do Apache Derby) juntamente com o JDK.
- O Java DB tem uma versão incorporada e uma versão de rede.

Seção 28.11 PreparedStatements

- `PreparedStatement`s são compiladas, portanto, executam mais eficientemente do que as `Statement`s.
- `PreparedStatement`s podem ter parâmetros, portanto, a mesma consulta pode executar com diferentes argumentos.
- Um parâmetro é especificado com um sinal de interrogação (`?`) na instrução SQL. Antes de executar uma `PreparedStatement`, você deve utilizar os métodos *set* da `PreparedStatement` para especificar os argumentos.
- O primeiro argumento do método `PreparedStatement setString` representa o número do parâmetro sendo configurado e o segundo argumento é o valor desse parâmetro.
- Os números de parâmetro são contados a partir de 1, iniciando com o primeiro ponto de interrogação (`?`).
- O método `setString` escapa automaticamente valores de parâmetro `String` conforme necessário.
- A interface `PreparedStatement` fornece métodos set para cada tipo SQL suportado.

- Uma coluna de identidade é a maneira padrão do SQL de representar uma coluna autoincrementada. A palavra-chave `IDENTITY` SQL marca uma coluna como uma coluna de identidade.

Seção 28.12 Procedures armazenadas

- O JDBC permite que programas invoquem procedimentos armazenados (ou *stored procedures*) para utilizar objetos `CallableStatement`.
- `CallableStatement` pode especificar parâmetros de entrada. `CallableStatement` pode especificar parâmetros de saída nos quais um procedimento armazenado pode colocar valores de retorno.

Seção 28.13 Processamento de transações

- O processamento de transação permite que um programa que interage com um banco de dados trate uma operação de banco de dados (ou conjunto de operações) como uma operação única — conhecida como operação atômica ou transação.
- No fim de uma transação, pode-se tomar uma decisão para confirmar ou reverter a transação.
- Confirmar uma transação finaliza a(s) operação(ões) de banco de dados — inserções, atualizações e exclusões não podem ser revertidas sem realizar uma nova operação de banco de dados.
- Reverter uma transação deixa o banco de dados no estado anterior à operação de banco de dados.
- O Java fornece processamento de transação via métodos de interface `Connection`.
- O método `setAutoCommit` especifica se cada instrução SQL confirma depois que ela for completada (um argumento `true`) ou se várias instruções SQL devem ser agrupadas como uma transação.
- Quando a confirmação automática (*autocommit*) estiver desativada, o programa deve seguir a última instrução SQL na transação com uma chamada para o método `Connection` `commit` (para confirmar as alterações no banco de dados) ou o método `Connection` `rollback` (para retornar o banco de dados ao seu estado anterior à transação).
- O método `getAutoCommit` determina o estado de confirmação automática para a `Connection`.

Terminologia

*, caractere curinga de SQL, 903
%, caractere curinga de SQL, 904
_, caractere curinga de SQL, 904
absolute, método de ResultSet, 919
AbstractTableModel, classe, 915
addTableModelListener, método de TableModel, 915
banco de dados, 899
banco de dados relacional, 900
CachedRowSet, interface, 924
CallableStatement, interface, 938
chave estrangeira, 901
chave primária, 900
close, método da interface JdbcRowSet, 926
coluna autoincrementada, 901
coluna em uma tabela de banco de dados, 900
com.sun.rowset, pacote, 925
commit, método da interface Connection, 939
concorrência de conjunto de resultados, 918
CONCUR_READ_ONLY, constante, 919
CONCUR_UPDATABLE, constante, 919
conectando tabelas de banco de dados, 901
confirmar uma transação, 939
Connection, interface, 913
consulta, 899
correspondência de padrão, 904
createStatement, método de Connection, 914
critérios de seleção, 904
DELETE, instrução SQL, 909
descoberta automática de driver (JDBC 4), 913
diagrama de relacionamento de entidades, 902
driver JDBC, 899
DriverManager, classe, 913
execute, método de JdbcRowSet, 925
executeQuery, método da interface PreparedStatement, 932
executeQuery, método da interface Statement, 914
executeUpdate, método da interface PreparedStatement, 932
fireTableStructureChanged, método de AbstractTableModel, 919
getAutoCommit, método da interface Connection, 939
getColumnClass, método de TableModel, 915
getColumnClassName, método de ResultSetMetaData, 919
getColumnCount, método de ResultSetMetaData, 914
getColumnCount, método de TableModel, 915
getColumnName, método de ResultSetMetaData, 919
getColumnName, método de TableModel, 915
getColumnType, método de ResultSetMetaData, 914
getConnection, método de DriverManager, 913
getInt, método de ResultSet, 914
getObject, método da interface ResultSet, 914
getRow, método da interface ResultSet, 919
getRowCount, método da interface TableModel, 915
getValueAt, método da interface TableModel, 915
IDENTITY, palavra-chave (SQL), 928
INNER JOIN, cláusula de SQL, 907
INSERT, instrução SQL, 908
interface, 914
Java Database Connectivity (JDBC), 899
Java DB, 926
javax.sql.rowset, pacote, 924
javax.swing.table, pacote, 923
JDBC API, 899
JdbcRowSet, interface, 924
JdbcRowSetImpl, classe, 925
JTable, classe, 915
last, método de ResultSet, 919
LIKE, operador (SQL), 904
linha em uma tabela de banco de dados, 900
metadados, 914
MySQL 5.0 Community Edition, 910
MySQL Connector/J, 910
next, método da interface ResultSet, 914
nome qualificado, 907
ON, cláusula, 907
operação atômica, 939
ORDER BY, cláusula de SQL, 905
parâmetro de saída para CallableStatement, 938
predicado, 904
PreparedStatement, interface, 927
prepareStatement, método da interface Connection, 929
procedure armazenada, 938
processamento de transação, 939
regexFilter, método da classe RowFilter, 924
Regra de Integridade de Entidade, 902
Regra de Integridade Referencial, 901
removeTableModelListener, método da interface TableModel, 915
ResultSet, interface, 914
ResultSetMetaData, interface, 914
reverter uma transação, 939

`rollback`, método da interface `Connection`, 939
`RowFilter`, classe, 924
`RowSet` conectado, 924
`RowSet` desconectado, 924
`RowSet`, interface, 924
`SELECT`, SQL, palavras-chave, 903
`SET`, cláusula de SQL, 909
`setAutoCommit`, método da interface `Connection`, 939
`setCommand`, método da interface `JdbcRowSet`, 925
`setPassword`, método da interface `JdbcRowSet`, 925
`setRowFilter`, método da classe `JTable`, 924
`setRowSorter`, método da classe `JTable`, 923
`setString`, método da interface `PreparedStatement`, 927
`setUrl`, método da interface `JdbcRowSet`, 925
`setUsername`, método da interface `JdbcRowSet`, 925
sistema de gerenciamento de bancos de dados relacional (RDBMS), 899
sistema de gerenciamento de bancos de dados (*Database Management System* — DBMS), 899
`SQLException`, classe, 913
`Statement`, interface, 914
Structured Query Language (SQL), 899
subprotocolo para comunicação, 913
tabela de um banco de dados, 900
`TableModel`, interface, 915
`TableRowSorter`, classe, 923
tipo de conjunto de resultados, 918
transação, 939
`TYPE_FORWARD_ONLY`, constante, 918
`TYPE_SCROLL_INSENSITIVE`, constante, 918
`TYPE_SCROLL_SENSITIVE`, constante, 918
`Types`, classe, 914
um para muitos, relacionamento, 903
`UPDATE`, instrução SQL, 908
`VALUES`, cláusula de SQL, 908
`WHERE`, cláusula de SQL, 904
`WindowAdapter`, classe, 924
`windowClosed`, método da interface `WindowListener`, 924
`WindowListener`, interface, 924

Exercício de autorrevisão

28.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) A linguagem padrão internacional de banco de dados é _____.
b) Uma tabela em um banco de dados consiste em _____ e _____ .
c) Os objetos de instrução retornam resultados de consulta de SQL como objetos _____.
d) O _____ identifica unicamente cada linha em uma tabela.
e) A palavra-chave de SQL _____ é seguida pelos critérios de seleção que especificam as linhas a selecionar em uma consulta.
f) As palavras-chave de SQL _____ especificam a ordem em que linhas são classificadas em uma consulta.
g) Mesclar linhas de múltiplas tabelas de banco de dados é chamado de fazer _____ das tabelas.
h) Um(a) _____ é uma coleção organizada de dados.
i) Um(a) _____ é um conjunto de colunas cujos valores correspondem aos valores de chave primária de outra tabela.
j) O método _____ _____ é utilizado para obter uma `Connection` com um banco de dados.
k) A interface _____ ajuda a gerenciar a conexão entre um programa Java e um banco de dados.
l) Um objeto _____ é utilizado para submeter uma consulta a um banco de dados.
m) Ao contrário de um objeto `ResultSet`, os objetos _____ e _____ são roláveis e atualizáveis por padrão.
n) _____, um `RowSet` desconectado, armazena os dados em cache de um `ResultSet` na memória.

Respostas do exercício de autorrevisão

28.1 a) SQL. b) linhas, colunas. c) `ResultSet`. d) chave primária. e) `WHERE`. f) `ORDER BY`. g) junção. h) banco de dados. i) chave estrangeira. j) `DriverManager`, `getConnection` k) `Connection`. l) `Statement`. m) `JdbcRowSet`, `CachedRowSet` n) `CachedRowSet`.

Exercícios

28.2 (***Aplicativo de consulta para o banco de dados books***) Utilizando as técnicas mostradas neste capítulo, defina um aplicativo de consulta completo para o banco de dados `books`. Forneça as seguintes consultas predefinidas:

a) Selecione todos os autores da tabela `Authors`.
b) Selecione um autor específico e liste todos os livros para esse autor. Inclua o título, ano e ISBN de cada livro. Ordene as informações alfabeticamente pelo sobrenome do autor e depois pelo nome.
c) Selecione um editor específico e liste todos os livros publicados por esse editor. Inclua o título, ano e ISBN. Ordene as informações alfabeticamente por título.
d) Forneça quaisquer outras consultas que você considerar apropriadas.

Exiba um `JComboBox` com nomes apropriados para cada consulta predefinida. Também permita que os usuários forneçam suas próprias consultas.

28.3 (***Aplicativo de manipulação de dados para o banco de dados books***) Defina um aplicativo de manipulação de dados para o banco de dados `books`. O usuário deve ser capaz de editar os dados existentes e adicionar novos dados ao banco de dados (obedecendo às restrições de integridade referenciais e de entidade). Permita ao usuário editar o banco de dados das seguintes maneiras:

a) Adicionar um novo autor.
b) Editar as informações existentes para um autor.
c) Adicione um novo título para um autor. (Lembre-se de que o livro deve ter uma entrada na tabela `AuthorISBN`).
d) Adicione uma nova entrada na tabela `AuthorISBN` para vincular autores com títulos.

28.4 (***Banco de dados de empregados***) Na Seção 10.7, introduzimos uma hierarquia de folhas de pagamento de empregados para calcular a folha de pagamento de cada empregado. Nesse exercício, fornecemos um banco de dados de empregados que corresponde à hierarquia de folhas de pagamento dos empregados. (Um script SQL para criar o banco de dados `employees` é fornecido com os exemplos deste capítulo.) Escreva um aplicativo que permita ao usuário:

a) Adicionar empregados à tabela `employee`.

b) Adicionar informações de folha de pagamento à tabela apropriada para cada novo empregado. Por exemplo, para um empregado assalariado adicione informações de folha de pagamento à tabela `salariedEmployees`.

A Figura 28.33 é o diagrama de relacionamento de entidade do banco de dados `employees`.

28.5 (***Aplicativo de consulta do banco de dados de empregados***) Modifique o Exercício 28.4 para fornecer uma `JComboBox` e uma `JTextArea` para permitir ao usuário realizar uma consulta que seja selecionada a partir da `JComboBox` ou definida na `JTextArea`. As consultas predefinidas de exemplo são:

a) Selecione todos os empregados que trabalham no Departamento de SALES.

b) Selecione os empregados por hora que trabalham mais de 30 horas.

c) Selecione todos os empregados comissionados em ordem decrescente da taxa de comissão.

28.6 (***Aplicativo de manipulação de dados do banco de dados employee***) Modifique o Exercício 28.5 para realizar as seguintes tarefas:

a) Aumente 10% do salário-base de todos os empregados comissionados com salário-base.

b) Se o aniversário do empregado cair no mês atual, adicione um bônus de US$ 100.

c) Para todos os empregados comissionados com vendas brutas acima de US$ 10.000 adicione um bônus de US$ 100.

28.7 (***Modificação do AddressBook: Atualize uma entrada existente***) Modifique o programa das figuras 28.30–28.32 para fornecer um `JButton` que permite ao usuário chamar um método denominado `updatePerson` na classe `PersonQueries` para atualizar a entrada atual no banco de dados `AddressBook`.

28.8 (***Modificação do AddressBook: Exclua uma entrada existente***) Modifique o programa do Exercício 28.7 para fornecer um `JButton` que permite ao usuário chamar um método denominado `deletePerson` na classe `PersonQueries` para excluir a entrada atual no banco de dados `AddressBook`.

28.9 (***Estudo de caso de ATM com um banco de dados***) Modifique o Estudo de Caso de ATM (capítulos 12–13) para utilizar um banco de dados de verdade para armazenar informações sobre contas. Fornecemos um script SQL para criar o `BankDatabase`, que tem uma única tabela consistindo em quatro colunas — `AccountNumber` (um `int`), `PIN` (um `int`), `AvailableBalance` (um `double`) e `TotalBalance` (um `double`).

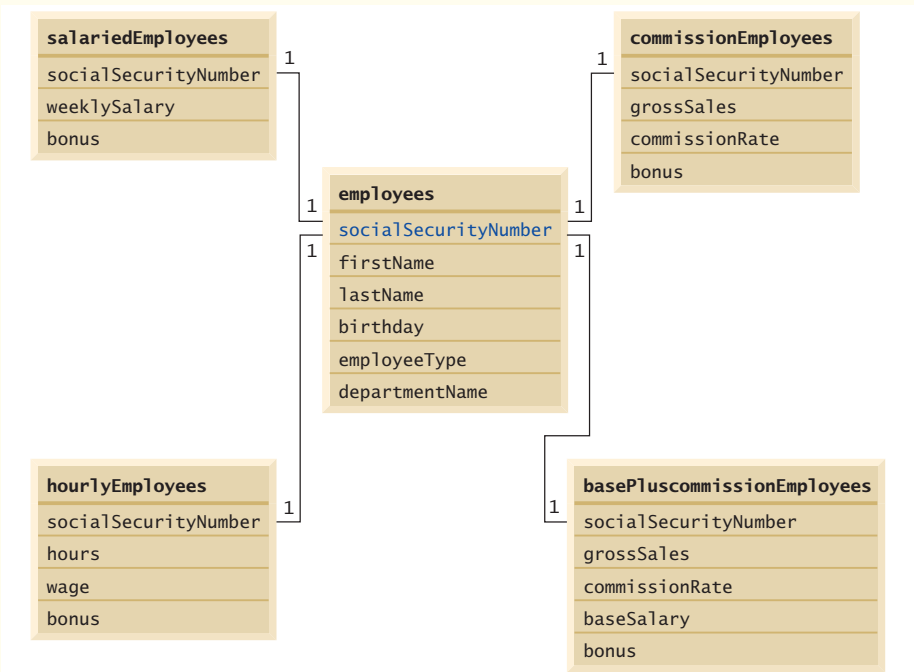


Figura 28.33 | Relações de tabela no banco de dados `employees`.

29

Se um homem qualquer descrever seu caso e colocar seu nome na primeira página, ele receberá uma resposta imediata. Se fizer com que eu precise virar a página, ele terá de esperar.
— **Lord Sandwich**

Regra um: nosso cliente sempre tem razão.
Regra dois: se você acha que nosso cliente está errado, veja a Regra 1.
— **Anônimo**

Uma questão justa deve ser seguida por uma ação em silêncio.
— **Dante Alighieri**

Você virá aqui e obterá livros que abrirão seus olhos, seus ouvidos e sua curiosidade, e farão uma revolução em sua mente.
— **Ralph Waldo Emerson**

Aplicativos Web JavaServer™ Faces

Objetivos

Neste capítulo, você aprenderá:

- O desenvolvimento de aplicativos Web utilizando as tecnologias Java e NetBeans.
- A criar JavaServer Pages com componentes JavaServer Faces.
- A criar aplicativos Web que consistem em múltiplas páginas.
- A validar a entrada de usuário em uma página Web.
- A manter informações de estado específicas ao usuário por todo um aplicativo Web com rastreamento de sessão e cookies.

Sumário

29.1 Introdução
29.2 Transações HTTP simples
29.3 Arquitetura de aplicativo multithread
29.4 Tecnologias Web Java
29.4.1 Servlets
29.4.2 JavaServer Pages
29.4.3 JavaServer Faces
29.4.4 Tecnologias Web no NetBeans
29.5 Criando e executando um aplicativo simples no NetBeans
29.5.1 Examinando um documento JSP
29.5.2 Examinando um arquivo Page Bean
29.5.3 Ciclo de vida de processamento de eventos
29.5.4 Criando um aplicativo Web no NetBeans
29.6 Componentes JSF
29.6.1 Componentes gráficos e de texto
29.6.2 Validação utilizando componentes validadores e validadores personalizados
29.7 Monitoramento de sessão
29.7.1 Cookies
29.7.2 Rastreamento de sessão com beans de sessão
29.8 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

29.1 Introdução

Neste capítulo, introduzimos o desenvolvimento de aplicativos Web com Java. Aplicativos baseados na Web criam conteúdo para clientes que utilizam navegadores Web. Esse conteúdo inclui a Extensible HyperText Markup Language (XHTML), criação de scripts do lado do cliente, imagens e dados binários. Se não estiver familiarizado com a XHTML, visite nosso XHTML Resource Center em `www.deitel.com/XHTML/` para introduções, tutoriais e outros recursos que irão ajudá-lo a aprender a XHTML. Para uma lista completa dos nossos Resource Centers, visite `www.deitel.com/ResourceCenters.html`.

Este capítulo começa com uma visão geral da arquitetura de aplicativos multicamadas e tecnologias Web baseadas em Java para implementar aplicativos multicamadas. Apresentamos então vários exemplos de aplicativos Web. O primeiro exemplo introduz o desenvolvimento Web com Java. No segundo exemplo, construímos um aplicativo Web que simplesmente mostra a aparência e funcionamento de vários componentes GUI de aplicativos Web. Em seguida, mostramos como utilizar componentes de validação e métodos de validação personalizados para assegurar que a entrada do usuário seja válida antes de ela ser submetida a processamento no servidor. O capítulo termina com dois exemplos de como manter informações específicas ao usuário em um aplicativo Web. O Capítulo 30 continua nossa discussão do desenvolvimento de aplicativos Web com Java com conceitos mais avançados.

Por todo este capítulo e o Capítulo 30, utilizamos o IDE do **NetBeans 6.5** e o servidor de aplicativo de código-fonte aberto **GlassFish v2 UR2**. Para implementar os exemplos apresentados neste capítulo, você deve instalar esses softwares. NetBeans está disponível em

```
www.netbeans.org/downloads/index.html
```

Baixe e execute a versão Java ou All do instalador — o instalador que você escolhe deve incluir o Java Web and EE e o Glassfish V2 UR2. As duas versões instalam o IDE do NetBeans e o servidor GlassFish. Supomos que você utilize as opções de instalação padrão. Depois de instalar o NetBeans, execute-o. Utilize então a opção Check for Updates do menu Help para certificar-se de que você tem os componentes mais recentes.

A maior parte do código que apresentamos neste capítulo é gerada pelo IDE do NetBeans. Reformatamos esse código e excluímos os comentários Javadoc que o IDE gera para corresponder com nossas convenções de codificação utilizadas por todo este livro e também para economizar espaço. Às vezes só mostramos uma parte do código. Nesses casos, fornecemos comentários indicando onde o código foi removido, e todos os números de linha correspondem ao código-fonte de exemplo completo.

29.2 Transações HTTP simples

Considere o que ocorre nos bastidores quando um usuário solicita uma página Web em um navegador. Na sua forma mais simples, uma página Web nada mais é do que um documento XHTML que descreve para um navegador como exibir e formatar as informações do documento. Documentos XHTML normalmente contêm hiperlinks que levam a diferentes páginas ou a outras partes da mesma página. Quando o usuário clica em um hiperlink, a página Web solicitada é carregada no navegador Web do usuário. Além disso, o usuário pode digitar o endereço de uma página no campo de endereço do navegador.

URIs

O protocolo HTTP permite a clientes e servidores interagir e trocar informações de uma maneira uniforme e confiável. O HTTP utiliza o URIs (Uniform Resource Identifier) para identificar dados na internet. URIs que especificam as localizações de documentos são chamados URLs (Uniform Resource Locators). URLs comuns fazem referência a arquivos, diretórios ou objetos podem realizar tarefas complexas, como pesquisas no banco de dados e pesquisas na internet. Se conhecer o URL de um recurso ou arquivo publicamente disponível em qualquer lugar na Web, você poderá acessá-lo por meio do HTTP.

Partes de um URL

Um URL contém informações que direcionam um navegador ao recurso que o usuário quer acessar. Computadores que executam softwares de servidor Web disponibilizam esses recursos. Vamos examinar os componentes do URL

```
http://www.deitel.com/books/downloads.html
```

O `http://` indica que o recurso deve ser obtido utilizando o protocolo HTTP. A parte `www.deitel.com`, é o **hostname** totalmente qualificado do servidor em que o recurso reside. O computador que hospeda e mantém recursos é conhecido como **host**. O `www.deitel.com` é convertido em um **endereço IP** — um valor numérico único que identifica o servidor, como um número de telefone define unicamente uma determinada linha telefônica. Essa conversão é realizada por um **servidor *Domain-Name System* (DNS)** — um computador que mantém um banco de dados dos nomes de host e seus endereços IP correspondentes — e o processo é chamado **pesquisa de DNS**. Para testar aplicativos Web, muitas vezes seu computador será usado como o host. Esse host é referenciado usando o nome de domínio reservado `localhost`, que costuma ser o endereço IP `127.0.0.1`. (Consulte informações adicionais sobre endereços IP em `en.wikipedia.org/wiki/IP_address`.) O hostname totalmente qualificado pode ser seguido por dois-pontos (`:`) e um número de porta. Servidores Web esperam por padrão as solicitações na porta 80, mas muitos servidores Web de desenvolvimento utilizam um número de porta diferente, como 8080 — como veremos na Seção 29.5.4.

O restante do URL (isto é, `/books/downloads.html`) especifica o nome do recurso solicitado (o documento XHTML `downloads.html`) e seu caminho ou localização (`/books`), no servidor Web. O caminho pode especificar a localização de um diretório real no sistema de arquivos do servidor Web. Mas, por razões de segurança, o caminho normalmente especifica a localização de um **diretório virtual**. O servidor converte o diretório virtual em uma localização real no servidor (ou em outro computador na rede do servidor), ocultando assim a localização real do recurso. Alguns recursos são criados dinamicamente usando outras informações armazenadas no computador do servidor, como um banco de dados. O hostname no URL para esse recurso especifica o servidor correto; as informações sobre o caminho e recurso que identificam o recurso com o qual interagir para responder à solicitação do cliente.

Fazendo uma solicitação e recebendo uma resposta

Quando dado um URL, um navegador Web realiza uma transação HTTP para recuperar e exibir a página Web nesse endereço. A Figura 29.1 ilustra a transação, mostrando a interação entre o navegador Web (o cliente) e o aplicativo de servidor Web (o servidor).

Na Figura 29.1, o navegador Web envia uma solicitação de HTTP ao servidor. A solicitação (na sua forma mais simples) é

```
GET /books/downloads.html HTTP/1.1
```

A palavra **`get`** é um **método HTTP** indicando que o cliente deseja obter um recurso do servidor. O restante da solicitação fornece o nome do caminho do recurso (por exemplo, um documento XHTML), o nome do protocolo e o número de versão (`HTTP/1.1`). A solicitação do cliente também contém alguns cabeçalhos obrigatórios e outros opcionais, como `Host` (que identifica o computador-alvo) ou `User-Agent` (que identifica o tipo e versão do navegador).

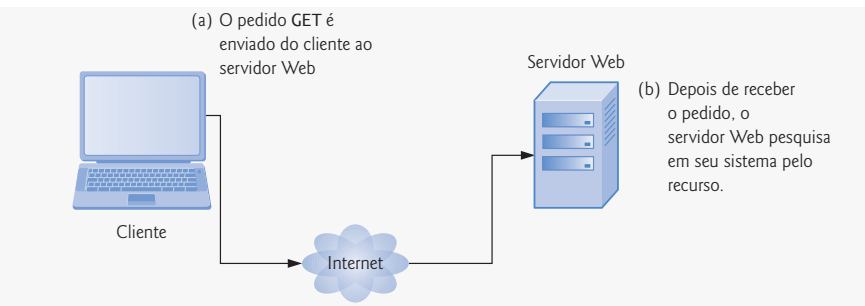


Figura 29.1 | Cliente interagindo com servidor Web. *Passo 1:* A solicitação `GET`.

Qualquer servidor que entende o HTTP (versão 1.1) pode converter essa solicitação e responder apropriadamente. A Figura 29.2 representa o servidor que responde a uma solicitação.

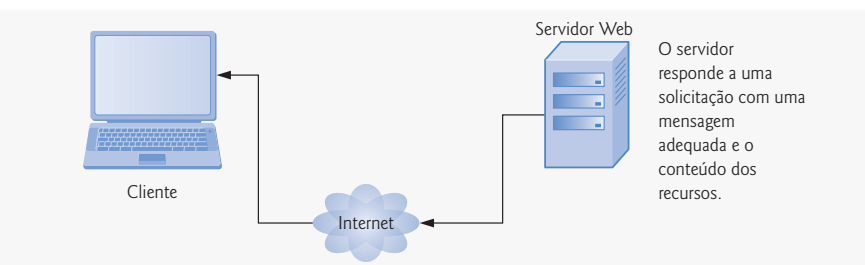


Figura 29.2 | Cliente interagindo com servidor Web. *Passo 2:* A resposta HTTP.

O servidor primeiro responde enviando uma linha do texto indicando a versão HTTP, seguida por um código numérico e uma frase descrevendo o status da transação. Por exemplo,

```
HTTP/1.1 200 OK
```

indica sucesso, enquanto

```
HTTP/1.1 404 Not found
```

informa o cliente que o servidor Web não pode localizar o recurso solicitado. Em uma solicitação bem-sucedida, o servidor acrescenta o documento solicitado à resposta HTTP. Uma lista completa dos códigos numéricos que indicam o status de uma transação HTTP pode ser encontrada em `www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.

Cabeçalhos HTTP

O servidor envia então um ou mais **cabeçalhos HTTP** que fornecem informações adicionais sobre os dados que serão enviados. Nesse caso, o servidor está enviando um documento de texto XHTML, portanto, um cabeçalho HTTP desse exemplo seria lido:

```
Content-type: text/html
```

As informações fornecidas nesse cabeçalho especificam o tipo **Multipurpose Internet Mail Extensions (MIME)** do conteúdo que o servidor está transmitindo ao navegador. O MIME é um padrão internet que especifica formatos de dados para que os programas possam interpretar os dados corretamente. Por exemplo, o tipo MIME `text/plain` indica que as informações enviadas são texto que pode ser exibido diretamente, sem qualquer interpretação do conteúdo como marcação XHTML. Da mesma forma, o tipo MIME `image/jpeg` indica que o conteúdo é uma imagem JPEG. Quando o navegador recebe esse tipo MIME, ele tenta exibir a imagem. Para uma lista de tipos MIME disponíveis, visite `www.w3schools.com/media/media_mimeref.asp`.

O cabeçalho ou conjunto de cabeçalhos é seguido por uma linha em branco, que indica ao navegador cliente que o servidor terminou de enviar os cabeçalhos HTTP. O servidor então envia o conteúdo do documento XHTML solicitado (`downloads.html`). O navegador do lado do cliente analisa a marcação XHTML que ele recebe e **renderiza** (ou exibe) os resultados. O servidor normalmente mantém a conexão aberta para processar outras solicitações do cliente.

Solicitações HTTP GET e POST

Os dois mais comuns **tipos de solicitação de HTTP** (também conhecidos como **métodos de solicitação**) são `GET` e `POST`. Uma solicitação `GET` em geral pede um recurso específico em um servidor. Os usos comuns das solicitações `GET` são recuperar uma imagem ou um documento XHTML ou, ainda, buscar resultados de pesquisa com base em um termo de pesquisa submetido pelo usuário. Uma solicitação `POST` geralmente posta (ou envia) dados para o servidor. Os usos comuns das solicitações `POST` são enviar dados de formulário ou documentos a um servidor.

Uma solicitação HTTP muitas vezes posta dados em um **handler de formulário do lado do servidor** que processa os dados. Por exemplo, quando um usuário realiza uma pesquisa ou participa de uma enquete baseada na Web, o servidor Web recebe as informações especificadas no formulário XHTML como parte da solicitação. Solicitações `GET` e `POST` podem ser utilizadas para enviar dados de formulário a um servidor Web, mas cada tipo de solicitação envia as informações de uma maneira diferente.

Uma solicitação `GET` envia informações ao servidor no URL, por exemplo, `www.google.com/search?q=deitel`. Nesse caso, `search` é o nome do handler de formulário no servidor do Google, `q` é o `name` de uma variável no formulário de pesquisa do Google e `deitel` é o termo da pesquisa. O `?` separa a **string de consulta** do restante do URL em uma solicitação. Um par de *nome/valor* é passado para o servidor com o *nome* e *o valor* separados por um sinal de igual (=). Se mais de um par de *nome/valor* for submetido, cada um é separado do próximo por caractere `&` (“e” comercial). O servidor utiliza os dados passados em uma string de consulta para recuperar um recurso apropriado do servidor. O servidor envia então uma **resposta** ao cliente. Uma solicitação `GET` pode ser iniciada submetendo um formulário XHTML cujo atributo `method` é configurado como `"get"`, digitando o URL (possivelmente contendo uma string de consulta) diretamente no barra de endereços do navegador ou por meio de um hiperlink.

Uma solicitação `POST` envia dados de formulário como parte da mensagem HTTP, não como parte do URL. Uma solicitação `GET` em geral limita a string de consulta (isto é, tudo à direita do `?`) a um número específico de caracteres (2.083 no Internet Explorer; mais em outros navegadores), assim frequentemente é necessário enviar grandes volumes de informações utilizando o método `POST`. O método `POST` também é às vezes preferido porque ele oculta os dados submetidos do usuário incorporando-os a uma mensagem HTTP.

Observação de engenharia de software 29.1

Os dados enviados em uma solicitação `POST` não são parte do URL, e o usuário não pode ver os dados por padrão. Mas há ferramentas disponíveis que exibem esses dados, portanto você não deve supor que os dados sejam seguros somente porque uma solicitação `POST` é usada.

Cache do lado do cliente

Navegadores muitas vezes armazenam em **cache** (salvam em disco) páginas Web para recarregamento rápido. Se não houver alterações entre a última versão armazenada no cache e a versão atual na Web, isso acelerará sua navegação. Uma resposta HTTP pode indicar por quanto tempo o conteúdo permanece atualizado. Se esse período de tempo não for atingido, o navegador poderá impedir outra solicitação ao servidor. Do contrário, o navegador solicita o documento do servidor. Portanto, o navegador minimiza o download da quantidade de dados

que deve ser feito para que você visualize uma página Web. Os navegadores em geral não armazenam em cache da resposta do servidor para uma solicitação `POST` porque a próxima solicitação `POST` pode não retornar o mesmo resultado. Por exemplo, em uma pesquisa, muitos usuários podem visitar a mesma página Web e responder uma pergunta. Os resultados de pesquisa então podem ser exibidos para o usuário. Cada nova resposta altera os resultados totais da pesquisa.

Ao utilizar um sistema de pesquisa baseado na Web, normalmente o navegador fornece as informações que você especifica em um formulário XHTML para o sistema de pesquisa com uma solicitação `GET`. O sistema de pesquisa realiza a pesquisa e, então, retorna os resultados para você como uma página Web. Essas páginas às vezes são armazenadas em cache pelo navegador caso você realize a mesma pesquisa novamente.

29.3 Arquitetura de aplicativo multithread

Aplicativos na Web são **aplicativos multicamadas** (às vezes conhecidos como **aplicativos** de *n* camadas). Aplicativos multicamadas dividem as funcionalidades em **camadas** separadas (isto é, agrupamentos lógicos das funcionalidades). Embora as camadas possam estar localizadas no mesmo computador, as camadas dos aplicativos baseados na Web costumam residir em computadores separados. A Figura 29.3 apresenta a estrutura básica de um aplicativo baseado na Web de três camadas.

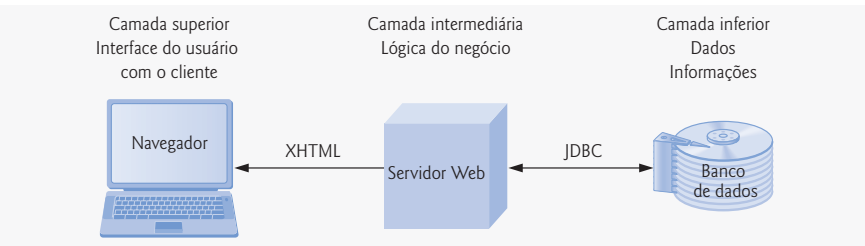


Figura 29.3 | Arquitetura de três camadas.

A **camada de informações** (também chamada **camada de dados** ou **camada inferior**) mantém os dados que pertencem ao aplicativo. Essa camada, em geral, armazena dados em um sistema de gerenciamento de banco de dados relacional (RDBMS). Discutimos RDBMSs no Capítulo 28. Por exemplo, uma loja de varejo poderia ter um banco de dados para armazenar informações sobre produtos como descrições, preços e quantidade em estoque. O mesmo banco de dados também poderia conter informações sobre clientes como nomes de usuário, endereços para cobrança e números de cartão de crédito. Essa camada pode conter múltiplos bancos de dados que, em conjunto, abrangem os dados necessários para nosso aplicativo.

A **camada intermediária** implementa a **lógica do negócio**, a **lógica do controlador** e a **lógica da apresentação** a fim de controlar interações entre os clientes do aplicativo e os dados do aplicativo. A camada intermediária funciona como um mediador entre os dados na camada de informações e os clientes do aplicativo. A lógica do controlador da camada intermediária processa as solicitações do cliente (como solicitações para visualizar um catálogo de produtos) e recupera dados do banco de dados. A lógica de apresentação da camada intermediária processa então os dados da camada de informações e apresenta o conteúdo ao cliente. Aplicativos Web costumam apresentar dados a clientes como documentos XHTML.

A lógica do negócio na camada intermediária impõe regras do negócio e assegura que os dados sejam confiáveis antes do aplicativo servidor atualizar o banco de dados ou apresentar os dados aos usuários. As regras do negócio determinam como os clientes podem e como não podem acessar dados de aplicativo, e como os aplicativos processam dados. Por exemplo, uma regra de negócio na camada intermediária do aplicativo na Web de uma loja de varejo poderia assegurar que a quantidade de todos os produtos permaneça positiva. Uma solicitação de cliente para configurar uma quantidade negativa no banco de dados de informações sobre produtos da camada inferior seria rejeitada pela lógica do negócio da camada intermediária.

A **camada cliente**, ou **camada superior**, é a interface com o usuário do aplicativo, que coleta a entrada e exibe a saída. Os usuários interagem diretamente com o aplicativo por meio da interface (em geral vista em um navegador Web), teclado e mouse. Em resposta às ações do usuário (por exemplo, clicar em um hiperlink), a camada de cliente interage com a camada intermediária para fazer solicitações e recuperar dados da camada de informações. A camada cliente exibe então os dados recuperados da camada intermediária ao usuário. A camada cliente nunca interage diretamente com a camada de informações.

29.4 Tecnologias Web Java

As tecnologias Web do Java evoluem continuamente a fim de fornecer aos desenvolvedores níveis mais altos de abstração e melhor separação das camadas do aplicativo, o que torna aplicativos Web mais fáceis de manter e mais extensíveis. Elas também permitem uma divisão efetiva do trabalho. Um designer gráfico pode construir a interface do aplicativo sem se preocupar com a lógica subjacente da página, que será tratada por um programador. Enquanto isso, você fica livre para se concentrar na lógica de negócio do aplicativo, deixando os detalhes da construção de um aplicativo atraente e fácil de usar ao designer. O NetBeans permite desenvolver a GUI de um aplicativo Web em uma ferramenta de design de arrastar e soltar enquanto permite tratar a lógica do negócio em classes Java separadas.

Aplicativos Java multicamadas costumam ser implementados utilizando os recursos do Java Enterprise Edition (Java EE). As tecnologias que utilizamos para desenvolver aplicativos Web nos capítulos 29–30 são parte do Java EE 5 (`java.sun.com/javaee`).

29.4.1 Servlets

Servlets são a visualização de nível mais baixo das tecnologias de desenvolvimento Web com Java que discutiremos neste capítulo. Eles utilizam o modelo de solicitação/resposta HTTP da comunicação entre o cliente e servidor.

Servlets estendem a funcionalidade de um servidor permitindo que ele gere conteúdo dinâmico. Por exemplo, servlets podem gerar dinamicamente documentos XHTML personalizados, ajudar a fornecer acesso seguro para um site Web, interagir com bancos de dados em nome de um cliente e manter informações de sessão únicas para cada cliente. Um componente de servidor Web chamado **contêiner de servlets** executa e interage com servlets. Os pacotes **`javax.servlet`** e **`javax.servlet.http`** fornecem as classes e interfaces para definir os servlets. O contêiner de servlets recebe solicitações HTTP de um cliente e direciona cada solicitação ao servlet apropriado. O servlet processa a solicitação e retorna uma resposta apropriada ao cliente — normalmente na forma de um documento XHTML ou XML (Extensible Markup Language) para exibir no navegador. A XML é uma linguagem utilizada para trocar dados estruturados na Web.

Arquitetonicamente, todos os servlets devem implementar a interface **`Servlet`** do pacote `javax.servlet`, que assegura que cada servlet pode executar no framework fornecido pelo contêiner de servlets. A interface `Servlet` declara os métodos utilizados pelo contêiner de servlets para gerenciar o ciclo de vida do servlet. Um ciclo de vida do servlet inicia quando o contêiner de servlets carrega-o na memória — geralmente em resposta à primeira solicitação ao servlet. Antes que o servlet possa tratar essa solicitação, o contêiner invoca o método `init` do servlet, que só é chamado uma vez durante o ciclo de vida de um servlet para inicializar o servlet. Depois que `init` completar a execução, o servlet estará pronto para responder à sua primeira solicitação. Todas as solicitações são tratadas pelo **método `service`** de um servlet, que é o método-chave para definir a funcionalidade de um servlet. O método `service` recebe a solicitação, processa-a e envia uma resposta ao cliente. Durante o ciclo de vida de um servlet, `service` é chamado uma vez por solicitação. Cada nova solicitação costuma ser tratada em um thread separado da execução (gerenciado pelo contêiner de servlets), assim cada servlet deve ser seguro a threads. Quando o contêiner de servlets termina o servlet (por exemplo, quando o contêiner de servlets precisa de mais memória ou quando ele é desativado), o método `destroy` do servlet é chamado para liberar quaisquer recursos mantidos pelo servlet.

29.4.2 JavaServer Pages

A tecnologia **JavaServer Pages (JSPs)** é uma extensão da tecnologia de servlet. Cada JSP é um documento que é convertido pelo contêiner JSP em um servlet. Diferentemente dos servlets, JSPs ajudam a separar a apresentação do conteúdo. As JavaServer Pages permitem aos programadores de aplicativo Web criar conteúdo dinâmico reutilizando componentes predefinidos e interagindo com componentes que utilizam script do lado do servidor. Os programadores de JSP podem utilizar componentes de softwares especiais chamados JavaBeans e bibliotecas de tags personalizados que encapsulam funcionalidades dinâmicas complexas. Um **JavaBean** é um componente reutilizável que segue certas convenções para o design de classe e que pode ser manipulado visualmente em uma ferramenta de construtor, como o NetBeans ou Eclipse. As classes JavaBean que permitem leitura e gravação das variáveis de instância devem fornecer métodos *get* e *set* apropriados, que, juntos, definem as propriedades das classes JavaBean. O conjunto completo das convenções do design de classe é discutido na especificação do JavaBean (`java-.sun.com/javase/technologies/desktop/javabeans/glasgow/index.html`).

Bibliotecas de tags personalizados

Lembre-se do Capítulo 23 de que documentos XHTML e XML, em geral, consistem em elementos que são delimitados por tags. **Bibliotecas de tags personalizados** são um recurso poderoso do JSP que permite aos desenvolvedores Java ocultar o código para acesso de banco de dados e outras operações complexas em **tags personalizados**. Para utilizar essas capacidades, você simplesmente adiciona os tags personalizados à página. Essa simplicidade permite a designers de página Web que não estão familiarizados com o Java aprimorar páginas Web com capacidades poderosas de processamento dinâmico e conteúdo dinâmico. As classes e interfaces JSP estão localizadas nos pacotes `javax-.servlet.jsp` e `javax.servlet.jsp.tagext`.

Componentes JSP

Há quatro componentes-chave para JSPs — diretivas, ações, elementos de script e bibliotecas de tags. **Diretivas** são mensagens ao **contêiner JSP** — o componente de servidor Web que executa JSPs. As diretivas permitem especificar configurações de página, incluir o conteúdo a partir de outros recursos e especificar bibliotecas de tags personalizados para uso em JSPs. As **ações** encapsulam funcionalidades em tags predefinidos que programadores podem incorporar em JSPs. As ações frequentemente são realizadas com base nas informações enviadas para o servidor como parte da solicitação de um cliente em particular. Elas também podem criar objetos Java para utilização em JSPs. Os **elementos de script** permitem inserir código Java que interage com componentes em uma JSP (e possivelmente outros componentes de aplicativo Web) para realizar o processamento de solicitação. As **bibliotecas de tags** fazem parte do **mecanismo de extensão de tag** que permite aos programadores criar tags personalizados. Tais tags permitem que os projetistas de página Web manipulem conteúdo da JSP sem conhecimento anterior do Java. A **JavaServer Pages Standard Tag Library (JSTL)** fornece funcionalidades para muitas tarefas comuns relacionadas a aplicativos Web, como iterar por uma coleção de objetos e executar instruções SQL.

Conteúdo estático

JSPs podem conter outro conteúdo estático. Por exemplo, as JSPs normalmente incluem marcação de XHTML ou XML. Tal marcação é conhecida como dados de **template fixa** ou **texto de template fixa**. Qualquer texto literal ou marcação XHTML em um JSP é convertido em uma `String` na representação servlet do JSP.

Processando uma solicitação JSP

Quando um servidor com JSP ativado recebe a primeira solicitação para uma JSP, o contêiner de JSPs traduz a JSP em um servlet Java que trata a solicitação atual e as futuras solicitações para a JSP. Portanto, JSPs contam com o mesmo mecanismo de solicitação/resposta que o dos servlets para processar solicitações e enviar respostas aos clientes.

Dica de desempenho 29.1

Alguns contêineres JSP convertem JSPs em servlets no momento da implantação do JSP (isto é, quando o aplicativo é instalado em um servidor Web). Isso elimina o overhead da conversão para o primeiro cliente que solicita cada JSP, uma vez que o JSP será convertido antes de ele ser solicitado por um cliente.

29.4.3 JavaServer Faces

JavaServer Faces (JSF) — suportado por servidores compatíveis com o Java Enterprise Edition 5 (Java EE 5), como o GlassFish v2 UR2 — é um framework de aplicativo Web que simplifica o design da interface com o usuário de um aplicativo e separa ainda mais a apresentação de um aplicativo Web da sua lógica de negócio. Um **framework** fornece bibliotecas e às vezes ferramentas de software para ajudá-lo a organizar e construir seus aplicativos. Embora o framework JSF possa utilizar muitas tecnologias para definir as páginas em aplicativos Web, este capítulo focaliza os aplicativos JSF que utilizam JavaServer Pages. O JSF fornece um conjunto de componentes de interface com o usuário, ou **componentes JSF**, que simplificam o design de páginas Web. Esses componentes são semelhantes aos componentes Swing utilizados para construir aplicativos GUI. O JSF fornece duas bibliotecas de tags personalizados JSP para adicionar esses componentes a uma página JSP. O JSF também inclui APIs para tratar eventos de componentes (como o processamento de modificações de estado dos componentes e validação da entrada de usuário), navegação entre páginas de aplicativo Web etc. Você cria a aparência e o funcionamento de uma página com o JSF adicionando elementos a um documento JSP e manipulando seus atributos. Você define o comportamento da página separadamente em arquivos de código-fonte Java relacionados.

Embora os componentes JSF padrão sejam suficientes para a maioria dos aplicativos Web básicos, você também pode escrever bibliotecas de componentes personalizados. Bibliotecas de componentes adicionais são disponibilizadas por vários projetos de código-fonte aberto e de fornecedores independentes. Por exemplo, o Oracle fornece quase 100 componentes na sua biblioteca ADF Faces. [*Nota*: Há muitos outros frameworks de aplicativo Web populares, incluindo Spring, Struts e Hibernate — todos os quais também são suportados no NetBeans. Optamos por demonstrar apenas o JavaServer Faces.]

29.4.4 Tecnologias Web no NetBeans

Aplicativos Web NetBeans que utilizam o framework JavaServer Faces consistem em uma ou mais páginas Web JSP. Esses arquivos JSP contêm a extensão de nome de arquivo `.jsp` e também elementos GUI das páginas Web. Os JSPs também podem conter JavaScript para adicionar funcionalidades à página. Os JSPs podem ser personalizados no NetBeans adicionando componentes JSF, incluindo rótulos, campos de texto, imagens, botões e outros componentes GUI. O IDE permite criar páginas visualmente arrastando e soltando esses componentes em uma página; você também pode personalizar uma página Web editando o arquivo `.jsp` manualmente.

Cada arquivo JSP criado no NetBeans representa uma página Web e tem uma classe JavaBean correspondente chamada **bean de página**. Uma classe JavaBean deve ter um construtor padrão (ou sem argumento), e métodos *get* e *set* para todas as propriedades do bean (isto é, variáveis de instância). O bean de página define as propriedades para todos os elementos da página com os quais você quer interagir programaticamente. O bean de página também contém handlers de evento e métodos de ciclo de vida de página para gerenciar tarefas, como inicialização e renderização de página, e outros códigos de suporte ao aplicativo Web.

Cada aplicativo Web NetBeans JSF define três mais JavaBeans. O objeto `RequestBean` é mantido no **escopo de solicitação** — ele só existe pelo tempo de duração da solicitação HTTP. Um objeto `SessionBean` tem **escopo de sessão** — ele existe por toda a sessão de navegação de um usuário ou até que a sessão expire. Há um único objeto `SessionBean` para cada usuário. Por fim, o objeto `ApplicationBean` tem **escopo de aplicativo** — ele é compartilhado por todas as instâncias de um aplicativo e existe pelo tempo que o aplicativo permanece implantado em um servidor Web. Esse objeto é utilizado para armazenagem ou processamentos de dados por todo o aplicativo; só há uma instância para o aplicativo, independentemente do número de sessões abertas.

29.5 Criando e executando um aplicativo simples no NetBeans

Nosso primeiro exemplo exibe a hora do dia do servidor Web em uma janela do navegador. Ao executar, esse programa exibe o texto `"Current time on the web server:"`, seguido pela hora do servidor Web. O aplicativo contém uma única página Web e, como mencionado anteriormente, consiste em dois arquivos relacionados — um documento JSP (Figura 29.4) e um arquivo de bean de página de suporte (Figura 29.6). O aplicativo também tem três beans de dados com escopo para os escopos de solicitação, sessão e aplicativo que não são utilizados nesse exemplo. Primeiro discutimos a marcação no documento JSP, a saída do aplicativo e o código no arquivo de bean de página, depois fornecemos instruções passo a passo para criar esse aplicativo Web. [*Nota*: A marcação na Figura 29.4 e outras listagens de arquivos JSP neste capítulo são as mesmas da marcação que aparece no NetBeans, mas reformatamos essas listagens para fins de apresentação.]

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Figura 29.4 Time.jsp -->
4  <!-- Documento JSP gerado pelo NetBeans que exibe -->
5  <!-- hora atual no servidor Web -->
```

```
6  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
7     xmlns:h="http://java.sun.com/jsf/html"
8     xmlns:jsp="http://java.sun.com/JSP/Page"
9     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
10    <jsp:directive.page contentType="text/html;charset=UTF-8"
11       pageEncoding="UTF-8"/>
12    <f:view>
13       <webuijsf:page id="page1">
14          <webuijsf:html id="html1">
15             <webuijsf:head id="head1" title="Web Time: A Simple Example">
16                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
17                <webuijsf:meta content="60" httpEquiv="refresh"/>
18             </webuijsf:head>
19             <webuijsf:body id="body1" style="-rave-layout: grid">
20                <webuijsf:form id="form1">
21                   <webuijsf:staticText id="timeHeader"
22                      style="font-size: 18px; left: 24px; top: 24px;
23                         position: absolute"
24                      text="Current time on the web server:"/>
25                   <webuijsf:staticText binding="#{Time.clockText}"
26                      id="clockText" style="background-color: black;
27                         color: yellow; font-size: 18px; left: 24px;
28                         top: 48px; position: absolute"/>
29                </webuijsf:form>
30             </webuijsf:body>
31          </webuijsf:html>
32       </webuijsf:page>
33    </f:view>
34 </jsp:root>
```

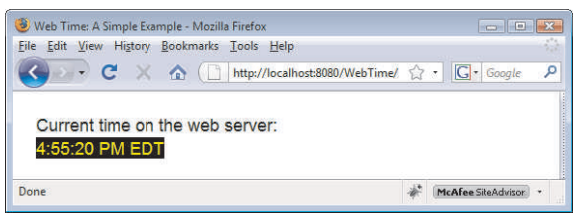


Figura 29.4 | O documento JSP gerado pelo NetBeans que exibe a hora atual no servidor Web.

NetBeans gera a maior parte da marcação mostrada na Figura 29.4 quando você configura o título da página Web, arrasta dois componentes Static Text até a página e especifica as propriedades dos componentes Static Text. Componentes **Static Text** exibem um texto que não pode ser editado pelo usuário. Mostramos esses passos mais adiante.

29.5.1 Examinando um documento JSP

Os documentos JSP neste e nos próximos exemplos são gerados quase inteiramente pelo NetBeans, que fornece um editor visual que permite construir a GUI de uma página arrastando e soltando componentes em uma área de design. O IDE gera um arquivo JSP em resposta às suas interações. A linha 1 da Figura 29.4 é a declaração XML, indicando o fato de que o JSP é expresso em sintaxe XML e a versão da XML que é utilizada. As linhas 3–5 são comentários que adicionamos ao JSP para indicar o número da figura, o nome do arquivo e o objetivo.

A linha 6 inicia o elemento raiz do JSP. Todos os JSPs precisam ter esse elemento **`jsp:root`**, que tem um atributo `version` para indicar a versão JSP que é utilizada (linha 6) e um ou mais atributos `xmlns` (linhas 6–9). Cada **atributo `xmlns`** especifica um prefixo e um URL para uma biblioteca de tags, permitindo que a página utilize elementos dessa biblioteca. Por exemplo, a linha 8 permite que a página utilize os elementos JSP padrão, mas o tag de cada elemento deve ser precedido pelo prefixo `jsp`. Todos os JSPs gerados pelo NetBeans incluem as bibliotecas de tags especificadas nas linhas 6–9 (a biblioteca de componentes básicos JSF, a biblioteca de componentes JSF HTML, a biblioteca de componentes padrão JSP e a biblioteca de componentes de interface com o usuário JSF).

As linhas 10–11 são o elemento **`jsp:directive.page`**. Seu atributo `contentType` especifica o tipo MIME (`text/html`) e o conjunto de caracteres (`UTF-8`) que a página utiliza. O atributo `page-Encoding` especifica a codificação de caracteres utilizada pelo código-fonte da página. Esses atributos ajudam o cliente (em geral um navegador Web) a determinar como exibir o conteúdo.

Todas as páginas que contêm componentes JSF são representadas em uma **árvore de componentes** (semelhante àquela mostrada na Figura 29.5) com a raiz do elemento JSF **`f:view`**, que é do tipo **`UIViewRoot`**. Essa estrutura de árvore de componentes é representada em um JSP aninhando todos os tags dos componentes JSF dentro do elemento `f:view` (linhas 12–33 nesse exemplo).

As linhas 13–18 iniciam a definição do JSP com os elementos **`webuijsf:page`**, **`webuijsf:html`** e **`webuijsf::head`**, todos provenientes da biblioteca de tags `webuijsf` (componentes da interface com o usuário JSF). O elemento `webuijsf:head` (linhas 15–18) tem um atributo `title` que especifica o título da página. Esse elemento também contém um elemento **`webuijsf:link`** (linha 16) que especifica a folha de estilo CSS utilizada pela página, e um elemento **`webuijsf:meta`** (linha 17) que atualiza a página a cada 60 segundos. O elemento **`webuijsf:body`** (linhas 19–30) contém um elemento **`webuijsf:form`** (linhas 20–29), que contém dois componentes

webuijsf:staticText (linhas 21–24 e 25–28) que exibem o texto da página. O componente `timeHeader` (linhas 21–24) tem um atributo `text` (linha 24) que especifica o texto a exibir (isto é, `"Current time on the web server"`). O componente `clockText` (linhas 25–28) não especifica um atributo `text` porque o texto desse componente será configurado programaticamente.

O elemento `webuijsf:staticText` nas linhas 25–28 tem o atributo `binding = "#{Time.clockText}"` (linha 25). Esse atributo utiliza a notação **JSF Expression Language** (isto é, `#{Time.clockText}`) para referenciar a propriedade `clockText` na classe `Time` que representa o bean de página (você verá essa classe na Figura 29.6). Você pode vincular um atributo de um elemento JSP ao valor de uma propriedade em quaisquer JavaBeans do aplicativo Web. Por exemplo, o atributo `text` de um componente `webuijsf:staticText` pode ser vinculado a uma propriedade `int` no `SessionBean` do aplicativo (como veremos na Seção 29.7.2).

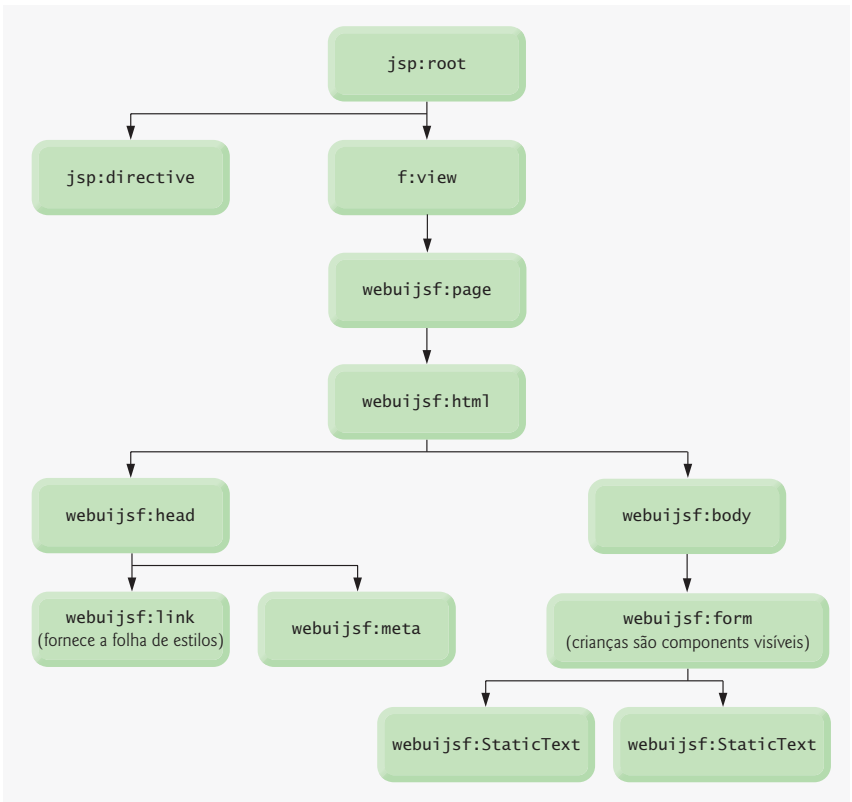


Figura 29.5 | Árvore de componentes JSF de exemplo dentro de um documento JSP.

Todos os elementos dos JSPs são mapeados pelo servidor de aplicativo para uma combinação de elementos XHTML e código JavaScript que permite ao navegador exibir a página. O JavaScript é uma linguagem de criação de scripts que é interpretada em todos os atuais navegadores Web populares. Ele pode ser utilizado para executar tarefas que manipulam elementos de página Web em um navegador Web e fornecem interatividade com o usuário. Você pode aprender mais sobre JavaScript no nosso JavaScript Resource Center em `www.deitel.com/JavaScript/`.

O mesmo componente Web pode mapear para diferentes elementos XHTML e código JavaScript, dependendo do navegador cliente e das configurações de propriedade do componente. Os componentes `webuijsf:staticText` (linhas 21–24, 25–28) costumam mapear para elementos XHTML **span**. Um elemento `span` contém o texto que é exibido em uma página Web e é, em geral, utilizado para controlar a formatação do texto. Os atributos `style` do elemento `webuijsf:staticText` de um JSP são mapeados para o correspondente atributo `style` do elemento `span`, que o navegador então utiliza para controlar a aparência do elemento.

29.5.2 Examinando um arquivo Page Bean

A Figura 29.6 apresenta o arquivo de bean de página autogerado. A linha 11 inicia a declaração da classe `Time` e indica que ela estende a classe **AbstractPageBean** (do pacote **com.sun.rave.web.ui.appbase**). Todas as classes de bean de página que suportam documentos JSP com componentes JSF devem estender a classe abstrata `Abstract-Page-Bean`, que fornece métodos de ciclo de vida de página. Observe que o IDE faz o nome de classe corresponder ao nome da página. O pacote **com.sun.webui.jsf.component** fornece classes para muitos dos componentes básicos da interface com usuário JSF, incluindo o componente `StaticText`.

```
1  // Figura 29.6: Time.java
2  // O arquivo de bean de página que configura clockText como a hora no servidor Web
3  package webtime;
4
5  import com.sun.rave.web.ui.appbase.AbstractPageBean;
```

```
6  import com.sun.webui.jsf.component.StaticText;
7  import javax.faces.FacesException;
8  import java.text.DateFormat;
9  import java.util.Date;
10
11 public class Time extends AbstractPageBean
12 {
13    private void _init() throws Exception
14    {
15    } // fim do método _init
16
17    private StaticText clockText = new StaticText();
18
19    public StaticText getClockText()
20    {
21       return clockText;
22    } // fim do método getClockText
23
24    public void setClockText( StaticText st )
25    {
26       this.clockText = st;
27    } // fim do método setClockText
28
29    public Time()
30    {
31    } // fim do construtor
32
33    @Override
34    public void init()
35    {
36       super.init();
37
38       try
39       {
40          _init();
41       } // fim do try
42       catch ( Exception e )
43       {
44          log( "Page1 Initialization Failure" , e );
45          throw e instanceof FacesException ? ( FacesException ) e :
46             new FacesException( e );
47       } // fim do catch
48    } // fim do método init
49
50    // o método chamado quando ocorre um postback
51    @Override
52    public void preprocess()
53    {
54    } // fim do método preprocess
55
56    // o método chamado antes de a página ser exibida
57    @Override
58    public void prerender()
59    {
60       clockText.setValue( DateFormat.getTimeInstance(
61          DateFormat.LONG ).format( new Date() ) );
62    } // fim do método prerender
63
64    // o método chamado depois da exibição completa, se init foi chamado
65    @Override
66    public void destroy()
67    {
68    } // fim do método
69
70    // retorna uma referência ao bean de sessão
71    protected SessionBean1 getSessionBean1()
72    {
73       return ( SessionBean1 ) getBean( "SessionBean1" );
74    } // fim do método
```

```
75
76       // retorna uma referência ao bean de solicitação
77       protected RequestBean1 getRequestBean1()
78       {
79          return ( RequestBean1 ) getBean( "RequestBean1" );
80       } // fim do método
81
82       // retorna uma referência ao bean de aplicativo
83       protected ApplicationBean1 getApplicationBean1()
84       {
85          return ( ApplicationBean1 ) getBean( "ApplicationBean1" );
86       } // fim do método
87    } // fim da classe Time
```

Figura 29.6 | O arquivo do bean de página que configura `clockText` como a hora no servidor Web.

Esse arquivo de bean de página define uma variável (linha 17) para interagir programaticamente com o elemento `clockText` do documento JSP da Figura 29.4. O IDE também autogera os métodos *get* e *set* (linhas 19–27) para essa variável, como veremos na Seção 29.5.4, *Passo*. A variável `clockText` é do tipo `StaticText`.

A única lógica requerida nessa página é configurar o texto do componente `clockText` como a hora atual no servidor. Fazemos isso no método `prerender` (linhas 58–62). O significado desse e de outros métodos de bean de página é discutido na Seção 29.5.3. As linhas 60–61 buscam e formatam a hora no servidor e configuram o valor de `clockText` como essa hora.

29.5.3 Ciclo de vida de processamento de eventos

Vários métodos no bean de página associam ao **ciclo de vida do processamento de eventos** JSF quatro etapas principais — inicialização, pré-processamento, pré-renderização e destruição. Cada um equivale a um método na classe de bean de página — `init`, `preprocess`, `prerender` e `destroy`, respectivamente. O NetBeans cria esses métodos sobrescritos para que você possa personalizá-los a fim de tratar tarefas de processamento de ciclo de vida, como exibição de um elemento em uma página somente se um usuário clicar em um botão.

O **método `init`** (Figura 29.6, linhas 34–48) é chamado pelo contêiner JSP na primeira vez que a página é solicitada e em postbacks. Um **postback** ocorre quando os dados de formulário são submetidos, e a página e seu conteúdo são enviados ao servidor para ser processados. O método `init` invoca sua versão da superclasse (linha 36) e então tenta chamar o método `_init` (declarado nas linhas 13–15). O método `_init` também é automaticamente gerado e trata tarefas de inicialização de componentes (se houver um), tais como configurar as opções de um grupo de botões de opção.

O **método `preprocess`** (linhas 52–54) é chamado depois de `init`, mas somente se a página estiver processando um postback. O **método `prerender`** (linhas 58–62) é chamado imediatamente antes de uma página ser renderizada (isto é, exibida) pelo navegador. Esse método deve ser utilizado para configurar as propriedades do componente; as propriedades que são configuradas mais cedo (como no método `init`) podem ser sobrescritas antes de a página ser de fato renderizada pelo navegador. Por essa razão, configuramos o valor de `clockText` no método `prerender`.

Por fim, o **método `destroy`** (linhas 66–68) é chamado depois que a página foi renderizada, mas somente se o método `init` for chamado. Esse método trata tarefas, como liberar os recursos utilizados para renderizar a página.

29.5.4 Criando um aplicativo Web no NetBeans

Agora que apresentamos o arquivo JSP, o arquivo de bean de página e a página Web XHTML resultante enviada ao navegador Web, discutiremos os passos para criar esse aplicativo. Para construir o aplicativo `WebTime`, siga estes passos no NetBeans:

Passo 1: criando o projeto do aplicativo Web

Selecione File> New Project... para exibir o diálogo New Project. Selecione Java Web no painel Categories, Web Application no painel Projects e clique em Next>. Mude o nome do projeto para `WebTime`. No campo Project Location, especifique onde você quer armazenar o projeto. Essas configurações criarão um diretório `WebTime` para armazenar os arquivos do projeto no diretório-pai que você especificou. Mantenha as outras configurações padrão e clique em Next>. Mantenha as opções padrão para Server and Settings e clique em Next>. Selecione Visual Web JavaServer Faces como o framework a utilizar nesse aplicativo Web, clique então em Finish para criar o projeto do aplicativo Web.

Passo 2: examinando a janela do editor visual do novo projeto

As várias figuras a seguir descrevem recursos importantes do NetBeans, começando com a janela do **editor visual** (Figura 29.7). O NetBeans cria uma página Web única nomeada `Page1` para cada novo projeto. Essa página é aberta por padrão no editor visual no **modo Design** quando o projeto é carregado pela primeira vez. À medida que você arrasta e solta novos componentes na página, o modo Design permite ver como o navegador irá renderizar a página. O arquivo JSP para essa página, nomeado `Page1.jsp`, pode ser visualizado clicando

no botão **JSP** na parte superior do editor visual ou clicando como botão direito do mouse em qualquer lugar no editor visual e selecionando **Edit JSP Source**. Como mencionado anteriormente, cada página da Web é suportada por um arquivo de bean de página. O NetBeans cria um arquivo nomeado `Page1.java` quando um novo projeto é criado. Para abri-lo, clique no botão **Java** na parte superior do editor visual ou clique com o botão direito do mouse em qualquer lugar no editor visual e escolha **Edit Java Source**.

O botão **Preview in Browser** na parte superior da janela do editor visual permite visualizar suas páginas em um navegador sem a necessidade de construir e executar o aplicativo inteiro. O botão **Refresh** redesenha a página no editor visual. O botão **Show Virtual Forms** permite ver quais elementos de formulário participam de formulários virtuais (discutidos no Capítulo 30). A lista drop-down **Target Browser Size** permite especificar a melhor resolução de navegador para visualizar a página e permite ver a aparência da página em diferentes resoluções da tela.

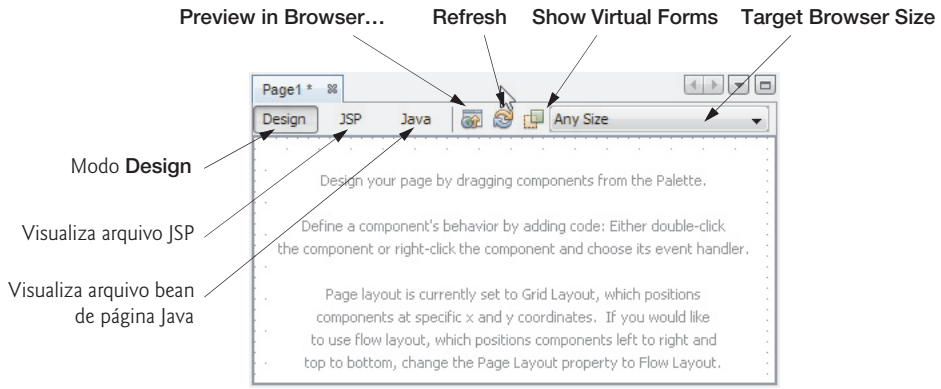
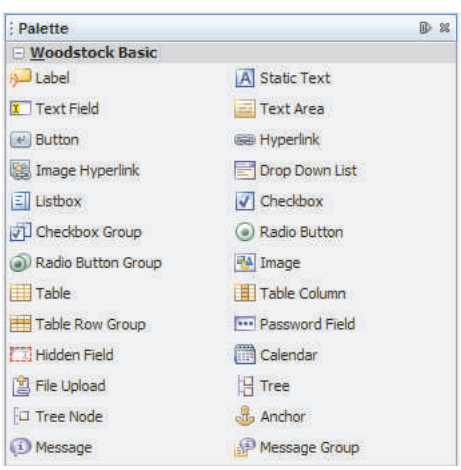


Figura 29.7 | Janela do editor visual no modo **Design**.

Passo 3: examinando a Palette no NetBeans

A Figura 29.8 mostra a **Pallete** exibida no IDE quando o projeto é carregado. A parte (a) exibe a lista **Woodstock Basic** dos componentes Web, e a parte (b) exibe os componentes **Woodstock Layout**. Woodstock é um conjunto dos componentes da interface com o usuário para aplicativos JavaServer Faces. Você pode aprender mais sobre esses componentes em `woodstock.dev.java.net`. Discutimos componentes específicos na Figura 29.8 uma vez que eles são utilizados por todo o capítulo.

(a) Categoria **Woodstock Basic** de **Palette**



(b) Categoria **Woodstock Layout** de **Palette** e outras categorias ocultas

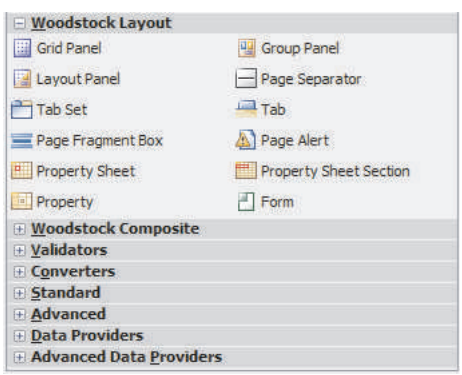


Figura 29.8 | **Palette** no NetBeans.

Passo 4: examinando a janela Projects

A Figura 29.9 exibe a janela **Projects** que aparece no canto superior esquerdo do IDE. Essa janela exibe a hierarquia de todos os arquivos incluídos no projeto. Os arquivos JSP para cada página são listados abaixo do nó **Web Pages**. Esse nó também inclui a pasta **resources** que contém folhas de estilo CSS para o projeto e todos os outros arquivos que as páginas talvez precisem para exibir apropriadamente, como arquivos de imagem. O código-fonte Java, incluindo o arquivo de bean de página para cada página Web e os beans de aplicativo, sessão e escopo de solicitação, pode ser encontrado sob o nó **Source Packages**.

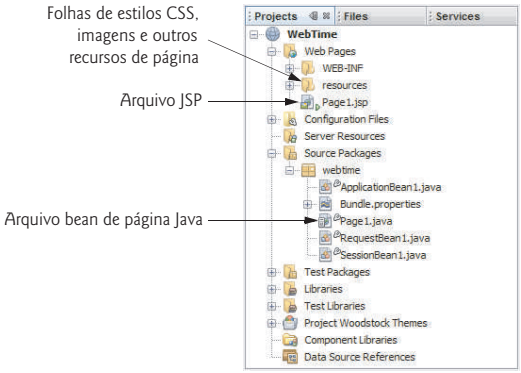


Figura 29.9 | Janela **Projects** para o projeto `WebTime`.

Passo 5: examinando arquivos JSP e Java no IDE

A Figura 29.10 exibe `Page1.jsp` — o documento JSP gerado pelo NetBeans para `Page1`. [*Nota:* Reformatamos o código para corresponder às nossas convenções de codificação.] Clique no botão **JSP** na parte superior do editor visual para visualizar esse arquivo. Quando esse arquivo é criado pela primeira vez, ele contém elementos para configurar a página, incluindo links para a folha de estilo da página e declaração das bibliotecas JSF que serão utilizadas. Por padrão, o NetBeans não mostra números de linha no editor de código-fonte. Para visualizar os números de linha, selecione **View> Show Line Numbers**.

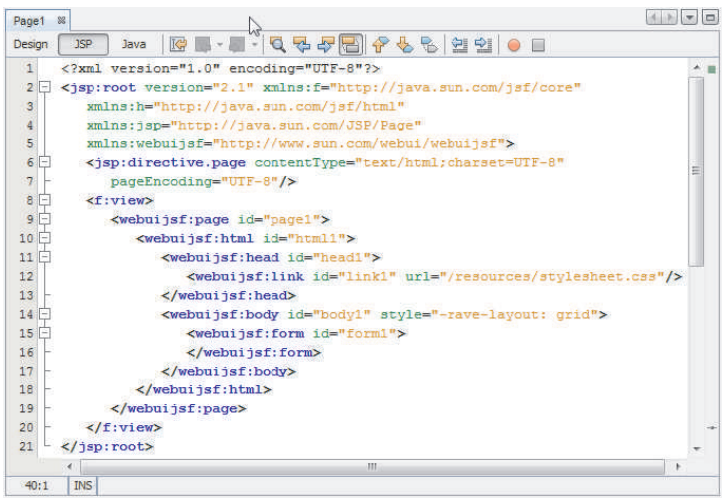


Figura 29.10 | O documento JSP gerado para Page1 pelo NetBeans.

A Figura 29.11 exibe parte do `Page1.java` — o arquivo de bean de página gerado pelo NetBeans para `Page1`. Clique no botão **Java** no editor para abrir o arquivo de bean de página. Esse arquivo contém uma classe Java com o mesmo nome que a página (isto é, `Page1`), que

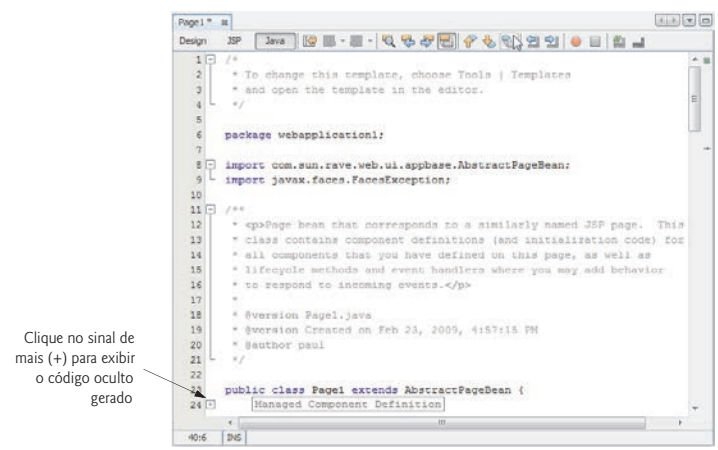


Figura 29.11 | Arquivo de bean de página para `Page1.jsp` gerado pelo NetBeans.

estende a classe `AbstractPageBean`. Como já mencionado, `AbstractPageBean` tem vários métodos que gerenciam o ciclo de vida da página. Quatro destes métodos — `init`, `preprocess`, `prerender` e `destroy` — são sobrescritos por `Page1.java`. Além do método `init`, esses métodos inicialmente estão vazios. Eles servem como reserva de espaço para personalizar o comportamento do seu aplicativo Web.

Passo 6: renomeando os arquivos JSP e Java

Em geral, é recomendável renomear os arquivos JSP e Java para que os nomes sejam relevantes ao seu aplicativo. Clique com o botão direito do mouse em `Page1.jsp` na janela **Projects** e selecione **Refactor> Rename** para exibir a caixa de diálogo **Rename Page1**. Digite o novo nome, `Time`, e marque a caixa de seleção **Apply Rename on Comments**. Para fazer as modificações imediatamente, clique no botão **Refactor**. Se você quiser visualizar as modificações antes que elas sejam aplicadas, clique no botão **Preview** para exibir a janela **Refactoring** na parte inferior do IDE. A **refatoração** é o processo de modificar o código-fonte para aprimorar sua legibilidade e resusabilidade sem alterar seu comportamento — por exemplo, renomeando métodos ou variáveis, ou dividindo métodos longos em métodos mais curtos. O NetBeans tem ferramentas de refatoração internas que automatizam algumas tarefas de refatoração. Usar essas ferramentas para renomear os arquivos de projeto atualiza o nome do arquivo JSP para `Time.jsp` e seu arquivo de bean de página correspondente para `Time.java`. A ferramenta de refatoração também muda o nome de classe no arquivo de bean de página e todas as vinculações de atributo no documento JSP para refletir o novo nome de classe. Se optar por visualizar as modificações de refatoração, as modificações só serão feitas depois de você clicar no botão **Do Refactoring** na janela **Refactoring**.

Passo 7: alterando o título da página

Antes de criar o conteúdo da página Web, atribuímos o título `"Web Time: A Simple Example"` a ela. Por padrão, a página não tem um título quando é gerada pelo IDE. Para adicionar um título, abra o arquivo JSP no modo **Design**. Na janela **Properties**, insira o novo título ao lado da propriedade **Title** e pressione *Enter*. Examine o JSP para ver se o atributo `title= "Web Time: A Simple Example"` foi automaticamente adicionado ao tag `webuijsf:head`. O título da página aparecerá na barra de título do navegador quando ela for exibida.

Passo 8: criando a página

Criar uma página Web é simples no NetBeans. Para adicionar componentes à página no modo **Design**, arraste e solte-os da **Palette** até a página. Cada componente é um objeto que tem propriedades, métodos e eventos. Você pode configurar essas propriedades e eventos visualmente utilizando a janela **Properties** ou programaticamente no arquivo de bean de página.

O IDE gera os tags JSP para os componentes que você arrasta e solta utilizando um layout de grade, como especificado no tag `webuijsf:body`. Os componentes são exibidos utilizando o **posicionamento absoluto** — eles aparecem exatamente onde são soltos na página. À medida que você adiciona componentes, o atributo `style` no elemento JSP de cada componente incluirá o número de pixels das margens superiores e esquerdas da página nas quais o componente é posicionado.

Esse exemplo utiliza dois componentes **Static Text**. Para adicionar o primeiro, arraste e solte-o da lista de componentes **Woodstock Basic** da **Palette** até a página. Edite o texto do componente digitando `"Current time on the web server:"` diretamente no componente. O texto também pode ser editado alterando a propriedade `text` do componente na janela **Properties**. O NetBeans é um editor WYSIWYG (What You See Is What You Get) — quando você faz uma modificação em uma página no modo **Design**, o IDE cria a marcação (visível no modo **JSP**) necessária para alcançar os efeitos visuais desejados vistos no modo **Design** e também poderia adicionar código Java ao arquivo `.java` do bean de página. Depois de adicionar o texto à página Web, alterne para o modo **JSP**. Você deve ver que o IDE aninhou um elemento `webuijsf:staticText` (com o atributo `text` que contém o texto que você acabou de inserir) no elemento `webuijsf:form` dentro do elemento `webuijsf:body`. O componente **Static Text** é vinculado ao objeto `staticText1` no arquivo de bean de página. De volta ao modo **Design**, clique no componente **Static Text** para selecioná-lo. Na janela **Properties**, clique no botão de reticências (...) ao lado da propriedade `style` a fim de abrir uma caixa de diálogo para editar o estilo do texto. Escolha `18 px` para o tamanho da fonte e clique em **OK**. Novamente na janela **Properties**, altere a propriedade `id` para `timeHeader`. Configurar a propriedade `id` também muda o nome da variável correspondente do componente no bean de página e atualiza seu atributo `binding` no JSP correspondentemente. O IDE agora deve se parecer como a Figura 29.12. Você pode examinar o arquivo JSP para ver se `font-size: 18 px` foi adicionado ao atributo `style` e se atributo `id` foi alterado para `timeHeader` no tag do componente.

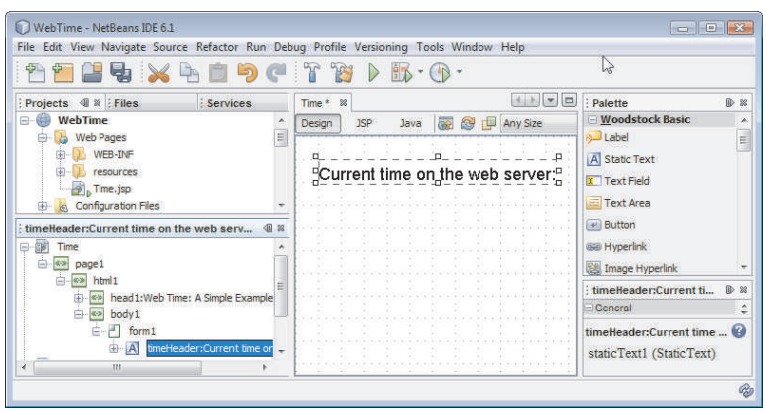


Figura 29.12 | `Time.jsp` depois de inserir o primeiro componente **Static Text**.

Solte um segundo componente **Static Text** na página e configure o `id` como `clockText`. Edite sua propriedade `style` para que o tamanho da fonte seja `18 px`, a cor do texto `yellow` e a cor do fundo `black`. Não edite o texto do componente, uma vez que isso será configurado programaticamente no arquivo de bean de página. O componente será exibido com o texto *Static Text* no IDE, mas ele só exibirá texto em tempo de execução se o texto foi configurado programaticamente. A Figura 29.13 mostra o IDE depois que o segundo componente é adicionado.

Passo 9: adicionando lógica de página

Depois de criar a interface com o usuário, você pode modificar o arquivo de bean de página para configurar o texto do elemento `clockText`. Para interagir com um componente JSF programaticamente, você primeiro precisa clique com o botão direito do mouse no controle no modo **Design** e escolha **Add Binding Attribute**. No arquivo de bean de página, isso cria uma variável que você pode utilizar para interagir com o componente bem como métodos *set* e *get* para acessar o componente.

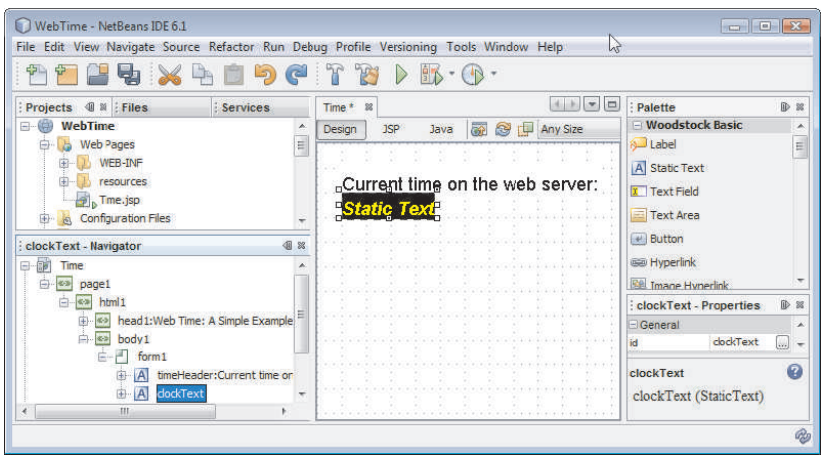


Figura 29.13 | `Time.jsp` depois de adicionar o segundo componente `StaticText`.

Nesse exemplo, adicionamos uma instrução ao método `prerender` (linhas 58–62 da Figura 29.6) no modo **Java**. Lembre-se de que utilizamos o método `prerender` para assegurar que `clockText` seja atualizado toda vez que a página for atualizada. As linhas 60–61 da Figura 29.6 configuram programaticamente o texto de `clockText` como a hora atual no servidor. Para que essa instrução funcione, você também precisará das duas `imports` mostradas nas linhas 8–9 da Figura 29.6. O IDE pode inserir essas instruções `import` para você. Simplesmente clique com o botão direito do mouse na janela do editor de código e escolha **Fix Imports** no menu pop-up que aparece.

Queremos que essa página seja atualizada uma vez por minuto para exibir uma hora atualizada. Para realizar isso, adicione `<webuijsf:meta content = "60" httpEquiv = "refresh"/>` ao documento JSP, como o último elemento aninhado no elemento `webuijsf:head`. Esse elemento instrui o navegador a recarregar a página automaticamente a cada 60 segundos. Você também pode adicionar esse tag arrastando um componente **Meta** da seção **Advanced** da **Palette** para sua página e então configurando o atributo `content` do componente como `60` e seu atributo `httpEquiv` como `refresh`. Se fizer isso, o componente **Meta** será exibido na janela **Navigator**.

Passo 10: examinando a janela Navigator

A Figura 29.14 exibe a **janela Navigator** no NetBeans. Expandimos o nó **Time**, que representa o arquivo de bean de página e mostra o conteúdo da árvore de componentes. Os beans de solicitação, sessão e escopo de aplicativo permanecem ocultos, uma vez que não adicionamos nenhuma propriedade a esses beans nesse exemplo. Clicar em um item na árvore de componentes da página seleciona o item no editor visual.

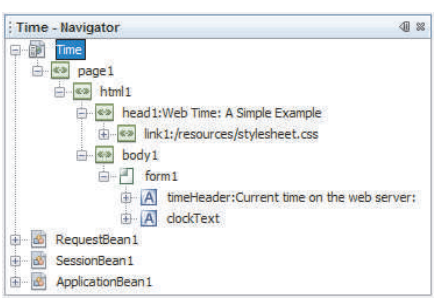


Figura 29.14 | Janela **Navigator** no NetBeans.

Passo 11: executando o aplicativo

Depois de criar a página Web, você pode visualizá-la de várias maneiras. Primeiro, selecione **Run> Build Project** e, depois da compilação terminar, selecione **Run> Run Project** para executar o aplicativo em uma janela de navegador. Você também pode executar um projeto que já foi compilado pressionando o ícone **Run Project** (▶) na barra de ferramentas na parte superior do IDE ou pressionando *F6*. Observe que o projeto

deve ser recompilado para refletir as alterações feitas quando o aplicativo for visualizado em um navegador — isso ocorrerá automaticamente se você executar o projeto sem o recompilar explicitamente. Como esse aplicativo foi compilado no sistema de arquivos local, o URL exibido na barra de endereços do navegador quando o aplicativo é executado será `http-://localhost:8080/WebTime/` (o Figura 29.4), onde `8080` é o número da porta no qual o servidor de teste — GlassFish v2 UR2 — executa por padrão.

Alternativamente, pressione *Ctrl + F5* para compilar o aplicativo e então executá-lo no modo de depuração — o depurador interno NetBeans pode ajudá-lo a solucionar problemas em aplicativos. (Visite `www.deitel.com/books/jhtp8/` para um vídeo que introduz o depurador NetBeans.) Se pressionar *F6*, o programa executará sem a depuração ativada.

Dica de prevenção de erro 29.1

Se tiver problemas para compilar seu projeto por causa de erros nos arquivos XML gerados pelo NetBeans utilizados para a compilação, tente limpar o projeto e o compilar novamente. Você pode fazer isso selecionando **Run > Clean and Build Project** *ou pressionando* Shift + F11.

Dica de prevenção de erro 29.2

Se tentar limpar e recompilar seu projeto e receber uma mensagem de erro indicando que um ou mais arquivos não podem ser excluídos, pare o servidor GlassFish e então tente novamente o processo de limpeza e recompilação. Para parar o servidor, clique na guia **Services** *no NetBeans, expanda o nó* **Servers**, *clique com o botão direito do mouse em* **GlassFish v2** *e escolha* **Stop**. *Depois de o servidor parar de executar, escolha* **Run> Clean and Build Project** *ou pressione* Shift + F11 *para limpar e recompilar o projeto.*

Por fim, você pode executar seu aplicativo compilado abrindo uma janela de navegador e digitando o URL da página Web no campo **Address**. Como seu aplicativo reside no sistema de arquivos local, você deve primeiro iniciar o servidor de aplicativo GlassFish. Se executou anteriormente o aplicativo utilizando um dos métodos citados, o servidor já estará inicializado. Do contrário, você pode iniciar o servidor a partir do IDE abrindo a guia **Services** (localizada no mesmo painel que o **Projects**), expandindo o nó **Servers**, dando um clique com o botão direito do mouse em **GlassFish v2** e selecionando **Start**. Você pode então digitar o URL (incluindo o número de porta para o servidor de aplicativo, 8080) no navegador para executar o aplicativo. Para esse exemplo não é necessário digitar o URL inteiro, `http://localhost:8080/WebTime/faces/Time.jsp`. O caminho para o arquivo `Time.jsp` (isto é, `faces/Time.jsp`) pode ser omitido, porque esse arquivo foi configurado por padrão como a página inicial do projeto. Para projetos com múltiplas páginas, você pode alterar a página inicial dando um clique com o botão direito do mouse na página desejada na janela **Projects** e selecionando **Set As Start Page**. A página inicial é indicada por um ícone com um símbolo de botão play verde () ao lado do nome da página na janela **Projects**.

29.6 Componentes JSF

Esta seção introduz alguns componentes JSF apresentados na **Palette** (Figura 29.8). A Figura 29.15 resume alguns componentes JSF utilizados nos exemplos do capítulo.

Componente JSF	Descrição
Label	Exibe texto que pode ser associado a um elemento de entrada.
Static Text	Exibe texto que o usuário não pode editar.
Text Field	Coleta entrada de usuário e exibe texto.
Button	Desencadeia um evento quando clicado.
Hyperlink	Exibe um hiperlink.
Drop Down List	Exibe uma lista drop-down de opções.
Radio Button Group	Botões de opção de grupos.
Image	Exibe imagens (por exemplo, GIF e JPG).

Figura 29.15 | Componentes JSF comumente utilizados.

29.6.1 Componentes gráficos e de texto

A Figura 29.16 exibe uma forma simples de coletar s entrada de usuário. Esse exemplo utiliza a maioria dos componentes na Figura 29.15. Todo o código na Figura 29.16 foi gerado pelo NetBeans em resposta a ações executadas no modo **Design**. Para criar esse aplicativo do zero, revise os passos na Seção 29.5.4. Esse exemplo não executa uma tarefa quando o usuário clica em **Register**. Mais adiante demonstramos como adicionar funcionalidades a muitos desses componentes.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Figura 29.16: WebComponents.jsp -->
<!-- Formulário de registro que demonstra componentes JSF -->
<jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
   <jsp:directive.page contentType="text/html;charset=UTF-8"
      pageEncoding="UTF-8"/>
   <f:view>
      <webuijsf:page id="page1">
         <webuijsf:html id="html1">
            <webuijsf:head id="head1">
               <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
            </webuijsf:head>
            <webuijsf:body id="body1" style="-rave-layout: grid">
               <webuijsf:form id="form1">
                  <webuijsf:staticText id="headerText"
                     style="font-size: 18px; left: 24px; top: 24px;
                        position: absolute"
                     text="This is a sample registration form"/>
                  <webuijsf:staticText id="instructionText"
                     style="font-style: italic; left: 24px; top: 48px;
                        position: absolute"
                     text="Please fill in all fields and click Register"/>
                  <webuijsf:image id="userImage" style="left: 24px;
                     top: 72px; position: absolute"
                     url="/resources/user.png"/>
                  <h:panelGrid columns="4" id="gridPanel"
                     style="height: 96px; left: 24px; top: 96px;
                        position: absolute" width="456">
                     <webuijsf:image id="firstNameImage"
                        url="/resources/fname.png"/>
                     <webuijsf:textField id="firstNameTextField"/>
                     <webuijsf:image id="lastNameImage"
                        url="/resources/lname.png"/>
                     <webuijsf:textField id="lastNameTextField"/>
                     <webuijsf:image id="emailImage"
                        url="/resources/email.png"/>
                     <webuijsf:textField id="emailTextField"/>
                     <webuijsf:image id="phoneImage"
                        url="/resources/phone.png"/>
                     <webuijsf:textField id="phoneTextField"/>
                  </h:panelGrid>
                  <webuijsf:image id="publicationsImage"
                     style="position: absolute; left: 24px; top: 216px"
                     url="/resources/publications.png"/>
                  <webuijsf:staticText id="publicationText"
                     style="left: 216px; top: 216px; position: absolute"
                     text="Which book would you like information about?"/>
                  <webuijsf:dropDown id="booksDropDown" items=
                     "#{WebComponents.booksDropDownDefaultOptions.
                        options}"
                     style="position: absolute; left: 24px; top: 240px"/>
                  <webuijsf:hyperlink id="deitelHyperlink"
                     style="position: absolute; left: 24px; top: 264px"
                     text="Click here to learn more about our books"
                     url="http://www.deitel.com"/>
                  <webuijsf:image id="osImage" style="left: 24px;
                     top: 312px; position: absolute"
                     url="/resources/os.png"/>
                  <webuijsf:radioButtonGroup id="osRadioButtonGroup"
                     items="#{WebComponents.
                        osRadioButtonGroupDefaultOptions.options}"
                     style="left: 24px; top: 336px; position: absolute"/>
                  <webuijsf:button id="registerButton" style="left: 359px;
                     top: 456px; position: absolute; width: 96px"
                     text="Register"/>
```

```
70                  </webuijsf:form>
71               </webuijsf:body>
72            </webuijsf:html>
73         </webuijsf:page>
74      </f:view>
75   </jsp:root>
```

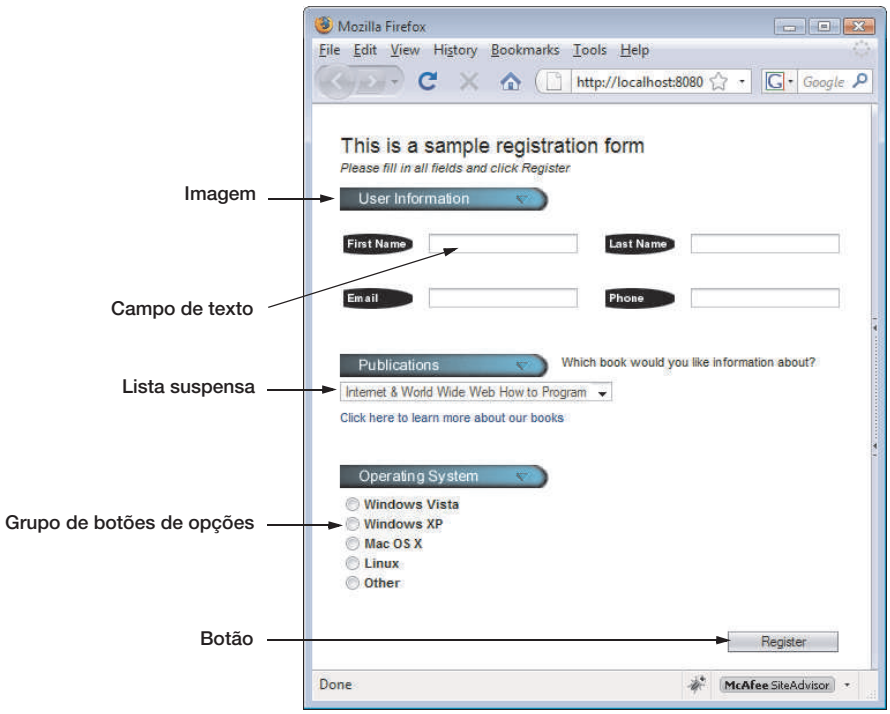


Figura 29.16 | Formulário de registro que demonstra componentes JSF.

Lembre-se de que NetBeans utiliza o posicionamento absoluto por padrão, portanto, os componentes são exibidos onde quer que eles sejam soltos no editor visual. Nesse exemplo, além do posicionamento absoluto, utilizamos um componente **Grid Panel** (linhas 30–45) do grupo de componentes Woodstock da Layout Palettte. O prefixo `h:` indica que ele pode ser encontrado na biblioteca de tags JSF HTML. Esse componente — um objeto da classe `HtmlPanelGrid` no pacote `javax.faces.component.html` — controla o posicionamento dos componentes que ele contém. O componente Grid Panel permite ao designer especificar o número de colunas que a grade deve conter. Os componentes podem então ser soltos dentro do painel, e eles serão automaticamente reposicionados em colunas uniformemente espaçadas na ordem na qual eles são soltos. Quando o número de componentes excede o número de colunas, o painel move os componentes adicionais para uma nova linha. O Grid Panel comporta-se como uma tabela XHTML e, na verdade, é exibido como uma tabela XHTML no navegador. Nesse exemplo, usamos o Grid Panel para controlar as posições dos componentes Image e Text Field na seção de informações do usuário da página.

Adicionando um componente de formatação a uma página Web

Para criar o layout para a seção User Information do formulário mostrado na Figura 29.16, arraste um componente Grid Panel até a página. Na janela Properties, altere o componente `id` para `gridPanel` e configure a propriedade `columns` do componente como `4`. O componente também tem propriedades para controlar o preenchimento de célula, espaçamento de célula e outros elementos da aparência do componente. Nesse caso, aceite os padrões para essas propriedades. Agora você pode simplesmente arrastar Imagens e Text Fields da seção informações de usuário até Grid Panel. O Grid Panel gerenciará o espaçamento e a organização em linhas e colunas.

Examinando componentes Web em um formulário de registro de exemplo

Aqui focalizaremos os novos elementos de interface com usuário no exemplo. As linhas 27–29 da Figura 29.16 definem uma **Image**, um objeto da classe `ImageComponent` que insere uma imagem em uma página Web. As imagens nesse exemplo estão localizadas no diretório `ch29\images`. As imagens a ser exibidas em uma página Web devem ser inseridas na pasta `resources` do projeto. Para adicionar imagens ao projeto, solte um componente Image na página e clique no botão de reticências (...) ao lado da propriedade url na janela Properties. Isso abre uma caixa de diálogo na qual você pode selecionar a imagem a exibir. Como ainda nenhuma imagem foi adicionada à pasta `resources`, clique no botão Add File, localize o diretório de imagens `ch29\images` e clique no botão Add File a fim de copiar o arquivo que você selecionou para a pasta `resources` do projeto na pasta `Web Pages`. Agora você pode selecionar a imagem e clicar em OK para inseri-la na página.

As linhas 30–45 contêm um elemento `h:panelGrid` que representa o componente Grid Panel. Dentro desse elemento, há oito componentes Image e **Text Field**. Text Fields permitem obter a entrada de texto do usuário. Por exemplo, a linha 35 define um controle Text Field utilizado para obter o primeiro nome. Você pode rotular um Text Field configurando a propriedade `label`, que posiciona o texto à esquerda do Text Field. Alternativamente, você pode rotular um Text Field arrastando e soltando um componente Label até a página, o que permite personalizar a posição e estilo do Label. Nesse exemplo, utilizamos imagens para indicar o objetivo de cada Text Field.

A ordem na qual Text Fields são arrastados até a página é a ordem em que seus elementos JSP são adicionados ao documento JSP. Por padrão, quando o usuário pressiona a tecla *Tab* para navegar entre campos de entrada, o foco alterna entre um componente e outro na ordem em que os elementos JSP ocorrem no documento JSP. Para especificar a ordem de navegação, arraste componentes até a página na ordem apropriada, ou configure que cada propriedade `tab-Index` do campo de entrada na janela Properties para configurar explicitamente a ordem das guias. Um componente com um índice de guia de 1 será o primeiro na sequência de guias.

As linhas 52–55 definem uma **Drop Down List**. Quando um usuário clica na lista drop-down, ela se expande e exibe uma lista em que o usuário pode fazer uma seleção. Esse componente é um objeto da classe `DropDown` e é vinculado ao objeto `booksDropDownDefault-Options`, um objeto `SingleSelectOptionsList` que gerencia a lista de opções. Esse objeto pode ser configurado automaticamente clicando com o botão direito do mouse na lista drop-down no modo Design e selecionando Configure Default Options..., que abre a caixa de diálogo Options Customizer para que você possa adicionar opções à lista. Cada opção consiste em uma `String` de exibição que o usuário visualizará no navegador e um valor `String` que é retornado quando você recupera a seleção do usuário de uma lista drop-down. O NetBeans constrói o objeto `SingleSelectOptionsList` no arquivo de bean de página com base nos pares de valor de exibição inseridos na caixa de diálogo Options Customizer. Para visualizar o código que constrói o objeto, feche a caixa de diálogo clicando em OK, abra o arquivo de bean de página e expanda o nó Managed Component Definition que vem depois da chave de abertura da definição de classe. O objeto é construído no método `_init`, que é chamado pelo método `init` na primeira vez que a página carrega.

O componente **Hyperlink** (linhas 56–59) da classe `Hyperlink` adiciona um hiperlink a uma página Web. A propriedade `url` desse componente especifica o recurso (`http://www.deitel.com` nesse caso) que é solicitado quando um usuário clica no hiperlink. Por padrão, componentes Hyperlink fazem páginas abrir na mesma janela de navegador, mas você pode configurar a propriedade `target` do componente para alterar esse comportamento.

As linhas 63–66 definem um componente **Radio Button Group** da classe `RadioButtonGroup`, que fornece uma série de botões de opção a partir da qual o usuário pode selecionar apenas um. Como um Drop Down List, um Radio Button Group é vinculado a um objeto `SingleSelectOptionList`. As opções podem ser editadas clicando com o botão direito do mouse no componente e selecionando Configure Default Options.... Assim como a lista drop-down, o `SingleSelectOptionsList` é automaticamente gerado pelo IDE e inserido no método `_init` da classe de bean de página.

As linhas 67–69 definem um componente **Button** da classe `Button` que desencadeia uma ação quando clicado. Em geral, um componente Button mapeia para um elemento XHTML `input` com o atributo `type` configurado como `submit`. Como afirmado antes, clicar no botão Register nesse exemplo não faz nada.

29.6.2 Validação utilizando componentes validadores e validadores personalizados

Validar a entrada de usuário é um passo importante ao coletar informações de usuários. A **validação** de formulário ajuda a evitar erros de processamento por causa de entrada de usuário incompleta ou impropriamente formatada. Por exemplo, você pode executar a validação para assegurar que todos os campos requeridos contenham dados ou que um campo de código do código postal tenha o número correto de dígitos. O NetBeans fornece três componentes validadores. Um **Length Validator** determina se um campo contém um número aceitável de caracteres. **Double Range Validators** e **Long Range Validators** determinam se a entrada numérica está dentro de intervalos aceitáveis. O pacote **`javax.faces.validators`** contém as classes para esses validadores. O NetBeans também permite validação personalizada com métodos validadores no arquivo de bean de página. O exemplo a seguir demonstra a validação utilizando tanto um validador de componente como uma validação personalizada.

Validando dados de formulário em um aplicativo Web

O exemplo nesta seção solicita que o usuário digite um nome, endereço de e-mail e número de telefone. Depois que o usuário insere quaisquer dados, mas antes que os dados sejam enviados ao servidor Web, a validação assegura que o usuário inseriu um valor em cada campo, que o nome digitado não excede 30 caracteres, e que o endereço de e-mail e os valores de número de telefone estão em um formato aceitável. Se o JavaScript não estiver ativado no cliente, a validação será então executada no servidor. Nesse exemplo, (555) 123–4567, 555–123–4567 e 123–4567 são todos considerados números de telefone válidos. Depois que os dados são submetidos, o servidor Web responde exibindo uma mensagem apropriada e um componente Grid Panel repetindo as informações submetidas. Observe que, em geral, um aplicativo do negócio real armazenaria os dados submetidos em um banco de dados ou em um arquivo no servidor. Simplesmente enviamos os dados de volta à página para demonstrar que o servidor recebeu os dados.

Criando a página Web

[*Nota*: Para criar esse aplicativo do zero, revise os passos na Seção 29.5.4.] Esse aplicativo Web introduz os componentes JSF Label e Message da seção Woodstock Basic da Palette. Cada um dos três campos de texto da página deve ter seu próprio rótulo e mensagem. Componentes **Label** descrevem outros componentes e podem ser associados a campos de entrada de usuário configurando a propriedade `for`. Componentes **Message** exibem mensagens de erro quando a validação falha. Essa página requer três Text Fields, três Labels e três Messages, bem como um Button de submissão. Para associar os componentes Label e componentes Message com seus componentes Text Field correspondentes,

mantenha pressionadas as teclas *Ctrl* e *Shift*, então arraste o rótulo ou a mensagem até o Text Field apropriado. Na janela Properties, observe que a propriedade `for` de cada componente Label e Message é configurada com o Text Field apropriado.

Você também deve adicionar um componente Static Text para exibir uma mensagem de sucesso de validação na parte inferior da página. Configure o texto como `"Thank you for your submission.<br/>We received the following information:"` e altere `id` do componente para `resultText`. Na janela Properties, redefina as propriedades `rendered` e `escape` do componente. A propriedade **`rendered`** controla se o componente será exibido na primeira vez que a página carrega. Configurar **`escape`** como `false` (isto é, desmarcado) permite que o navegador reconheça o tag `<br/>` de XHTML para que ele possa iniciar uma nova linha do texto em vez de exibir os caracteres `"<br/>"` na página Web.

Observações sobre a aparência e comportamento 29.1

Ao configurar propriedade `rendered` de um componente como `false` (desmarcada), o componente não mais aparecerá no editor visual. Para selecionar esse controle a fim de manipular as suas propriedades, utilize a janela Navigator *no modo* Design *(como mostrado na Figura 29.12).*

Adicione um componente Grid Panel abaixo do componente `resultText`. O painel deve ter duas colunas, uma para exibir componentes Static Text que rotulam os dados validados do usuário (nomeados `nameText`, `emailText` e `phoneText`, respectivamente), e uma para exibir componentes Static Text que exibem esses dados (nomeados `nameValueText`, `emailValueText` e `phoneValueText`, respectivamente). A propriedade `rendered` do painel deve ser configurada como `false` para que não seja exibida inicialmente.

Adicionando atributos de vinculação para interagir programaticamente com componentes

Lembre-se de que para cada controle com o qual você planeja interagir programaticamente, será necessário clicar com o botão direito do mouse no controle no modo Design e selecionar Add Binding Attribute para adicionar uma propriedade do controle ao arquivo de bean de página. Nesse exemplo, você deve fazer isso para cada um dos componentes Text Field (para poder obter seus valores), para o componente `resultText` Static Text (para poder exibi-lo), para o componente Grid Panel (para poder exibi-lo) e para os componentes Static Text `nameValueText`, `emailValueText` e `phoneValueText` no Grid Panel (para poder configurar o texto).

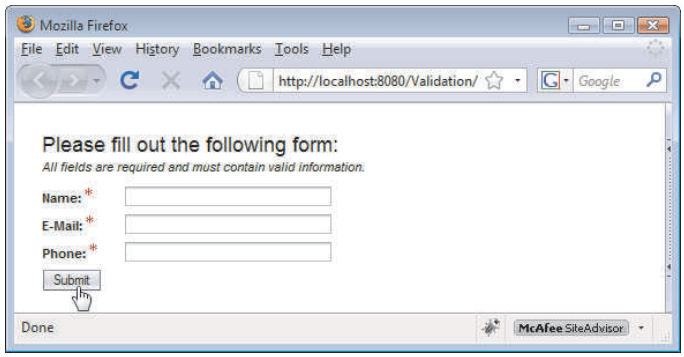
Revisando o documento JSP

O arquivo JSP para essa página é exibido na Figura 29.17. As linhas 31–35, 42–46 e 54–58 definem **`webuijsf:textFields`** para recuperar o nome, endereço de correio eletrônico e número de telefone do usuário, respectivamente. As linhas 28–30, 39–41 e 51–53 definem **`webuijsf:labels`** para cada um desses campos de texto. As linhas 36–38, 47–50 e 59–62 definem os elementos **`webuijsf:message`** dos campos de texto. As linhas 63–66 definem um Submit `webuijsf:button`. As linhas 67–71 criam um `webuijsf:staticText` nomeado `resultText` que exibe o texto quando o usuário submete com sucesso o formulário, e as linhas 72–91 definem um `h:panelGrid` que contém componentes para exibir a entrada de usuário validada no navegador.

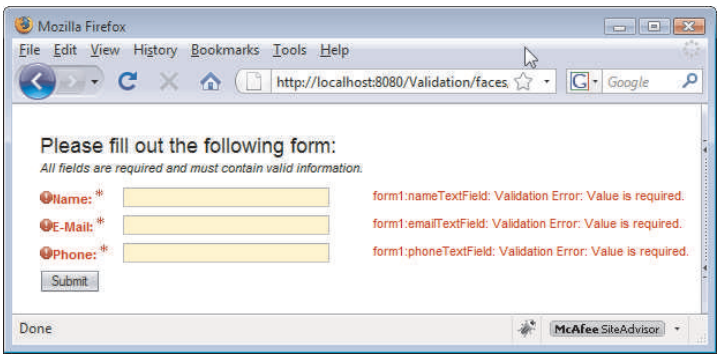
```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Figura 29.17 Validation.jsp -->
4  <!-- JSP que demonstra a validação da entrada de usuário. -->
5  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
6     xmlns:h="http://java.sun.com/jsf/html"
7     xmlns:jsp="http://java.sun.com/JSP/Page"
8     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
9     <jsp:directive.page contentType="text/html;charset=UTF-8"
10       pageEncoding="UTF-8"/>
11    <f:view>
12       <webuijsf:page id="page1">
13          <webuijsf:html id="html1">
14             <webuijsf:head id="head1">
15                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
16             </webuijsf:head>
17             <webuijsf:body id="body1" style="-rave-layout: grid">
18                <webuijsf:form id="form1">
19                   <webuijsf:staticText id="headerText"
20                      style="font-size: 18px; left: 24px; top: 24px;
21                         position: absolute"
22                      text="Please fill out the following form:"/>
23                   <webuijsf:staticText id="instructionText"
24                      style="font-style: italic; left: 24px; top: 48px;
25                         position: absolute"
26                      text="All fields are required and must contain
27                         valid information."/>
```

```
28                      <webuijsf:label for="nameTextField" id="nameLabel"
29                         style="left: 24px; top: 72px; position: absolute"
30                         text="Name:"/>
31                      <webuijsf:textField binding="#{Page1.nameTextField}"
32                         columns="30" id="nameTextField" required="true"
33                         style="position: absolute; left: 96px; top: 72px"
34                         validatorExpression=
35                            "#{Page1.nameLengthValidator.validate}"/>
36                      <webuijsf:message for="nameTextField" id="nameMessage"
37                         showDetail="false" showSummary="true"
38                         style="left: 312px; top: 72px; position: absolute"/>
39                      <webuijsf:label for="emailTextField" id="emailLabel"
40                         style="left: 24px; top: 96px; position: absolute"
41                         text="E-Mail:"/>
42                      <webuijsf:textField binding="#{Page1.emailTextField}"
43                         columns="30" id="emailTextField" required="true"
44                         style="left: 96px; top: 96px; position: absolute"
45                         validatorExpression=
46                            "#{Page1.emailTextField_validate}"/>
47                      <webuijsf:message for="emailTextField"
48                         id="emailMessage" showDetail="false"
49                         showSummary="true"
50                         style="left: 312px; top: 96px; position: absolute"
51                      <webuijsf:label for="phoneTextField" id="phoneLabel"
52                         style="left: 24px; top: 120px; position: absolute"
53                         text="Phone:"/>
54                      <webuijsf:textField binding="#{Page1.phoneTextField}"
55                         columns="30" id="phoneTextField" required="true"
56                         style="left: 96px; top: 120px; position: absolute"
57                         validatorExpression=
58                            "#{Page1.phoneTextField_validate}"/>
59                      <webuijsf:message for="phoneTextField"
60                         id="phoneMessage" showDetail="false"
61                         showSummary="true" style="left: 312px; top: 120px;
62                            position: absolute"/>
63                      <webuijsf:button actionExpression=
64                         "#{Page1.submitButton_action}" id="submitButton"
65                         style="position: absolute; left: 24px; top: 144px"
66                         text="Submit"/>
67                      <webuijsf:staticText binding="#{Page1.resultText}"
68                         escape="false" id="resultText" rendered="false"
69                         style="left: 24px; top: 168px; position: absolute"
70                         text="Thank you for your submission.&lt;br/&gt;We
71                            received the following information:"/>
72                      <h:panelGrid binding="#{Page1.gridPanel}"
73                         columns="2" id="gridPanel" rendered="false"
74                         style="border-width: 1px; border-style: solid;
75                            background-color: #ffff99; height: 96px;
76                            left: 24px; top: 216px; position: absolute"
77                         width="264">
78                         <webuijsf:staticText id="nameText" text="Name:"/>
79                         <webuijsf:staticText
80                            binding="#{Page1.nameValueText}"
81                            id="nameValueText"/>
82                         <webuijsf:staticText id="emailText"
83                            text="E-Mail:"/>
84                         <webuijsf:staticText
85                            binding="#{Page1.emailValueText}"
86                            id="emailValueText"/>
87                         <webuijsf:staticText id="phoneText" text="Phone:"/>
88                         <webuijsf:staticText
89                            binding="#{Page1.phoneValueText}"
90                            id="phoneValueText"/>
91                      </h:panelGrid>
92                   </webuijsf:form>
93                </webuijsf:body>
94             </webuijsf:html>
95          </webuijsf:page>
96       </f:view>
97    </jsp:root>
```

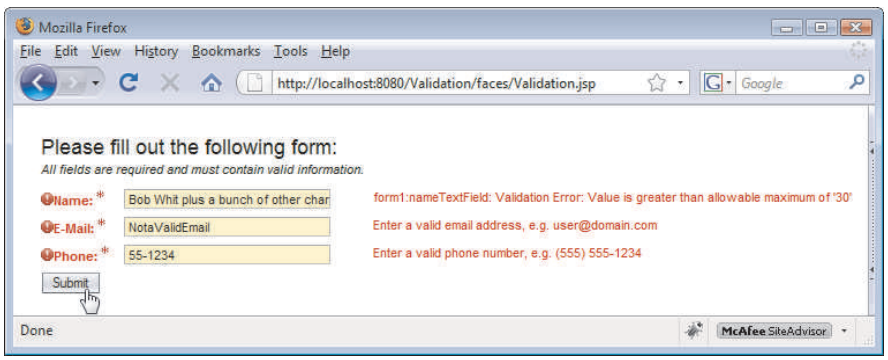
(a) Submetendo o formulário antes de inserir quaisquer informações



(b) Mensagens de erro exibidas depois de submeter o formulário vazio



(c) Mensagens de erro exibidas depois de submeter informações inválidas



(d) Formulário submetido com sucesso

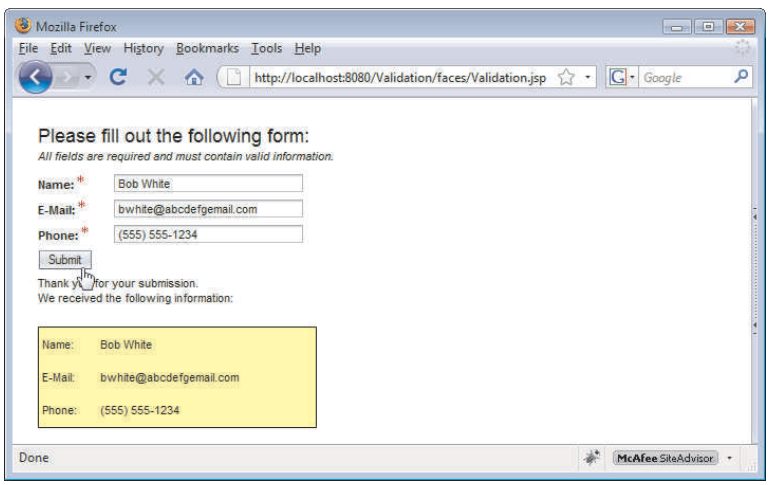


Figura 29.17 | O JSP que demonstra a validação da entrada de usuário.

Configurando a propriedade `Required` de um componente de entrada

Assegurar que o usuário fez uma seleção ou inseriu algum texto em um elemento de entrada requerido é um tipo básico de validação. Isso é realizado marcando a caixa de diálogo **`required`** na janela **Properties** do elemento. Se você adicionar um componente validador ou o método validador personalizado em um campo de entrada, a propriedade `required` do campo deve ser configurada como `true` (marcada) para a validação ocorrer. A propriedade `required` das três entradas `webuijsf:textFields` nesse exemplo é configurada como `true`. Também observe no editor visual que o rótulo para um campo requerido é automaticamente marcado por um asterisco vermelho. Se um usuário submeter esse formulário com campos de texto vazios, a mensagem de erro padrão para um campo requerido será exibida no componente `webuijsf:message` do campo vazio associado. Para personalizar a mensagem de erro, você deve fornecer um validador personalizado.

Utilizando o componente `LengthValidator`

Nesse exemplo, utilizamos o componente **Length Validator** (localizado na seção **Validators** da **Palette**) para assegurar que o comprimento do nome do usuário não excede 30 caracteres. Isso poderia ser útil para assegurar que um valor se ajustará em um determinado campo de banco de dados.

Para adicionar um **Length Validator** a um componente, simplesmente arraste o validador da **Palette** e solte-o no campo para validar. Um nó **lengthValidator1** aparecerá na janela **Outline**. Para editar as propriedades do componente de validação, clique nesse nó e configure as propriedades `maximum` e `minimum` como o número desejado de caracteres na janela **Properties**. Aqui, só configuramos a propriedade `maximum` em 30. Também alteramos o `id` do componente para `nameLengthValidator`. Observe que a propriedade `validatorExpression` do `nameTextField` foi vinculada ao método do `validate` `nameLengthValidator` no arquivo de bean de página (linhas 34–35). Lembre-se de que a maior parte da validação no lado do cliente pode ser burlada, portanto, validação importante sempre deve ser realizada no servidor.

Esse validador permite a usuários digitar quanto texto no campo eles quiserem, e se eles excederem o limite, a mensagem de erro padrão de validação de tamanho será exibida no componente `webuijsf:message` do campo depois que o usuário clicar no botão **Submit**. É possível limitar o tamanho da entrada de usuário sem utilizar a validação. Configurando a propriedade `maxLength` de um **Text Field**, o cursor do **Text Field** não ultrapassará o número admissível máximo de caracteres, assim o usuário não conseguira submeter dados que excedem o limite de tamanho.

Utilizando expressões regulares para executar validação personalizada

Algumas das tarefas de validação mais comuns envolvem validar a entrada de usuário. Por exemplo, talvez seja necessário verificar endereços de e-mail inseridos e números de telefone pelo usuário para assegurar que estejam em conformidade com o padrão de formatação para endereços de e-mail e números de telefone válidos. Corresponder a entrada de usuário contra uma expressão regular é um modo efetivo de assegurar que a entrada está formatada apropriadamente. Isso é frequentemente feito no lado do cliente antes que os dados sejam submetidos ao servidor para que o usuário seja notificado imediatamente se eles fornecerem entrada inválida. Os servidores costumam revalidar os dados também para objetivos de segurança. O NetBeans não fornece componentes para a validação utilizando expressões regulares, assim adicionaremos nossos próprios métodos validadores personalizados ao arquivo de bean de página. Para adicionar um validador personalizado a um componente de entrada, clique com o botão direito do mouse no componente e escolha **Edit Event Handler> validate**. Isso cria um método de validação para o componente com um corpo vazio no arquivo de bean de página. Adicionaremos o código a esse método mais adiante. Observe que os atributos `validatorExpression` de `emailTextField` e `phoneTextField` são vinculados aos seus respectivos métodos de validação personalizados no arquivo de bean de página (linhas 45–46 e 57–58).

Examinando no arquivo de bean de página um formulário que recebe entrada de usuário

A Figura 29.18 contém o arquivo de bean de página do arquivo JSP na Figura 29.17. A linha 21 configura o tamanho máximo para o `nameLengthValidator`, que é uma propriedade desse bean de página. Lembre-se de que o campo de texto de nome foi vinculado a essa propriedade no documento JSP. Os métodos `emailTextField_validate` (linhas 189–200) e `phoneTextField_validate` (linhas 204–216) são métodos validadores personalizados que verificam o endereço de correio eletrônico e número de telefone inserido pelo usuário, respectivamente. O método `submitButton_action` (linhas 219–230) ecoa os dados de volta ao usuário se a validação for bem-sucedida. Os métodos validadores são chamados antes do handler de evento, assim se a validação falhar, `submitButton_action` não será chamado e a entrada de usuário não será ecoada.

```
1  // Figura 29.18: Validation.java
2  // Validando a entrada de usuário.
3  package validation;
4
5  import com.sun.rave.web.ui.appbase.AbstractPageBean;
6  import com.sun.webui.jsf.component.StaticText;
7  import com.sun.webui.jsf.component.TextField;
8  import javax.faces.FacesException;
9  import javax.faces.application.FacesMessage;
10 import javax.faces.component.UIComponent;
```

```
11 import javax.faces.component.html.HtmlPanelGrid;
12 import javax.faces.context.FacesContext;
13 import javax.faces.validator.LengthValidator;
14 import javax.faces.validator.ValidatorException;
15
16 public class Validation extends AbstractPageBean
17 {
18    // Definição de componente gerenciado
19    private void _init() throws Exception
20    {
21       nameLengthValidator.setMaximum( 30 );
22    } // fim do método _init
23
24    // Para economizar espaço, omitimos o código nas linhas 24-186. O código-fonte
25    // completo é fornecido com os exemplos deste capítulo.
26
187   // valida o endereço de e-mail inserido contra a expressão regular
188   // isso representa a forma de um endereço de e-mail válido
189   public void emailTextField_validate( FacesContext context,
190      UIComponent component, Object value )
191   {
192      String email = String.valueOf( value );
193
194      // se o endereço de e-mail inserido não estiver em um formato válido
195      if ( !email.matches(
196         "\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*" ) ) {
197         throw new ValidatorException( new FacesMessage(
198            "Enter a valid e-mail address, e.g. user@domain.com" ) );
199      } // fim do if
200   } // fim do método emailTextField_validate
201
202   // valida o número de telefone inserido contra a expressão regular
203   // isso representa a forma de um número de telefone válido
204   public void phoneTextField_validate( FacesContext context,
205      UIComponent component, Object value )
206   {
207      String phone = String.valueOf( value );
208
209      // se o número de telefone inserido não estiver em um formato válido
210      if ( !phone.matches(
211         "((\\(\\d{3}\\) ?)|(\\d{3}-))?\\d{3}-\\d{4}" ) )
212      {
213         throw new ValidatorException( new FacesMessage(
214            "Enter a valid phone number, e.g. (555) 555-1234" ) );
215      } // fim do if
216   } // fim do método phoneTextField_validate
217
218   // exibe os valores que o usuário inseriu e submeteu
219   public String submitButton_action()
220   {
221      String name = String.valueOf(nameTextField.getValue() );
222      String email = String.valueOf(emailTextField.getValue() );
223      String phone = String.valueOf(phoneTextField.getValue() );
224      nameValueText.setValue( name );
225      emailValueText.setValue( email );
226      phoneValueText.setValue( phone );
227      gridPanel.setRendered( true );
228      resultText.setRendered( true );
229      return null;
230   } // fim do método submitButton_action
231 } // fim da classe Validation
```

Figura 29.18 | O bean de página para validar a entrada de usuário e reexibir essa entrada se válida.

Os dois métodos validadores personalizados nesse arquivo de bean de página validam o conteúdo de um campo de texto contra uma expressão regular utilizando o método `String` `matches`, que recebe uma expressão regular como um argumento e retorna `true` se a `String` estiver em conformidade com o formato especificado.

Para o método `emailTextField_validate`, utilizamos a seguinte expressão de validação

```
\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
```

Observe que cada barra invertida na expressão regular `String` (linha 196) deve ser escapada com outra barra invertida (como em `\\`), porque o caractere de barra invertida normalmente representa o início de uma sequência de escape no Java. Essa expressão regular indica que um endereço de e-mail é válido se a parte antes do símbolo `@` contiver um ou mais caracteres de texto (isto é, caracteres alfanuméricos ou sublinhados), seguidos por zero ou mais `String`s compreendidas por um hífen, sinal de adição, ponto ou apóstrofo e caracteres de texto adicionais. Depois do símbolo `@`, um endereço de e-mail válido deve conter um ou vários grupos de caracteres de texto potencialmente separados por hífens ou pontos, seguidos por um ponto requerido e outro grupo de um ou mais caracteres de texto potencialmente separados por hífens ou pontos. Por exemplo, os endereços de e-mail `personal.email@white.email.com`, `bob-white@my-email.com` e `bob.white@email.com` de `bob` são todos válidos. Se o usuário inserir texto em um `emailTextField` que não tem o formato correto e tentar submeter o formulário, as linhas 197–198 lançarão uma `ValidatorException`. O componente `Message` captura essa exceção e exibe a mensagem em vermelho.

A expressão regular em `phoneTextField_validate` assegura que `phoneTextField` contém um número de telefone válido antes de o formulário ser submetido. A entrada de usuário é correspondida contra a expressão regular

```
((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}
```

(Novamente, cada barra invertida é escapada na expressão regular `String` na linha 211.) Essa expressão indica que um número de telefone pode conter um código de área de três dígitos entre parênteses seguido por um espaço opcional ou sem parênteses e seguido por um hífen obrigatório. Depois de um código de área opcional, um número de telefone deve conter três dígitos, um hífen e outros quatro dígitos. Por exemplo, `(555) 123-4567`, `555-123-4567` e `123-4567` são todos números de telefone válidos. Se um usuário inserir um número de telefone inválido, as linhas 213–214 lançarão uma `ValidatorException`. O componente `Message` captura essa exceção e exibe a mensagem de erro em vermelho.

Se os seis validadores forem bem-sucedidos (isto é, cada `TextField` contém dados, o nome terá menos de 30 caracteres e o endereço de e-mail e número de telefone são válidos), clicar no botão **Submit** envia os dados do formulário ao servidor. Como mostrado na Figura 29.17 (d), o método `submitButton_action` exibe os dados submetidos no componente `gridPanel` de Grid Panel (linhas 221–227) e exibe o componente `resultsText` **Static Text** (linha 228).

29.7 Monitoramento de sessão

Nos primórdios da internet, e-businesses não podiam fornecer o tipo de serviço personalizado que lojas físicas (*brick-and-mortar*) geralmente forneciam. Para resolver esse problema, lojas virtuais começaram a estabelecer mecanismos por meio dos quais eles poderiam personalizar experiências de navegação dos usuários, personalizando o conteúdo para usuários específicos e, ao mesmo tempo, permitindo que eles pulassem informações irrelevantes. Os negócios alcançam esse nível de serviço monitorando o movimento de cada cliente nos seus sites Web e combinando os dados coletados com as informações fornecidas pelos consumidores, incluindo informações de cobrança, preferências pessoais, interesses e hobbies.

Personalização

A **personalização** permite que sites de comércio eletrônico se comuniquem efetivamente com seus clientes e também aprimora a capacidade do usuário de localizar produtos e serviços desejados.. Empresas que fornecem conteúdo de interesse particular para usuários podem estabelecer relacionamentos com clientes e aprimorar esses relacionamentos ao longo do tempo. Além disso, visando aos consumidores com ofertas pessoais, recomendações, anúncios, promoções e serviços, as lojas de comércio eletrônico estimulam a fidelidades dos clientes. Sites Web podem utilizar tecnologias sofisticadas para permitir que visitantes personalizem home pages para que estas se ajustem às suas necessidades e preferências individuais. De maneira semelhante, sites de compra on-line muitas vezes armazenam informações pessoais para os clientes, personalizando notificações e ofertas especiais de acordo com seus interesses. Esses serviços estimulam os clientes a visitar os sites e fazer compras mais frequentemente.

Privacidade

Há, porém, um dilema entre serviços de comércio eletrônico personalizados e proteção da privacidade. Alguns consumidores aceitam a ideia do conteúdo personalizado, mas outros temem as possíveis consequências adversas se as informações que eles fornecem aos sites de comércio eletrônico forem publicadas ou coletadas por tecnologias de rastreamento. Consumidores e defensores da privacidade perguntam: E se os e-businesses aos quais fornecemos dados pessoais venderem ou disponibilizarem essas informações a outra organização sem nosso conhecimento? E se não quisermos que nossas ações na Internet — um meio supostamente anônimo — sejam rastreadas e registradas por partes desconhecidas? E se as partes não autorizadas ganharem acesso aos dados privados confidenciais, como números de cartão de crédito ou histórico médico? Todas essas são questões que devem ser discutidas e abordadas por programadores, consumidores, sites de comércio eletrônico e legisladores.

Reconhecendo clientes

Para fornecer serviços personalizados aos consumidores, os sites de comércio eletrônico devem ser capazes de reconhecer os clientes quando eles solicitam informações de um site. Como discutido, o sistema de solicitação/resposta no qual a Web opera é facilitado pelo HTTP. Infelizmente, o HTTP é um **protocolo sem estado** — ele não suporta conexões persistentes que permitiriam aos servidores Web manter informações de estado em relação a determinados clientes. Desse modo, os servidores Web não podem determinar se uma solicitação vem de um determinado cliente ou se uma série de solicitações vem de um ou mais clientes. Para superar esse problema, os sites podem fornecer mecanismos para identificar clientes específicos. Uma sessão representa um único cliente em um site Web. Se o cliente sair de um site e então retornar mais tarde, o cliente ainda será reconhecido como o mesmo usuário. Para ajudar o servidor a distinguir entre clientes, cada cliente deve identificar-se para o servidor.

O rastreamento de clientes específicos, conhecido como **rastreamento de sessão**, pode ser alcançado de vários modos nos JSPs. Uma técnica popular utiliza cookies (Seção 29.7.1); outra utiliza o objeto `SessionBean` (Seção 29.7.2). Técnicas de rastreamento de sessão adicionais incluem o uso de elementos `input form` do tipo `"hidden"` e rescrita de URL. Com elementos de formulário `"hidden"`, um Web Form pode gravar dados de rastreamento de sessão em um `form` na página Web que ele retorna ao cliente em resposta a uma solicitação prévia. Quando o usuário submete o formulário na nova página Web, todos os dados de formulário, incluindo os campos `"hidden"`, são enviados ao handler de formulário no servidor Web. Com a rescrita de URL, o servidor Web incorpora as informações de rastreamento de sessão diretamente nos URLs dos hiperlinks em que o usuário clica para enviar solicitações subsequentes ao servidor Web.

29.7.1 Cookies

Cookies fornecem aos desenvolvedores Web uma ferramenta para personalizar páginas Web. Um cookie é um dado, em geral, armazenado em um arquivo de texto no computador do usuário. Um cookie mantém informações sobre o cliente durante e entre sessões de navegador. Na primeira vez que um usuário visita o site Web, o computador do usuário poderia receber um cookie; esse cookie é então reativado toda vez que o usuário revisita esse site. O objetivo é criar um registro anônimo que contém os dados utilizados para personalizar as futuras visitas do usuário ao site. Por exemplo, cookies em um aplicativo de compra poderiam armazenar identificadores únicos para usuários. Quando um usuário adiciona itens a um carrinho de compras on-line ou realiza outra tarefa resultante em uma solicitação para o servidor Web, o servidor recebe um cookie do cliente contendo o identificador único do usuário. O servidor utiliza então o identificador único para localizar o carrinho de compras e executar qualquer processamento necessário.

Além de identificar usuários, cookies também podem indicar preferências de compra dos clientes. Quando um servidor Web recebe uma solicitação de um cliente, o servidor pode examinar o(s) cookie(s) que ele enviou ao cliente durante a comunicação anterior, identificar as preferências do cliente e exibir imediatamente produtos que interessam ao cliente.

Cada interação baseada em HTTP entre um cliente e um servidor inclui um cabeçalho contendo as informações sobre a solicitação (quando a comunicação é do cliente para o servidor) ou sobre a resposta (quando a comunicação é do servidor para o cliente). Quando uma página recebe uma solicitação, o cabeçalho inclui informações como o tipo de solicitação (por exemplo, `GET` ou `POST`) e quaisquer cookies que foram enviados anteriormente a partir do servidor a ser armazenado na máquina do cliente. Quando o servidor formula sua resposta, as informações de cabeçalho contêm todos os cookies que o servidor quer armazenar no computador do cliente e outras informações, como o tipo MIME da resposta.

A **data de expiração** de um cookie determina quanto tempo o cookie permanece no computador do cliente. Se você não configurar uma data de expiração para um cookie, o navegador Web manterá o cookie pela duração da sessão de navegação. Do contrário, o navegador Web manterá o cookie até a data de expiração ocorrer. Quando o navegador solicita um recurso de um servidor Web, os cookies anteriormente enviados ao cliente por esse servidor Web são retornados ao servidor Web como parte da solicitação formulada pelo navegador. Cookies são excluídos quando expiram.

Dica de portabilidade 29.1

Os clientes podem desativar cookies nos navegadores Web para mais privacidade. Quando esses clientes utilizam aplicativos Web que dependem de cookies para manter informações de estado, os aplicativos não executarão corretamente.

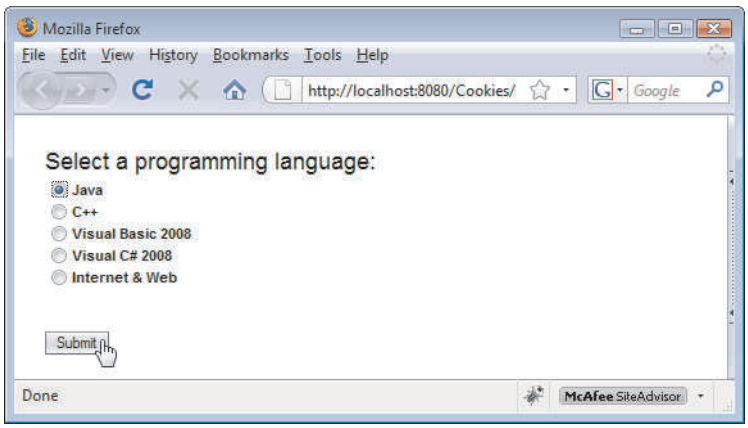
Utilizando cookies para fornecer recomendações sobre livros

O próximo aplicativo Web mostra como utilizar cookies. O exemplo contém duas páginas. Na página `Options.jsp` (figuras 29.19 e 29.21), os usuários selecionam uma linguagem de programação favorita a partir de um grupo de botões de opção e submetem o formulário ao servidor Web para processamento. O servidor Web responde criando um cookie que armazena a linguagem selecionada e o número ISBN de um livro recomendado sobre esse tema. O servidor exibe então diferentes componentes no navegador que permitem ao usuário visualizar as opções e selecionar outra linguagem de programação favorita ou visualizar a página `Recommendations.jsp` no nosso aplicativo (Figuras 29.22–29.23), que lista livros recomendados que pertencem à(s) linguagem(ns) de programação que o usuário selecionou. Como iremos ocultar e mostrar programaticamente os componentes no arquivo `Options.jsp`, cada componente na página requer um atributo de vinculação. Quando o usuário clica no hiperlink para visualizar os livros recomendados, os cookies anteriormente armazenados no cliente são enviados ao servidor, lidos pelo aplicativo e utilizados para formar a lista de livros recomendados.

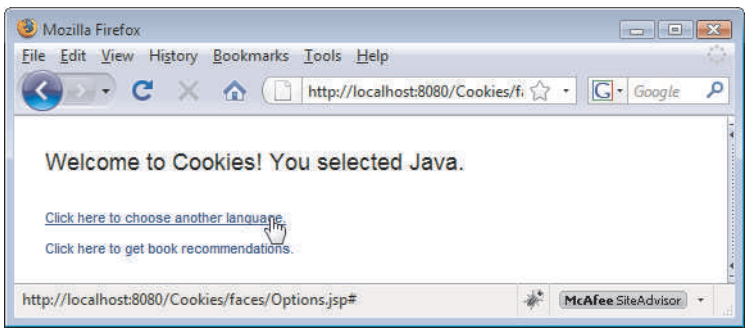
O arquivo `Options.jsp` na Figura 29.19 contém um **Radio Button Group** (linhas 24–28) com as opções **Java**, **C++**, **Visual Basic 2008**, **Visual C# 2008** e **Internet & Web**. Lembre-se de que você pode configurar a exibição e o valor de `String`s dos botões de opção clicando com o botão direito do mouse no **Radio Button Group** e selecionando **Configure Default Options...**. O código para essas opções é mostrado nas linhas 22–34 da Figura 29.21. O usuário seleciona uma linguagem de programação clicando em um botão de opção e então pressionando **Submit** para enviar a seleção ao servidor. Isso cria uma solicitação `POST` HTTP para o aplicativo Web no servidor, que obtém a seleção do usuário, cria um cookie que contém a seleção e o adiciona ao cabeçalho de resposta HTTP que é enviado ao cliente como parte da resposta. O navegador armazena então o cookie no computador cliente.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Figura 29.19: Options.jsp -->
4  <!-- Arquivo JSP que permite ao usuário selecionar uma linguagem de programação -->
5  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
6     xmlns:h="http://java.sun.com/jsf/html"
7     xmlns:jsp="http://java.sun.com/JSP/Page"
8     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
9     <jsp:directive.page contentType="text/html;charset=UTF-8"
10       pageEncoding="UTF-8"/>
11    <f:view>
12       <webuijsf:page id="page1">
13          <webuijsf:html id="html1">
14             <webuijsf:head id="head1">
15                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
16             </webuijsf:head>
17             <webuijsf:body id="body1" style="-rave-layout: grid">
18                <webuijsf:form id="form1">
19                   <webuijsf:staticText
20                      binding="#{Options.instructionText}"
21                      id="instructionText" style="font-size: 18px;
22                         left: 24px; top: 24px; position: absolute"
23                      text="Select a programming language:"/>
24                   <webuijsf:radioButtonGroup
25                      binding="#{Options.languageRadioButtonGroup}"
26                      id="languageRadioButtonGroup" items="#{Options.
27                         languageRadioButtonGroupDefaultOptions.options}"
28                      style="left: 24px; top: 48px; position: absolute"/>
29                   <webuijsf:button
30                      actionExpression="#{Options.submitButton_action}"
31                      binding="#{Options.submitButton}" id="submitButton"
32                      style="left: 23px; top: 168px; position: absolute"
33                      text="Submit"/>
34                   <webuijsf:staticText binding="#{Options.responseText}"
35                      id="responseText" rendered="false"
36                      style="font-size: 18px; left: 24px; top: 24px;
37                         position: absolute"/>
38                   <webuijsf:hyperlink
39                      actionExpression="#{Options.languagesLink_action}"
40                      binding="#{Options.languagesLink}" id="languagesLink"
41                      rendered="false" style="left: 24px; top: 72px;
42                         position: absolute"
43                      text="Click here to choose another language."/>
44                   <webuijsf:hyperlink
45                      binding="#{Options.recommendationsLink}"
46                      id="recommendationsLink" rendered="false"
47                      style="left: 24px; top: 96px; position: absolute"
48                      text="Click here to get book recommendations."
49                      url="/faces/Recommendations.jsp"/>
50                </webuijsf:form>
51             </webuijsf:body>
52          </webuijsf:html>
53       </webuijsf:page>
54    </f:view>
55 </jsp:root>
```

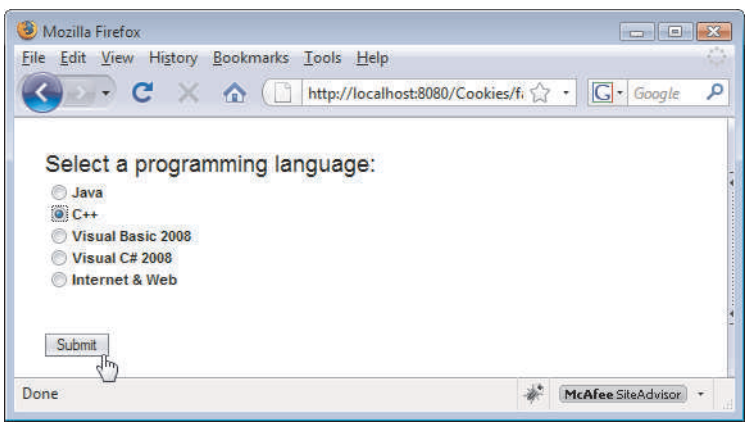
(a) O usuário seleciona uma linguagem de programação e clica em **Submit** para solicitar novamente `Options.jsp`.



(b) `Options.jsp` exibe uma mensagem de boas-vindas e fornece links que permitem ao usuário selecionar outras linguagens ou visualizar recomendações sobre o livro. O usuário escolhe selecionar outra linguagem, que solicita novamente `Options.jsp`.



(c) O usuário seleciona uma linguagem de programação e clica em Submit para solicitar novamente Options.jsp.



(d) `Options.jsp` exibe uma mensagem de boas-vindas e fornece links que permitem ao usuário selecionar outra linguagem ou visualizar recomendações sobre o livro. O usuário escolhe visualizar recomendações sobre o livro.

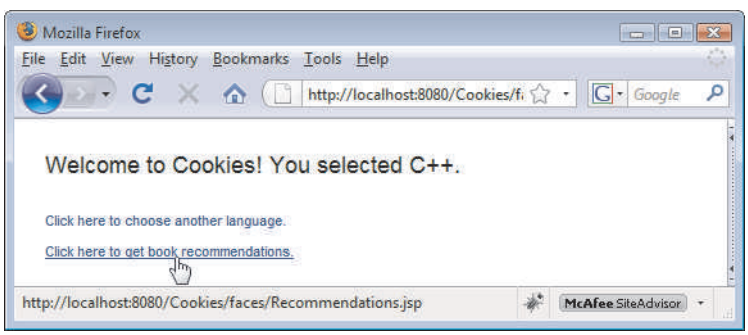


Figura 29.19 | O arquivo JSP que permite ao usuário selecionar uma linguagem de programação.

Quando o usuário clica em **Submit**, os elementos `webuijsf:staticText`, `webuijsf:radioButtonGroup` e `webuijsf:button` utilizados para selecionar uma linguagem permanecem ocultos, e um elemento `webuijsf:staticText` e dois elementos `webuijsf:hyperlink` são exibidos. As propriedades `rendered` de um `webuijsf:staticText` e de dois `webuijsf:hyperlinks` são inicialmente configuradas como `false` (linhas 35, 41 e 46). Isso indica que esses componentes não são visíveis quando a página carrega, uma vez que queremos que primeiro o usuário visualize a página para incluir somente os componentes para selecionar uma linguagem de programação e submeter a seleção.

O primeiro hiperlink (linhas 38–43) solicita essa página e o segundo (linhas 44–49) solicita `Recommendations.jsp`. A propriedade `url` não é configurada para o primeiro link; discutimos isso mais adiante. A propriedade `url` do segundo link é configurada como `/faces/Recommendations.jsp`. Lembre-se de que no início do capítulo configuramos uma propriedade `url` para um site Web remoto (`http://www.deitel.com`). Para configurar essa propriedade para uma página dentro do aplicativo em uso, você pode clicar no botão de reticências (...) ao lado da propriedade `url` na janela **Properties** a fim de abrir uma caixa de diálogo que contém uma lista das páginas do aplicativo, e então selecionar uma página existente como o destino do link.

Adicionando e vinculando a uma nova página Web

Para configurar a propriedade `url` como uma página de destino (isto é, `Recommendations.jsp`) no aplicativo em uso, a página de destino já deve existir. Para criar `Recommendations.jsp`, clique com o botão direito do mouse no nó **Web Pages** na janela **Projects** e selecione **New> Visual Web JSF Page...** no menu que aparece. Na caixa de diálogo **New Visual Web JSF Page**, mude o nome da página para `Recommendations` e clique em **Finish** para criar os arquivos `Recommendations.jsp` e `Recommendations.java`. (Discutiremos o conteúdo desses arquivos em breve.) Você agora pode selecionar `Recommendations.jsp` como o valor `url` para `recommendationsLink`. É possível ver o valor `url` na linha 49.

Ocultando e exibindo elementos de uma página e solicitando a página novamente

Quando o usuário clica em `languagesLink`, queremos solicitar `Options.jsp` novamente e exibir a lista de opções para que usuário possa fazer outra escolha. Em vez de configurar a propriedade `url` de `languagesLink`, adicionaremos um handler de ação desse componente ao bean de página. Você pode fazer isso clicando com botão direito do mouse em `languagesLink` na janela **Navigator** (enquanto no modo **Design**) e selecionando **Edit action Event Handler**. O handler de ação permitira exibir e ocultar componentes da página sem redirecionar o usuário a outra página. Especificar um `url` de destino sobrescreveria o handler de ação do componente e redirecionaria o usuário à página especificada. Nesse caso é importante não configurar a propriedade `url` porque queremos ocultar alguns elementos da página e exibir outros. Como utilizamos o `languagesLink` para recarregar a página atual, simplesmente retornamos `null` a partir do handler de ação, o que faz `Options.jsp` recarregar.

Adicionando um handler de ação a um Hyperlink e redirecionando a outra página

Se precisar adicionar um handler de ação a um hiperlink que também deve direcionar o usuário a outra página, você precisará adicionar uma regra ao arquivo **Page Navigation** (Figura 29.20). [*Nota*: Isso não é necessário para o exemplo atual.] Para editar esse arquivo, clique com o botão direito do mouse em qualquer lugar no designer visual e escolha **Page Navigation**. Clique no ícone de sinal de adição (⊞) para `Options.jsp` no designer de navegação para exibir seus componentes que poderiam fazer a página solicitar outra página. Localize o link cuja regra de navegação você gostaria de configurar (por exemplo, `recommendationsLink`) e arraste-o para a página de destino (por exemplo, `Recommendations.jsp`). Agora o link pode direcionar o usuário a uma nova página (`Recommendations.jsp`) e você também pode inserir o código no handler de ação do componente **Hyperlink** que será executado quando o usuário clicar no link. Editar o arquivo **Page Navigation** também é útil se você quiser que os elementos de ação que não podem especificar uma propriedade `url`, como botões, direcionem os usuários para outra página.

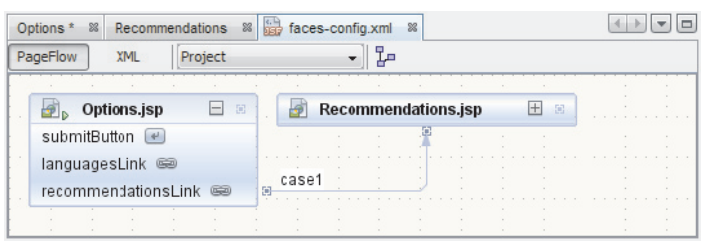


Figura 29.20 | Editando o arquivo **Page Navigation**.

Arquivo de bean de página para `Options.jsp`

A Figura 29.21 contém o código que grava um cookie na máquina cliente quando o usuário seleciona uma linguagem de programação. O arquivo também determina quais componentes aparecem na página, exibindo os componentes para escolher uma linguagem ou os hiperlinks para navegar pelo aplicativo, dependendo das ações do usuário.

```
1  // Figura 29.21: Options.java
2  // O bean de página que armazena a seleção de idioma do usuário
3  // como um cookie no cliente.
4  package cookies;
5
6  import com.sun.rave.web.ui.appbase.AbstractPageBean;
```

```
 7  import com.sun.webui.jsf.component.Button;
 8  import com.sun.webui.jsf.component.Hyperlink;
 9  import com.sun.webui.jsf.component.RadioButtonGroup;
10  import com.sun.webui.jsf.component.StaticText;
11  import com.sun.webui.jsf.model.SingleSelectOptionsList;
12  import java.util.HashMap;
13  import javax.faces.FacesException;
14  import javax.servlet.http.Cookie;
15  import javax.servlet.http.HttpServletResponse;
16
17  public class Options extends AbstractPageBean
18  {
19     // Definição de componente gerenciado
20     private void _init() throws Exception
21     {
22        languageRadioButtonGroupDefaultOptions.setOptions(
23           new com.sun.webui.jsf.model.Option[]
24           {
25              new com.sun.webui.jsf.model.Option( "Java", "Java" ),
26              new com.sun.webui.jsf.model.Option( "C++", "C++" ),
27              new com.sun.webui.jsf.model.Option( "Visual-Basic-2008",
28                 "Visual Basic 2008" ),
29              new com.sun.webui.jsf.model.Option( "Visual-C#-2008",
30                 "Visual C# 2008" ),
31              new com.sun.webui.jsf.model.Option( "Internet-&-Web",
32                 "Internet & Web" )
33           } // fim do inicializador de array
34        ); // fim da chamada para setOptions
35     } // fim do método _init
36
37     private SingleSelectOptionsList languageRadioButtonGroupDefaultOptions =
38        new SingleSelectOptionsList();
39
40     // Para economizar espaço, omitimos o código nas linhas 40–121. O código-fonte
41     // completo é fornecido com os exemplos deste capítulo.
42
122    private HashMap<String, String> books = new HashMap<String, String>();
123
124    // Constrói uma nova instância do bean de página e inicializa as propriedades
125    // que mapeiam os idiomas para números ISBN dos livros recomendados.
126    public Options()
127    {
128       // inicializa o objeto HashMap dos valores a ser armazenados como cookies.
129       books.put( "Java", "0132222205" );
130       books.put( "C++", "0136152503" );
131       books.put( "Visual-Basic-2008", "013605305X" );
132       books.put( "Visual-C#-2008", "013605322X" );
133       books.put( "Internet-&-Web", "0131752421" );
134    } // fim do construtor Options
135
136    // Para economizar espaço, omitimos o código nas linhas 136–185. O código-fonte
137    // completo é fornecido com os exemplos deste capítulo.
138
186    // Handler de ação do botão Submit. Verifica se um idioma
187    // foi selecionado e, nesse caso, cria um cookie para esse idioma e
188    // configura responseText para indicar o idioma selecionado.
189    public String submitButton_action()
190    {
191       String message = "Welcome to Cookies! You ";
192
193       // se o usuário fez uma seleção
194       if ( languageRadioButtonGroup.getSelected() != null )
195       {
196          String language =
197             languageRadioButtonGroup.getSelected().toString();
198          message += "selected " + language.replace( '-', ' ' ) + ".";
199
200          // obtém o número de ISBN do livro para o idioma dado
201          String ISBN = books.get( language );
```

```
202
203             // cria o cookie utilizando um par de nome/valor do idioma ISBN
204             Cookie cookie = new Cookie( language, ISBN );
205
206             // adiciona o cookie ao cabeçalho de resposta para inseri-lo na máquina do usuário
207             HttpServletResponse response =
208                ( HttpServletResponse ) getExternalContext().getResponse();
209             response.addCookie( cookie );
210          } // fim do if
211          else
212          {
213             message += "did not select a language.";
214          } // fim de else
215
216          responseText.setValue( message );
217          languageRadioButtonGroup.setRendered( false ); // oculta
218          instructionText.setRendered( false ); // oculta
219          submitButton.setRendered( false ); // oculta
220          responseText.setRendered( true ); // exibe
221          languagesLink.setRendered( true ); // exibe
222          recommendationsLink.setRendered( true ); // exibe
223          return null; // reloads the page
224       } // fim do método submitButton_action
225
226       // reexibe os componentes para selecionar um idioma
227       public String languagesLink_action()
228       {
229          responseText.setRendered( false ); // oculta
230          languagesLink.setRendered( false ); // oculta
231          recommendationsLink.setRendered( false ); // oculta
232          languageRadioButtonGroup.setRendered( true ); // exibe
233          instructionText.setRendered( true ); // exibe
234          submitButton.setRendered( true ); // exibe
235          return null; // reloads the page
236       } // fim do método languagesLink_action
237    } // fim da classe Options
```

Figura 29.21 | O bean de página que armazena a seleção de idioma do usuário em um cookie cliente.

Como mencionado anteriormente, o método `_init` trata a inicialização de componentes. Como essa página contém um objeto `RadioButtonGroup` que requer inicialização, o método `_init` (linhas 20–35) constrói um array de objetos `Option` a ser exibidos pelos botões.

As linhas 129–133 no construtor inicializam um objeto `HashMap` (definido na linha 122) — uma estrutura de dados que armazena pares chave/valor. Nesse caso, as chaves e os valores são `String`s. O aplicativo utiliza a chave para armazenar e recuperar o valor associado no objeto `HashMap`. Nesse exemplo, as chaves contêm os nomes da linguagem de programação, e os valores contêm os números de ISBN dos livros recomendados. A classe `HashMap` fornece o método `put`, que recebe como argumentos uma chave e um valor. Um valor que é adicionado via método `put` é inserido no `HashMap` em uma localização determinada pela chave. O valor de uma entrada `HashMap` específica pode ser determinado invocando o método `get` no objeto `HashMap` com a chave desse valor como um argumento.

Observe que o NetBeans pode importar automaticamente quaisquer pacotes ausentes que seu arquivo Java precise. Por exemplo, depois de adicionar o objeto `HashMap` a `Options.java`, você pode clicar com o botão direito do mouse na janela do editor Java e selecionar **Fix Imports** para importar `java.util.HashMap`. Essa opção também pode remover declarações `import` não utilizadas.

Clicar em **Submit** invoca o handler de evento `submitButton_action` (linhas 189–224), que exibe uma mensagem indicando a linguagem selecionada no elemento `responseText` e adiciona um novo cookie à resposta. Se uma linguagem foi selecionada (linha 194), o item selecionado é recuperado (linhas 196–197). A linha 198 adiciona a linguagem selecionada à string `message`.

A linha 201 recupera o ISBN do idioma selecionado a partir do objeto `HashMap` `books`. A linha 204 cria então um novo objeto `Cookie` (no pacote `javax.servlet.http`), utilizando o idioma selecionado como o nome do cookie e um ISBN correspondente como o valor do cookie. Esse cookie é adicionado ao cabeçalho de resposta HTTP nas linhas 207–209. Um objeto da classe `HttpServletResponse` (do pacote `javax.servlet.http`) representa a resposta. Esse objeto pode ser acessado invocando o método herdado `getExternalContext` no bean de página, então invocando `getResponse` no objeto resultante. Se uma linguagem não foi selecionada, a linha 213 configura a mensagem de resultados para indicar que nenhuma seleção foi feita. [*Nota*: Nomes de cookies não podem conter espaços em branco. Por essa razão, hifenizamos os nomes de múltiplas palavras que representam os nomes de cookies (isto é, os nomes da linguagem de programação nesse exemplo) nas linhas 27, 29, 31 e 131–133. Utilizamos o método `replace` `String` para substituir os hífens por espaços quando exibimos o nome da linguagem.]

As linhas 216–222 controlam a aparência da página depois que o usuário clica em **Submit**. A linha 216 configura o `responseText` para exibir a `String` `message`. Como o usuário acabou de submeter uma seleção de linguagem, os componentes utilizados para coletar a seleção permanecem ocultos (linhas 217–219), e `responseText` e os links utilizados para navegar pelo aplicativo são exibidos (linhas 220–222). O handler de ação retorna `null` na linha 223, que recarrega `Options.jsp`.

As linhas 227–236 contêm o handler de evento do `languagesLink`. Quando o usuário clica nesse link, `responseText` e os dois links permanecem ocultos (linhas 229–231), e os componentes que permitem ao usuário selecionar uma linguagem são exibidos novamente (linhas 232–234). O método retorna `null` na linha 235, fazendo `Options.jsp` recarregar.

Exibindo recomendações sobre livros com base nos valores de cookie

Depois de clicar em **Submit**, o usuário pode solicitar uma recomendação sobre livros. O hiperlink para recomendações sobre livros direciona o usuário a `Recommendations.jsp` (Figura 29.22) para exibir as recomendações com base nas seleções de idioma do usuário.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!-- Figura 29.22 Recommendations.jsp -->
4  <!-- Exibe recomendações sobre livros utilizando cookies -->
5  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
6     xmlns:h="http://java.sun.com/jsf/html"
7     xmlns:jsp="http://java.sun.com/JSP/Page"
8     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
9     <jsp:directive.page contentType="text/html;charset=UTF-8"
10       pageEncoding="UTF-8"/>
11    <f:view>
12       <webuijsf:page id="page1">
13          <webuijsf:html id="html1">
14             <webuijsf:head id="head1">
15                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
16             </webuijsf:head>
17             <webuijsf:body id="body1" style="-rave-layout: grid">
18                <webuijsf:form id="form1">
19                   <webuijsf:label for="recommendationsTextArea"
20                      id="recommendationsLabel" style="font-size: 14px;
21                         left: 24px; top: 24px; position: absolute"
22                      text="Recommendations"/>
23                   <webuijsf:textArea
24                      binding="#{Recommendations.recommendationsTextArea}"
25                      columns="60" id="recommendationsTextArea" rows="6"
26                      style="left: 24px; top: 48px; position: absolute"/>
27                   <webuijsf:hyperlink id="optionsLink"
28                      style="left: 24px; top: 168px; position: absolute"
29                      text="Click here to choose another language."
30                      url="/faces/Options.jsp"/>
31                </webuijsf:form>
32             </webuijsf:body>
33          </webuijsf:html>
34       </webuijsf:page>
35    </f:view>
36 </jsp:root>
```

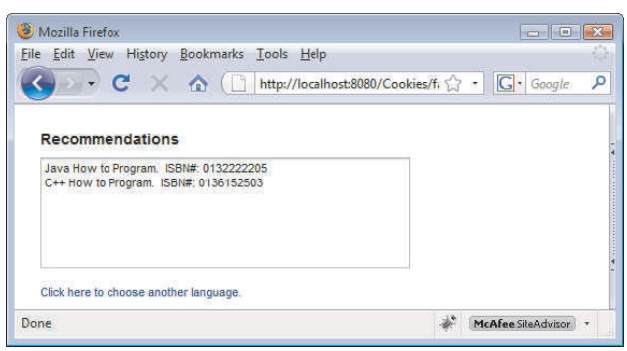


Figura 29.22 | O arquivo JSP que exibe recomendações sobre livros com base em cookies.

`Recommendations.jsp` contém um **Label** (linhas 19–22), uma **Text Area** (linhas 23–26) e um **Hyperlink** (linhas 27–30). O **Label** exibe o texto `Recommendations` na parte superior da página. Um componente **Text Area** pode exibir múltiplas linhas de texto. A **Text Area** nesse exemplo exibe as recomendações criadas pelo bean de página `Recommendations.java` (Figura 29.23), ou o texto `"No Recommendations. Please select a language."` O **Hyperlink** permite ao usuário retornar a `Options.jsp` (especificado pelo link `url`) para selecionar linguagens adicionais.

Bean de página que cria recomendações de livros a partir de cookies

Em `Recommendations.java` (Figura 29.23), o método `prerender` (linhas 65–96) recupera os cookies do cliente, utilizando o método `getCookies` do objeto de solicitação (linhas 68–70). Um objeto da classe `HttpServletRequest` (do pacote `javax.servlet.http`) representa a solicitação. Esse objeto pode ser obtido invocando o método `getExternalContext` no bean de página, e depois invocando `getRequest` no objeto resultante. A chamada a `getCookies` retorna um array dos cookies anteriormente gravados no cliente. Cookies só podem ser lidos por um aplicativo se eles foram criados por um servidor no domínio no qual o aplicativo está em execução — por razões de segurança, um servidor Web não pode acessar cookies criados por servidores em outros domínios. Por exemplo, um cookie criado por um servidor Web no domínio `deitel.com` não pode ser lido por um servidor Web em nenhum outro domínio.

```
1  // Figura 29.23: Recommendations.java
2  // Exibe recomendações sobre livros com base em cookies que
3  // contêm as linguagens de programação selecionadas pelo usuário.
4  package cookies;
5
6  import com.sun.rave.web.ui.appbase.AbstractPageBean;
7  import com.sun.webui.jsf.component.TextArea;
8  import javax.faces.FacesException;
9  import javax.servlet.http.Cookie;
10 import javax.servlet.http.HttpServletRequest;
11
12 public class Recommendations extends AbstractPageBean
13 {
14    // Para economizar espaço, omitimos o código nas linhas 14-62. O código-fonte
15    // completo é fornecido com os exemplos deste capítulo.
16
63    // processa e exibe as seleções do usuário
64    @Override
65    public void prerender()
66    {
67       // recupera os cookies
68       HttpServletRequest request =
69          ( HttpServletRequest ) getExternalContext().getRequest();
70       Cookie[] cookies = request.getCookies();
71
72       // se houver cookies, armazena os
73       // livros e números de ISBN correspondentes em uma String
74       String recommendations = "";
75
76       if ( cookies.length > 1 )
77       {
78          for ( int i = 0; i < cookies.length - 1; i++ )
79          {
80             String language = cookies[ i ].getName().replace( '-', ' ' );
81
82             // ignora o cookie criado quando o usuário retorna a
83             // Options.jsp a partir de Recommendations.jsp
84             if ( !language.equals( "/Recommendations.jsp" ) )
85                recommendations += language + " How to Program.  ISBN#: " +
86                   cookies[ i ].getValue() + "\n";
87          } // for final
88       } // fim do if
89       else
90       {
91          recommendations =
```

```
92                "No recommendations. Please select a language.";
93          } // fim de else
94
95          recommendationsTextArea.setText( recommendations );
96       } // fim do método prerender
97
98       // Para economizar espaço, omitimos o código nas linhas 98-116. O código-fonte
99       // completo é fornecido com os exemplos deste capítulo.
117   } // fim da classe Recommendations
```

Figura 29.23 | O bean de página que exibe recomendações sobre livros com base nos cookies que armazenam as linguagens selecionadas pelo usuário.

A linha 76 determina se existem pelo menos dois cookies. As linhas 78–87 adicionam as informações contidas no(s) cookie(s) à string `recommendations`, desde que o nome do cookie não seja `"/Recommendations.jsp"` — um cookie com esse nome é adicionado quando o usuário vai de `Recommendations.jsp` a `Options.jsp`. O loop recupera o nome e o valor de cada cookie, utilizando a variável de controle para determinar o valor atual no array de cookies. Se nenhuma linguagem foi selecionada, as linhas 91–92 atribuem a `recommendations` uma mensagem que instrui o usuário a selecionar uma linguagem. A linha 95 configura `recommendationsTextArea` para exibir a string `recommendations` resultante. Resumimos os métodos `Cookie` comumente utilizados na Figura 29.24.

Método	Descrição
`getDomain`	Retorna uma `String` contendo o domínio do cookie (isto é, o domínio a partir do qual o cookie foi gravado). Isso determina quais servidores Web podem receber o cookie. Por padrão, os cookies são enviados para o servidor Web que originalmente enviou o cookie para o cliente. Alterar a propriedade `Domain` faz o cookie retornar a um servidor Web diferente daquele que originalmente o gravou.
`getMaxAge`	Retorna um `int` indicando por quantos segundos o cookie persistirá no navegador. Isso é –1 por padrão, significando que o cookie persistirá até que o navegador seja fechado.
`getName`	Retorna uma `String` contendo o nome do cookie.
`getPath`	Retorna uma `String` contendo o caminho para um diretório no servidor ao qual o cookie é aplicado. Cookies podem ser direcionados a diretórios específicos no servidor Web. Por padrão, um cookie é retornado apenas para aplicativos que operam no mesmo diretório que o aplicativo que enviou o cookie ou um subdiretório desse diretório. Alterar a propriedade `Path` faz o cookie retornar a um diretório além daquele em que foi originalmente gravado.
`getSecure`	Retorna um valor `boolean` indicando se o cookie deve ser transmitido por meio de um protocolo seguro. O valor `true` faz com que um protocolo seguro seja utilizado.
`getValue`	Retorna uma `String` contendo o valor do cookie.

Figura 29.24 | Métodos `javax.servlet.http.Cookie`.

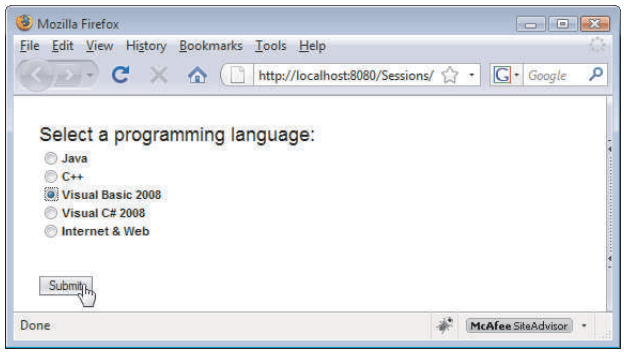
29.7.2 Rastreamento de sessão com beans de sessão

Você também pode realizar o rastreamento de sessão com a subclasse de **`AbstractSessionBean`** que é fornecida em cada aplicativo Web. Por padrão, a subclasse é nomeada `SessionBean1`. Quando um usuário solicita uma página no aplicativo Web, um objeto `SessionBean1` é criado no servidor. As propriedades desse objeto podem ser acessadas por toda a sessão de navegador invocando o método `getSessionBean1` de bean de página. Para demonstrar as técnicas de rastreamento de sessão utilizando o `SessionBean1`, modificamos os arquivos de bean de página nas figuras 29.21 e 29.23 para que elas utilizem o objeto `SessionBean1` a fim de armazenar as seleções de linguagem do usuário. Começamos com o arquivo `Options.jsp` atualizado (Figura 29.25). A Figura 29.27 apresenta o arquivo `SessionBean1.java`, e a Figura 29.28 apresenta o arquivo de bean de página modificado para `Options.jsp`.

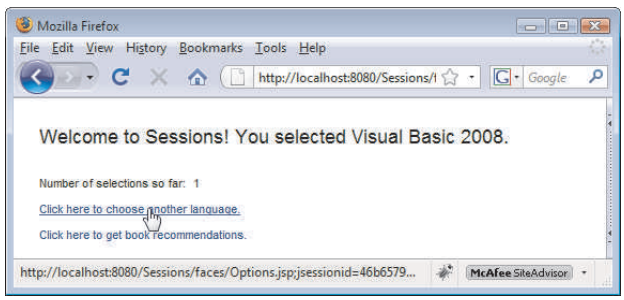
O arquivo `Options.jsp` na Figura 29.25 é semelhante àquele da Figura 29.19. As linhas 38–47 definem dois elementos `webuijsf:staticText` adicionais. O primeiro exibe o texto `"Number of selections so far:"`. O segundo atributo `text` é vinculado à propriedade `numberOfSelections` no objeto `SessionBean1` (linha 47). Discutimos como vincular o atributo `text` a uma propriedade de bean de sessão mais adiante.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Figura 29.25 Options.jsp -->
<!-- Arquivo JSP que permite ao usuário selecionar uma linguagem de programação -->
<jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:jsp="http://java.sun.com/JSP/Page"
   xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
   <jsp:directive.page contentType="text/html;charset=UTF-8"
      pageEncoding="UTF-8"/>
   <f:view>
      <webuijsf:page id="page1">
         <webuijsf:html id="html1">
            <webuijsf:head id="head1">
               <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
            </webuijsf:head>
            <webuijsf:body id="body1" style="-rave-layout: grid">
               <webuijsf:form id="form1">
                  <webuijsf:staticText
                     binding="#{Options.instructionText}"
                     id="instructionText" style="font-size: 18px;
                        left: 24px; top: 24px; position: absolute"
                     text="Select a programming language:"/>
                  <webuijsf:radioButtonGroup
                     binding="#{Options.languageRadioButtonGroup}"
                     id="languageRadioButtonGroup" items="#{Options.
                        languageRadioButtonGroupDefaultOptions.options}"
                     style="left: 24px; top: 48px; position: absolute"/>
                  <webuijsf:button
                     actionExpression="#{Options.submitButton_action}"
                     binding="#{Options.submitButton}" id="submitButton"
                     style="left: 23px; top: 168px; position: absolute"
                     text="Submit"/>
                  <webuijsf:staticText binding="#{Options.responseText}"
                     id="responseText" rendered="false"
                     style="font-size: 18px; left: 24px; top: 24px;
                        position: absolute"/>
                  <webuijsf:staticText
                     binding="#{Options.selectionsText}"
                     id="selectionsText" rendered="false"
                     style="left: 24px; top: 72px; position: absolute"
                     text="Number of selections so far:"/>
                  <webuijsf:staticText
                     binding="#{Options.selectionsValueText}"
                     id="selectionsValueText" rendered="false"
                     style="left: 168px; top: 72px; position: absolute"
                     text="#{SessionBean1.numberOfSelections}"/>
                  <webuijsf:hyperlink
                     actionExpression="#{Options.languagesLink_action}"
                     binding="#{Options.languagesLink}" id="languagesLink"
                     rendered="false" style="left: 24px; top: 96px;
                        position: absolute"
                     text="Click here to choose another language."/>
                  <webuijsf:hyperlink
                     binding="#{Options.recommendationsLink}"
                     id="recommendationsLink" rendered="false"
                     style="left: 24px; top: 120px; position: absolute"
                     text="Click here to get book recommendations."
                     url="/faces/Recommendations.jsp"/>
               </webuijsf:form>
            </webuijsf:body>
         </webuijsf:html>
      </webuijsf:page>
   </f:view>
</jsp:root>
```

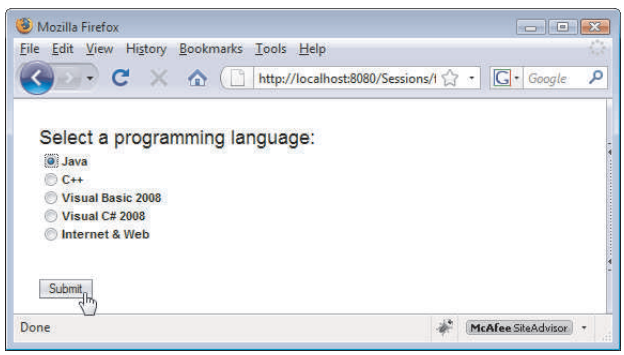
(a) O usuário seleciona a linguagem de programação e clica em **Submit** para solicitar novamente `Options.jsp`.



(b) `Options.jsp` exibe uma mensagem de boas-vindas e fornece links que permitem ao usuário selecionar outra linguagem ou visualizar recomendações sobre o livro. O usuário escolhe selecionar outra linguagem, que solicita novamente `Options.jsp`.



(c) O usuário seleciona uma linguagem de programação e clica em **Submit** para solicitar novamente `Options.jsp`.



(d) `Options.jsp` exibe uma mensagem de boas-vindas e fornece links que permitem ao usuário selecionar outra linguagem ou visualizar recomendações sobre o livro. O usuário escolhe visualizar recomendações de livro.

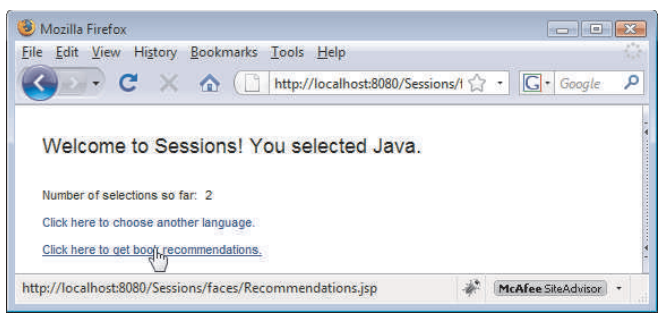


Figura 29.25 | O arquivo JSP que permite ao usuário selecionar uma linguagem de programação.

Adicionando propriedades ao `SessionBean`

Esse exemplo utiliza o rastreamento de sessão para armazenar as linguagens selecionadas pelo usuário e o número de seleções que o usuário faz. Para armazenar informações de sessão, adicionamos propriedades à classe `SessionBean1`.

Comece clicando duas vezes no nó `SessionBean1` na janela **Navigator** para abrir `SessionBean1.java` no editor. Declare uma variável de instância `numberOfSelections` do tipo `int` para armazenar o número de seleções que o usuário faz. Em seguida, clique com o botão direito do mouse na variável no editor e selecione **Refactor> Encapsulate Fields...**. Na caixa de diálogo que aparece, mantenha as opções padrão e clique no botão **Refactor**. Isso cria métodos *get* e *set* para a variável de instância `numberOfSelections` na parte inferior do arquivo de código-fonte. Em conjunto, a variável de instância e seus métodos *get* e *set* representam a propriedade `numberOfSelections` do bean.

Como você verá, manipulamos a propriedade `numberOfSelections` no arquivo de bean de página para rastrear o número de linguagens que o usuário seleciona. Você pode ver na Figura 29.25 (b) e (d) que exibimos esse valor na página toda vez que o usuário faz outra seleção. Para exibir o valor no elemento `selectionsValueText`, mude para o modo **Design**, clique com o botão direito do mouse no ele-

mento na janela **Navigator** ou no editor visual e selecione **Bind to Data...**. Na caixa de diálogo **Bind to Data** (Figura 29.26), clique na guia **Bind to an Object**, selecione a propriedade `numberOfSelections` no nó `SessionBean1` e clique em **OK**. O elemento `selectionsValueText` agora está vinculado ao valor da propriedade `numberOfSelections` de `SessionBean1`. Quando o valor da propriedade muda, o texto na página muda correspondentemente — não é necessário configurar programaticamente o texto no arquivo do bean de página.

Agora que adicionamos uma propriedade à classe `SessionBean1` para armazenar o número de seleções, vamos adicionar outra propriedade para armazenar as seleções em si. Queremos armazenar seleções como pares de chave/valor da linguagem selecionada e o número de ISBN de um livro relacionado, assim como as seleções foram armazenadas utilizando cookies. Para fazer isso, adicione uma variável de instância `HashMap` nomeada `selections` à classe `SessionBean1`, refatore então o código para criar métodos *get* e *set* como você fez para `numberOfSelections`. As duas propriedades que adicionamos são mostradas no arquivo `SessionBean1.java` (Figura 29.27).

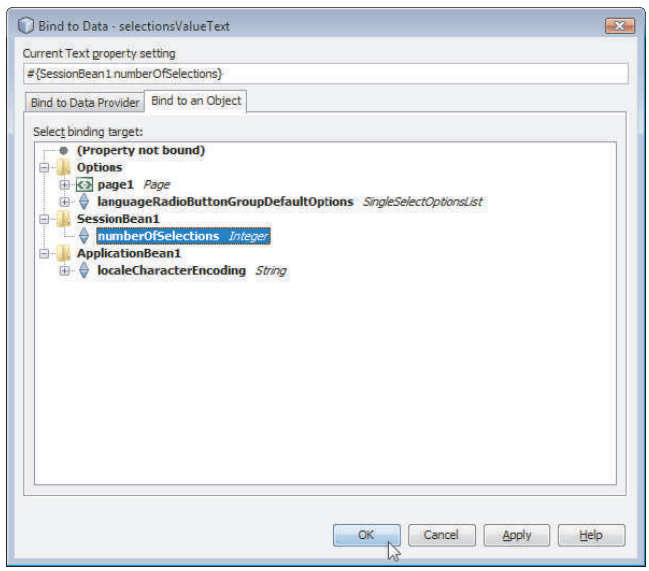


Figura 29.26 | Caixa de diálogo **Bind to Data**.

```
1  // Figura 29.27: SessionBean1.java
2  // Arquivo SessionBean para armazenar seleções de idioma.
3  package sessions;
4
5  import com.sun.rave.web.ui.appbase.AbstractSessionBean;
6  import java.util.HashMap;
7  import javax.faces.FacesException;
8
9  public class SessionBean1 extends AbstractSessionBean
10 {
11    private int numberOfSelections; // armazena o número de linguagens selecionadas
12
13    // armazena HashMap que contém as seleções do usuário
14    private HashMap<String, String> selections =
15       new HashMap<String, String>();
16
17    // Para economizar espaço, omitimos o código nas linhas 17-69. O código-fonte
18    // completo é fornecido com os exemplos deste capítulo.
19
70    public int getNumberOfSelections()
71    {
72       return numberOfSelections;
73    } // fim do método getNumberOfSelections
74
75    public void setNumberOfSelections( int numberOfSelections )
76    {
77       this.numberOfSelections = numberOfSelections;
78    } // fim do método setNumberOfSelections
79
80    public HashMap<String, String> getSelectedLanguages()
81    {
82       return selections;
```

```
83       } // fim do método getSelectedLanguages
84
85       public void setSelectedLanguages( HashMap<String, String> selections )
86       {
87          this.selections = selections;
88       } // fim do método setSelectedLanguages
89    } // fim da classe SessionBean1
```

Figura 29.27 | Arquivo `SessionBean` para armazenar seleções de idioma.

A linha 11 declara a variável de instância `numberOfSelections`, e as linhas 70–73 e 75–78 definem os métodos *get* e *set*, respectivamente, para completar a propriedade `numberOfSelections`. As linhas 14–15 definem o objeto `HashMap selections` que armazenará as seleções de usuário. As linhas 80–83 e 85–88 são métodos *get* e *set* para essa propriedade. Lembre-se de que o IDE gerou os métodos *get* e *set* quando clicamos com o botão direito do mouse em cada variável de instância, selecionamos **Refactor > Encapsulate Fields...** e clicamos no botão **Refactor**.

Manipulando propriedades de bean de sessão em um arquivo de bean de página

O arquivo de bean de página para a página `Options.jsp` é exibido na Figura 29.28. Como a maior parte desse exemplo é idêntica ao anterior, só discutiremos os novos recursos. Como não utilizaremos cookies nesse exemplo, não será necessário hifenizar os nomes da linguagem de programação que foram anteriormente utilizados como nomes de cookies (linhas 24, 26, 28 e 151–153).

```
1    // Figura 29.28: Options.java
2    // O bean de página que armazena a seleção de idioma do usuário em um SessionBean.
3    package sessions;
4
5    import com.sun.rave.web.ui.appbase.AbstractPageBean;
6    import com.sun.webui.jsf.component.Button;
7    import com.sun.webui.jsf.component.Hyperlink;
8    import com.sun.webui.jsf.component.RadioButtonGroup;
9    import com.sun.webui.jsf.component.StaticText;
10   import com.sun.webui.jsf.model.SingleSelectOptionsList;
11   import java.util.HashMap;
12   import javax.faces.FacesException;
13
14   public class Options extends AbstractPageBean
15   {
16      // Managed Component Definition"
17      private void _init() throws Exception
18      {
19         languageRadioButtonGroupDefaultOptions.setOptions(
20            new com.sun.webui.jsf.model.Option[]
21            {
22               new com.sun.webui.jsf.model.Option( "Java", "Java" ),
23               new com.sun.webui.jsf.model.Option( "C++", "C++" ),
24               new com.sun.webui.jsf.model.Option( "Visual Basic 2008",
25                  "Visual Basic 2008" ),
26               new com.sun.webui.jsf.model.Option( "Visual C# 2008",
27                  "Visual C# 2008" ),
28               new com.sun.webui.jsf.model.Option( "Internet & Web",
29                  "Internet & Web" )
30            } // fim do inicializador de array
31         ); // fim da chamada para setOptions
32      } // fim do método _init
33
34      private SingleSelectOptionsList languageRadioButtonGroupDefaultOptions =
35         new SingleSelectOptionsList();
36
37      // Para economizar espaço, omitimos o código nas linhas 36-141. O código-fonte
38      // completo é fornecido com os exemplos deste capítulo.
39
142     private HashMap<String, String> books = new HashMap<String, String>();
143
144     // Constrói uma nova instância do bean de página e inicializa as propriedades
145     // que mapeiam os idiomas para números ISBN dos livros recomendados.
146     public Options()
```

```
147       {
148          // inicializa o objeto HashMap dos valores a serem armazenados na sessão.
149          books.put( "Java", "0132222205" );
150          books.put( "C++", "0136152503" );
151          books.put( "Visual Basic 2008", "013605305X" );
152          books.put( "Visual C# 2008", "013605322X" );
153          books.put( "Internet & Web", "0131752421" );
154       } // fim do construtor Options
155
156       // Para economizar espaço, omitimos o código nas linhas 156-205. O código-fonte
157       // completo é fornecido com os exemplos deste capítulo.
158
206       // Handler de ação do botão Submit. Verifica se um idioma
207       // foi selecionado e, nesse caso, cria uma entrada para essa linguagem e
208       // configura responseText para indicar o idioma selecionado.
209       public String submitButton_action()
210       {
211          String message = "Welcome to Sessions! You ";
212
213          // se o usuário fez uma seleção
214          if ( languageRadioButtonGroup.getSelected() != null )
215          {
216             String language =
217                languageRadioButtonGroup.getSelected().toString();
218             message += "selected " + language + ".";
219
220             // obtém o número de ISBN do livro para o idioma dado
221             String ISBN = books.get( language );
222
223             // adiciona a seleção ao objeto Properties de SessionBean
224             HashMap<String, String> selections =
225                getSessionBean1().getSelectedLanguages();
226             String result = selections.put( language, ISBN );
227
228             // incrementa numberOfSelections em SessionBean
229             // se o usuário não fez essa seleção antes
230             if ( result == null )
231             {
232                int selected = getSessionBean1().getNumberOfSelections();
233                getSessionBean1().setNumberOfSelections( ++selected );
234             } // fim do if
235          } // fim do if
236          else
237          {
238             message += "did not select a language.";
239          } // fim de else
240
241          responseText.setValue( message );
242          languageRadioButtonGroup.setRendered( false ); // oculta
243          instructionText.setRendered( false ); // oculta
244          submitButton.setRendered( false ); // oculta
245          responseText.setRendered( true ); // exibe
246          selectionsText.setRendered( true ); // exibe
247          selectionsValueText.setRendered( true ); // exibe
248          languagesLink.setRendered( true ); // exibe
249          recommendationsLink.setRendered( true ); // exibe
250          return null; // reloads the page
251       } // fim do método submitButton_action
252
253       // reexibe os componentes para selecionar um idioma
254       public String languagesLink_action()
255       {
256          responseText.setRendered( false ); // oculta
257          selectionsText.setRendered( false ); // oculta
258          selectionsValueText.setRendered( false ); // oculta
259          languagesLink.setRendered( false ); // oculta
260          recommendationsLink.setRendered( false ); // oculta
261          languageRadioButtonGroup.setRendered( true ); // exibe
262          instructionText.setRendered( true ); // exibe
```

```
263          submitButton.setRendered( true ); // exibe
264          return null; // reloads the page
265       } // fim do método languagesLink_action
266    } // fim da classe Options
```

Figura 29.28 | O bean de página que armazena seleções de idioma em uma propriedade `SessionBean`.

O handler de ação `submitButton` (linhas 209–251) armazena as seleções do usuário em `SessionBean1` e incrementa o número de seleções feitas, se necessário. As linhas 224–225 recuperam a partir de `SessionBean1` o objeto `HashMap` que contém as seleções do usuário. A linha 226 adiciona a seleção atual ao `HashMap`. O método `put` retorna o valor anteriormente associado com a nova chave, ou `null` se essa chave ainda não foi armazenada no objeto `HashMap`. Se a adição da nova propriedade retornar `null`, o usuário fez então uma nova seleção. Nesse caso, as linhas 232–233 incrementam a propriedade `numberOfSelections` de `SessionBean1`. As linhas 242–249 e o handler de ação `languaguesLink` (linhas 254–265) controlam os componentes que são exibidos, assim como nos exemplos de cookies.

Observação de engenharia de software 29.2

Uma vantagem do uso de propriedades de bean de sessão (em vez de cookies) é que elas podem armazenar qualquer tipo de objeto (não apenas `Strings`) como valores de atributo. Isso fornece maior flexibilidade e poder ao manter informações de estado do cliente.

Exibindo recomendações com base em valores de sessão

Como no exemplo de cookies, esse aplicativo fornece um link para `Recommendations.jsp`, que exibe uma lista de recomendações sobre livros com base nas seleções de idioma do usuário. Como esse JSP é idêntico à versão na Figura 29.22, mostramos apenas a saída de exemplo dessa página na Figura 29.29.

Bean de página que cria recomendações sobre livro a partir de uma propriedade `SessionBean`

A Figura 29.30 apresenta o bean de página para `Recommendations.jsp`. Mais uma vez, boa parte dele é semelhante ao bean de página utilizado no exemplo de cookies. Discutimos apenas os novos recursos.

As linhas 67–68 recuperam o objeto `HashMap` que contém as seleções do usuário a partir do bean de sessão, e a linha 69 recupera o número de seleções feitas. Se alguma seleção foi feita, as linhas 78–82 acrescentam recomendações sobre livros à string `recommendations`. A linha 78 utiliza o método `keySet` de `HashMap` para obter um `Set` das chaves no `HashMap`, e a linha 81 utiliza cada chave para obter o ISBN do livro correspondente.

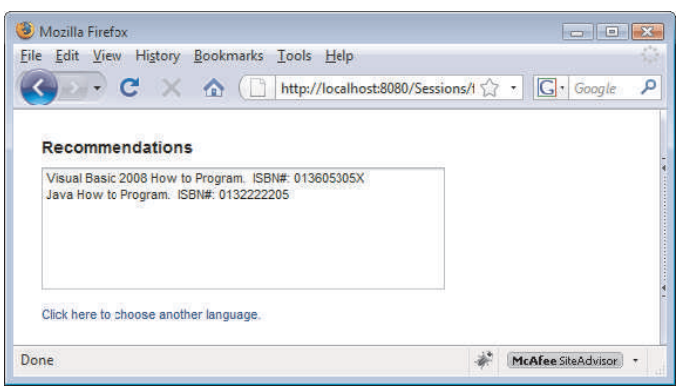


Figura 29.29 | O arquivo JSP que exibe recomendações sobre livros com base nas seleções de idioma armazenadas no escopo de sessão.

```
1  // Figura 29.30: Recommendations.java
2  // Exibe recomendações sobre livros com base em uma propriedade HashMap no
3  // bean de sessão que contém as linguagens de programação selecionadas pelo usuário.
4  package sessions;
5
6  import com.sun.rave.web.ui.appbase.AbstractPageBean;
7  import com.sun.webui.jsf.component.TextArea;
8  import java.util.HashMap;
9  import javax.faces.FacesException;
10
11 public class Recommendations extends AbstractPageBean
12 {
```

```
13       // Para economizar espaço, omitimos o código nas linhas 13-61. O código-fonte
14       // completo é fornecido com os exemplos deste capítulo.
15
62       // processa e exibe as seleções do usuário
63       @Override
64       public void prerender()
65       {
66          // recupera as seleções do usuário e o número de seleções feitas
67          HashMap<String, String> languages =
68             getSessionBean1().getSelectedLanguages();
69          int numberSelected = getSessionBean1().getNumberOfSelections();
70
71          // se houver seleções, armazena os correspondentes
72          // livros e números de ISBN correspondentes em uma String
73          String recommendations = "";
74
75          // se pelo menos uma seleção foi feita
76          if ( numberSelected > 0 )
77          {
78             for ( String language : languages.keySet() )
79             {
80                recommendations += language + " How to Program.  ISBN#: " +
81                   languages.get( language ) + "\n";
82             } // for final
83          } // fim do if
84          else
85          {
86             recommendations =
87                "No recommendations. Please select a language.";
88          } // fim de else
89
90          recommendationsTextArea.setText( recommendations );
91       } // fim do método prerender
92
93       // Para economizar espaço, omitimos o código nas linhas 92-111. O código-fonte
94       // completo é fornecido com os exemplos deste capítulo.
112   } // fim da classe Recommendations
```

Figura 29.30 | Exibe recomendações sobre livros com base em uma propriedade `SessionBean`.

29.8 Conclusão

Neste capítulo, introduzimos o desenvolvimento de aplicativos Web utilizando JavaServer Pages e JavaServer Faces no NetBeans. Começamos elencando as transações HTTP simples que acontecem quando você solicita e recebe uma página Web por meio de um navegador Web. Discutimos então as três camadas (isto é, a camada cliente ou superior, a camada da lógica do negócio ou intermediária e a camada de informações ou inferior) que cobrem a maioria dos aplicativos Web.

Você aprendeu o papel dos arquivos JSP e dos arquivos de bean de página, e a relação entre eles. Aprendeu a utilizar o NetBeans para compilar visualmente aplicativos Web utilizando as capacidades de arrastar e soltar do NetBeans, você então compilou e executou-os.

Demonstramos vários componentes JSF comuns utilizados para exibir texto e imagens em páginas Web. Também discutimos componentes de validação e métodos validadores personalizados, que permitem assegurar que a entrada do usuário satisfaz os requisitos do seu aplicativo.

Enumeramos os benefícios de manter informações de usuário em múltiplas páginas de um site Web. Demonstramos então como você pode incluir essas funcionalidades em um aplicativo Web utilizando cookies ou propriedades da classe de bean de sessão que é incluída em cada aplicativo Web.

No Capítulo 30, continuamos nossa discussão do desenvolvimento de aplicativos Web em Java com conceitos mais avançados. Você aprenderá a acessar um banco de dados a partir de um aplicativo Web JSF, utilizar componentes JSF compatíveis com AJAX e utilizar formulários virtuais. O AJAX ajuda aplicativos baseados na Web a fornecer interatividade e responsividade que os usuários em geral esperam de aplicativos desktop.

Resumo

Seção 29.1 Introdução

- Aplicativos baseados na Web criam conteúdo, como a Extensible HyperText Markup Language (XHTML), criação de script no lado do cliente, imagens e dados binários, para clientes baseados em navegadores Web.

Seção 29.2 Transações HTTP simples

- Na sua forma mais simples, uma página Web nada mais é do que um documento XHTML que descreve para um navegador Web como exibir e formatar as informações do documento.
- Documentos XHTML normalmente contêm hiperlinks que levam a diferentes páginas ou a outras partes da mesma página. Quando o usuário clica em um hiperlink, a página da Web solicitada é carregada no navegador.
- O protocolo HTTP permite a clientes e servidores interagir e trocar informações.
- O HTTP utiliza o URIs (Uniform Resource Identifiers) para identificar dados na internet. URIs que especificam as localizações de documentos são chamados URLs (Uniform Resource Locators).
- Um URL contém informações que direcionam um navegador ao recurso que o usuário quer acessar. Computadores que executam softwares de servidor Web disponibilizam esses recursos.
- O computador que hospeda e mantêm recursos normalmente é tratado como host.
- Nomes de host são convertidos em endereços IP pelos servidores de Domain-Name System (DNS).
- Quando dado um URL, um navegador Web realiza uma transação HTTP para recuperar e exibir a página Web nesse endereço.
- Cabeçalhos HTTP fornecem informações adicionais sobre os dados que serão enviados.
- O Multipurpose Internet Mail Extensions (MIME) é um padrão internet que especifica formatos de dados para que os programas possam interpretar os dados corretamente.
- Os dois tipos mais comuns de solicitação de HTTP são `GET` e `POST`. Uma solicitação `GET` em geral pede um recurso específico em um servidor. Uma solicitação `POST` geralmente posta (ou envia) dados para o servidor.
- Tanto solicitações `GET` como solicitações `POST` podem ser utilizadas para enviar dados de formulário a um servidor Web, mas cada tipo de solicitação envia as informações de uma maneira diferente. Uma solicitação `GET` envia informações ao servidor no URL. Uma solicitação `POST` envia dados de formulário como parte da mensagem HTTP.
- Os navegadores costumam armazenar em cache páginas Web para recarregamento rápido. Se não houver alterações entre a última versão armazenada no cache e a versão atual na Web, isso acelerará sua navegação.
- Uma resposta HTTP pode indicar por quanto tempo o conteúdo permanece atualizado. Se esse período de tempo não for atingido, o navegador poderá impedir outra solicitação ao servidor.
- Os navegadores, em geral, não armazenam em cache da resposta do servidor para uma solicitação `POST` porque a próxima solicitação `POST` pode não retornar o mesmo resultado.

Seção 29.3 Arquitetura de aplicativo multithread

- Aplicativos baseados na Web são aplicativos multicamadas (ou aplicativos de *n* camadas) que dividem as funcionalidades em camadas separadas (isto é, agrupamentos lógicos das funcionalidades). Embora as camadas possam estar localizadas no mesmo computador, as camadas dos aplicativos baseados na Web costumam residir em computadores separados.
- A camada de informações mantém os dados que pertencem ao aplicativo.
- A camada intermediária implementa a lógica do negócio, a lógica do controlador e a lógica da apresentação a fim de controlar interações entre os clientes do aplicativo e os dados do aplicativo. A lógica do negócio na camada intermediária impõe regras do negócio e assegura que os dados são confiáveis antes de o aplicativo servidor atualizar o banco de dados ou apresentar os dados aos usuários. As regras do negócio determinam como os clientes podem e como não podem acessar dados de aplicativo, e como os aplicativos processam dados.
- A camada cliente é a interface com o usuário do aplicativo, que coleta a entrada e exibe a saída. Os usuários interagem diretamente com o aplicativo por meio da interface com o usuário. Em resposta às ações do usuário (por exemplo, clicar em um hiperlink), a camada de cliente interage com a camada intermediária para fazer solicitações e recuperar dados da camada de informações.

Seção 29.4 Tecnologias Web Java

- Aplicativos Java multicamadas são implementados utilizando os recursos do Java Enterprise Edition.

Section 29.4.1 Servlets

- Servlets utilizam o modelo de solicitação/resposta HTTP da comunicação entre cliente e servidor.
- Servlets estendem a funcionalidade de um servidor permitindo que ele gere conteúdo dinâmico.
- Um componente de servidor Web chamado contêiner de servlets executa e interage com servlets. O contêiner de servlets recebe solicitações HTTP de um cliente e direciona cada solicitação ao servlet apropriado. O servlet processa a solicitação e retorna uma resposta apropriada ao cliente.

- Todos os servlets devem implementar a interface `Servlet` do pacote `javax.servlet`, que assegura que cada servlet pode executar no framework fornecido pelo contêiner de servlets.
- A interface `Servlet` declara métodos de ciclo de vida dos servlets. O contêiner invoca o método `init` do servlet somente uma vez durante o ciclo de vida de um servlet para inicializar o servlet. Depois que `init` completar a execução, o servlet estará pronto para responder à sua primeira solicitação. Cada solicitação é tratada por um método `service` do servlet, que recebe a solicitação, processa e envia uma resposta ao cliente.
- Quando o contêiner de servlets termina o servlet, o método `destroy` do servlet é chamado para liberar quaisquer recursos mantidos pelo servlet.

Section 29.4.2 JavaServer Pages

- A tecnologia JavaServer Pages (JSPs) é uma extensão da tecnologia de servlet. Cada JSP é convertido pelo contêiner JSP em um servlet.
- JSPs ajudam a separar a apresentação do conteúdo. As JSPs permitem criar conteúdo dinâmico reutilizando componentes predefinidos e interagindo com componentes que utilizam script do lado do servidor.
- Um JavaBean é um componente reutilizável que segue certas convenções para o design de classe e que pode ser manipulado visualmente em uma ferramenta de construtor, como o NetBeans ou Eclipse. As classes JavaBean que permitem leitura e gravação das variáveis de instância devem fornecer métodos *get* e *set* apropriados — juntas, estas definem as propriedades das classes JavaBean.
- Bibliotecas de tags personalizados permitem ocultar o código para acesso de banco de dados e outras operações complexas nos tags personalizados. Para utilizar essas capacidades, você simplesmente adiciona os tags personalizados à página. Essa simplicidade permite a designers de página Web que não estão familiarizados com o Java aprimorar páginas Web com capacidades poderosas de processamento dinâmico e conteúdo dinâmico.
- Há quatro componentes-chave para JSPs — diretivas, ações, elementos de script e bibliotecas de tags. Diretivas são mensagens ao contêiner JSP que permitem especificar configurações de página, incluir conteúdo de outros recursos e especificar bibliotecas de tags personalizados para uso em JSPs. As ações encapsulam funcionalidades em tags predefinidos que programadores podem incorporar em JSPs. Elementos de script permitem inserir código Java que interage com componentes em um JSP para executar o processamento de solicitações. As bibliotecas de tags fazem parte do mecanismo de extensão de tag que permite aos programadores criar tags personalizados. A JavaServer Pages Standard Tag Library (JSTL) fornece funcionalidade para muitas tarefas comuns de aplicativo Web.
- JSPs podem conter conteúdo estático, como marcação XML ou XHTML. Tal marcação é conhecida como dados de template fixa ou texto de template fixa.
- Quando um servidor com JSP ativado recebe a primeira solicitação para uma JSP, o contêiner de JSPs traduz a JSP em um servlet Java que trata a solicitação atual e as futuras solicitações para a JSP.

Section 29.4.3 JavaServer Faces

- O JavaServer Faces (JSF) é um framework de aplicativo Web que simplifica o design da interface com o usuário de um aplicativo e amplia a separação entre a apresentação de um aplicativo Web e a lógica do negócio.
- Componentes JSF simplificam o design de páginas Web. O JSF fornece duas bibliotecas de tags personalizados JSP para adicionar esses componentes a uma página JSP. O JSF também inclui APIs para tratar eventos de componentes, navegação entre páginas de aplicativo Web e mais.
- Você cria a aparência e o funcionamento de uma página com o JSF adicionando elementos a um documento JSP e manipulando seus atributos. Você define o comportamento da página separadamente nos arquivos Java relacionados.

Section 29.4.4 Tecnologias Web no NetBeans

- Aplicativos Web NetBeans que utilizam o framework JavaServer Faces consistem em uma ou mais páginas Web JSP. Esses JSPs têm a extensão de nome de arquivo `.jsp` e contêm os elementos GUI das páginas Web. JSPs podem ser personalizados no NetBeans adicionando componentes JSF.
- Cada arquivo JSP criado no NetBeans representa uma página Web e tem uma classe JavaBean correspondente chamada bean de página. Uma classe JavaBean deve ter um construtor padrão (ou sem argumento), e métodos *get* e *set* para todas as propriedades do bean (isto é, variáveis de instância). O bean de página define as propriedades para todos os elementos da página com os quais você quer interagir programaticamente. O bean de página também contém handlers de evento e métodos de ciclo de vida de página para gerenciar tarefas, como inicialização e renderização de página, e outros códigos de suporte ao aplicativo Web.
- Cada aplicativo Web NetBeans JSF define três mais JavaBeans. O objeto `RequestBean` é mantido no escopo de solicitação — ele só existe pelo tempo de duração de uma solicitação HTTP. Um objeto `SessionBean` tem escopo de sessão — ele existe por toda a sessão de navegação de um usuário ou até que a sessão expire. Há um único objeto `SessionBean` para cada usuário. Por fim, o objeto `ApplicationBean` tem escopo de aplicativo — ele é compartilhado por todas as instâncias de um aplicativo e existe desde que o aplicativo permaneça implantado em um servidor Web. Esse objeto é utilizado para armazenagem ou processamento de dados por todo o aplicativo; somente uma instância existe para o aplicativo, independentemente do número de sessões abertas.

Section 29.5.1 Examinando um documento JSP

- Todos os JSPs devem ter um elemento `jsp:root`, que tem um atributo `version` para indicar a versão JSP que é utilizada e um ou mais atributos `xmlns`. Cada atributo `xmlns` especifica um prefixo e um URL para uma biblioteca de tags, permitindo que a página utilize elementos dessa biblioteca.
- O atributo `contentType` do elemento `jsp:directive.page` especifica o tipo MIME (`text/html`) e o conjunto de caracteres (`UTF-8`) que a página utiliza. Seu atributo `pageEncoding` especifica a codificação de caractere utilizada pelo código-fonte da página. Esses atributos ajudam o navegador a renderizar o conteúdo.
- Todas as páginas que contêm componentes JSF são representadas em uma árvore de componentes com o elemento JSF raiz `f:view` em que outros tags de componentes JSF são aninhados.

- O elemento `webuijsf:head` tem um atributo `title` que especifica o título da página.
- O elemento `webuijsf:body` contém um elemento `webuijsf:form` no qual os componentes JSF da interface com o usuário são definidos.
- Um `webuijsf:staticText` exibe o texto na página.
- A JSF Expression Language permite vincular um atributo de um elemento JSP ao valor de uma propriedade em qualquer JavaBean do aplicativo Web.
- Elementos JSP são mapeados pelo servidor de aplicativo para uma combinação de elementos XHTML e código JavaScript que permite ao navegador exibir a página.
- O JavaScript é uma linguagem de criação de scripts que é interpretada em todos os atuais navegadores Web populares. Eles podem ser utilizados para manipular elementos de página Web em um navegador e fornecer interatividade com o usuário.
- O mesmo componente Web pode mapear para diferentes elementos XHTML e código JavaScript, dependendo do navegador cliente e das configurações de propriedade do componente.

Section 29.5.2 Examinando um arquivo Page Bean

- Classes de bean de página estendem a classe `AbstractPageBean` (do pacote `com.sun.rave.web.ui.appbase`), que fornece métodos de ciclo de vida de página.
- O pacote `com.sun.webui.jsf.component` fornece boa parte dos componentes GUI JSF básicos.

Section 29.5.3 Ciclo de vida de processamento de eventos

- Vários métodos no bean de página associam ao ciclo de vida do processamento de eventos JSF quatro etapas principais — inicialização, pré-processamento, pré-renderização e destruição. Cada um equivale a um método na classe de bean de página — `init`, `preprocess`, `prerender` e `destroy`, respectivamente.
- O contêiner JSP chama `init` na primeira vez que a página é solicitada e nos postbacks. Um postback ocorre quando os dados de formulário são submetidos para processamento no servidor.
- O método `preprocess` é chamado depois de `init`, mas somente se a página estiver processando um postback. O método `prerender` configura propriedades de componente imediatamente antes de uma página ser exibida pelo navegador.
- O método `destroy` é chamado depois que a página foi exibida, mas somente se o método `init` for chamado. Esse método trata tarefas, como liberar os recursos utilizados para renderizar a página.

Section 29.5.4 Criando um aplicativo Web no NetBeans

- Para criar um projeto de aplicativo Web, selecione **File> New Project...** para exibir a caixa de diálogo **New Project**. Selecione **Java Web** no painel **Categories**, **Web Application** no painel **Projects** e clique em **Next >**. Nomeie o projeto e, no campo **Project Location**, especifique onde você quer armazenar o projeto. Mantenha as outras configurações padrão e clique em **Next>**. Mantenha as opções padrão para **Server** e **Settings** e clique em **Next >**. Selecione **Visual Web JavaServer Faces** como o framework a utilizar nesse aplicativo Web, clique então em **Finish** para criar o projeto do aplicativo Web.
- A **Palette** exibida no IDE mostra os componentes Woodstock. Woodstock é um conjunto dos componentes da interface com o usuário para aplicativos JavaServer Faces.
- A janela **Projects** exibe a hierarquia de todos os arquivos incluídos no projeto. Os arquivos JSP para cada página são listados abaixo do nó **Web Pages**. O código Java está no nó **Source Packages**.
- A refatoração é o processo de modificar o código-fonte para aprimorar sua legibilidade e capacidade de reutilização sem alterar seu comportamento. O NetBeans tem ferramentas de refatoração internas.
- Para adicionar componentes à página no modo **Design**, arraste e solte-os da **Palette** até a página. Cada componente é um objeto que tem propriedades, métodos e eventos. Você pode manipular estes visualmente utilizando a janela **Properties** ou programaticamente no arquivo de bean de página.
- O NetBeans é um editor WYSIWYG (What You See Is What You Get) — quando você faz uma modificação em uma página no modo **Design**, o IDE cria a marcação (visível no modo **JSP**) necessária para alcançar os efeitos visuais desejados vistos no modo **Design** e também poderia adicionar código Java ao arquivo `.java` do bean de página.
- Na janela **Properties**, clique no botão de reticências (...) ao lado da propriedade `style` de um componente para abrir uma caixa de diálogo a fim de editar o estilo do componente.
- Para interagir com um componente JSF programaticamente, você primeiro deve clicar com o botão direito do mouse no controle no modo **Design** e escolher **Add Binding Attribute**. Isso cria uma variável no bean de página que você pode utilizar para interagir com o componente e métodos *set* e *get* para acessar o componente.
- Execute seu projeto selecionando **Run> Build Project**, então **Run> Run Project**. Você também pode executar um projeto que já foi compilado pressionando o ícone **Run Project** ( ) na barra de ferramentas na parte superior do IDE ou pressionando *F6*.
- Pressione *Ctrl* + *F5* para compilar o aplicativo, então o execute no modo de depuração. Se pressionar *F6*, o programa executará sem a depuração ativada.
- Para projetos com múltiplas páginas, você pode alterar a página inicial dando um clique com o botão direito do mouse na página desejada na janela **Projects** e selecionando **Set As Start Page**. A página inicial é indicada por um ícone com um símbolo de botão play verde ( ) ao lado do nome da página na janela **Projects**.

Section 29.6.1 Componentes gráficos e de texto

- O NetBeans utiliza o posicionamento absoluto por padrão, portanto, os componentes são exibidos onde quer que eles sejam soltos no editor visual.
- Um Grid Panel permite ao designer especificar o número de colunas que a grade deve conter. Os componentes são posicionados em colunas uniformemente espaçadas na ordem na qual eles são adicionados.
- Para adicionar uma imagem, solte um componente Image na página e clique no botão de reticências (...) ao lado da propriedade url na janela Properties, selecione então a imagem a exibir.
- Você pode rotular um Text Field configurando a propriedade `label`, que posiciona o texto à esquerda do Text Field. Alternativamente, você pode rotular um Text Field arrastando e soltando um componente Label até a página, o que permite personalizar a posição e estilo do Label.
- A ordem na qual os componentes são arrastados até a página é a ordem na qual seus elementos JSP são adicionados ao documento JSP.
- Por padrão, quando um usuário pressiona a tecla *Tab* para navegar entre campos de entrada, o foco alterna entre um componente e outro na ordem em que os elementos JSP ocorrem no documento JSP. Para especificar a ordem de navegação, arraste os componentes para a página na ordem apropriada, ou configure cada propriedade `tabIndex` do campo de entrada na janela Properties.
- Uma Drop Down List exibe uma lista de opções em que o usuário pode fazer uma seleção. Um objeto `SingleSelectOptionsList` gerencia a lista de opções. Esse objeto pode ser configurado clicando com o botão direito do mouse na lista drop-down no modo Design e selecionando Configure Default Options..., que abre a caixa de diálogo Options Customizer para você poder adicionar opções à lista.
- O componente Hyperlink adiciona um hiperlink a uma página Web. Sua propriedade `url` especifica o recurso a solicitar quando um usuário clica no hiperlink.
- Um Radio Button Group fornece botões de opção a partir dos quais o usuário pode selecionar somente um. Um Radio Button Group é vinculado a um objeto `SingleSelectOptionList`. As opções podem ser editadas clicando com o botão direito do mouse no componente e selecionando Configure Default Options....
- Um componente Button desencadeia uma ação quando clicado e, em geral, mapeia para um elemento XHTML `input` com o atributo `type` configurado como `submit`.

Section 29.6.2 Validação utilizando componentes validadores e validadores personalizados

- A validação de formulário ajuda a evitar erros de processamento por causa de entrada de usuário incompleta ou impropriamente formatada. Um Length Validator determina se um campo contém um número aceitável de caracteres. Double Range Validators e Long Range Validators determinam se a entrada numérica está incluída em intervalos aceitáveis. Você também pode criar métodos validadores personalizados no arquivo de bean de página.
- Componentes message exibem mensagens de erro quando a validação falha.
- Para associar componentes Label e Message com uma Text Field correspondente, mantenha pressionado as teclas *Ctrl* e *Shift* e arraste o Label ou Message até o Text Field. Na janela Properties, observe que a propriedade `for` de cada componente Label e Message é configurada com o Text Field apropriado.
- Para assegurar que o usuário fez uma seleção ou inseriu algum texto em um elemento de entrada requerido marque a caixa `required` na janela Properties do elemento. A propriedade `required` de um componente deve ser configurada como `true` (marcada) para a validação ocorrer.
- Para adicionar um Length Validator a um componente, arraste o validador da Palette e solte-o no campo a validar. Configure as propriedades `maximum` e `minimum` do Length Validator como o número desejado de caracteres na janela Properties.
- A validação no lado do cliente pode ser burlada. Realize validação importante no servidor.
- É possível limitar o tamanho da entrada de usuário configurando a propriedade `maxLength` de um Text Field.
- Corresponder a entrada de usuário contra uma expressão regular é um modo efetivo de assegurar que a entrada está formatada apropriadamente. Isso pode ser feito com validadores personalizados.

Seção 29.7 Monitoramento de sessão

- A personalização permite que sites de comércio eletrônico se comuniquem efetivamente com seus clientes e também aprimora a capacidade do usuário de localizar produtos e serviços desejados.
- Há, porém, um dilema entre serviços de comércio eletrônico personalizados e proteção da privacidade. Alguns consumidores aceitam a ideia do conteúdo personalizado, mas outros temem as possíveis consequências adversas se as informações que eles fornecem aos sites de comércio eletrônico forem publicadas ou coletadas por tecnologias de rastreamento.
- O HTTP é um protocolo sem estado — ele não suporta conexões persistentes que permitiriam aos servidores Web manter informações de estado em relação a determinados clientes. Desse modo, os servidores Web não podem determinar se uma solicitação vem de um determinado cliente ou se uma série de solicitações vem de um ou mais clientes.
- Para ajudar o servidor a distinguir entre clientes, cada cliente deve identificar-se para o servidor. O rastreamento de clientes específicos, conhecido como rastreamento de sessão, pode ser alcançado de várias maneiras. Uma técnica popular utiliza cookies; outra utiliza o objeto `SessionBean`.
- Com elementos de formulário `"hidden"`, um Web Form pode gravar dados de rastreamento de sessão em um `form` na página Web que ele retorna ao cliente em resposta a uma solicitação anterior. Quando o usuário submete o formulário na nova página Web, todos os dados de formulário, incluindo os campos `"hidden"`, são enviados ao handler de formulário no servidor Web. Com a rescrita de URL, o servidor Web incorpora as informações de rastreamento de sessão diretamente nos URLs dos hiperlinks em que o usuário clica para enviar solicitações subsequentes ao servidor Web.

Section 29.7.1 Cookies

- Um cookie é um dado, em geral, armazenado em um arquivo de texto no computador do usuário. Um cookie mantém informações sobre o cliente durante e entre sessões de navegador.
- Na primeira vez que um usuário visita o site Web, o computador do usuário poderia receber um cookie; esse cookie é então reativado toda vez que o usuário revisita esse site. As informações coletadas visam ser um registro anônimo que contém dados utilizados para personalizar as futuras visitas do usuário ao site.
- Cada interação baseada em HTTP entre um cliente e um servidor inclui um cabeçalho contendo as informações sobre a solicitação (quando a comunicação é do cliente para o servidor) ou sobre a resposta (quando a comunicação é do servidor para o cliente).
- Quando uma página recebe uma solicitação, o cabeçalho inclui informações, como o tipo de solicitação e quaisquer cookies que foram enviados anteriormente do servidor para serem armazenados na máquina do cliente. Quando o servidor formula sua resposta, as informações de cabeçalho contêm todos os cookies que o servidor quer armazenar no computador do cliente e outras informações, como o tipo MIME da resposta.
- A data de expiração de um cookie determina quanto tempo o cookie permanece no computador do cliente. Se você não configurar a data de expiração de um cookie, o navegador Web manterá o cookie pelo tempo de duração da sessão de navegação. Do contrário, ele manterá o cookie até a data de expiração.
- Configurar o handler de ação para um Hyperlink permite responder a um clique sem redirecionar o usuário a outra página.
- Para adicionar um handler de ação a um Hyperlink que também deve direcionar o usuário a outra página, você deve adicionar uma regra ao arquivo Page Navigation. Para editar esse arquivo, clique com o botão direito do mouse no Designer Visual e escolha Page Navigation..., então arraste o Hyperlink apropriado até a página de destino.
- Um objeto cookie é uma instância da classe `Cookie` no pacote `javax.servlet.http`.
- Um objeto da classe `HttpServletResponse` (do pacote `javax.servlet.http`) representa a resposta. Esse objeto pode ser acessado invocando o método `getExternalContext` no bean de página e então invocando `getResponse` no objeto resultante.
- Um objeto da classe `HttpServletRequest` (do pacote `javax.servlet.http`) representa a solicitação. Esse objeto pode ser obtido invocando o método `getExternalContext` no bean de página, e depois invocando `getRequest` no objeto resultante.
- O método `addCookie HttpServletRespone` adiciona um cookie à resposta HTTP.
- O método `HttpServletRequest getCookies` retorna os cookies anteriormente gravados no cliente.
- Um servidor Web não pode acessar cookies criados por servidores em outros domínios.

Section 29.7.2 Rastreamento de sessão com beans de sessão

- Você pode executar o rastreamento de sessão com a subclasse `AbstractSessionBean` fornecida em todo aplicativo Web do NetBeans. Por padrão, a subclasse é nomeada `SessionBean1`. As propriedades desse objeto podem ser acessadas por toda uma sessão de navegador invocando o método `getSessionBean1` no bean de página e então invocando os métodos apropriados para acessar as propriedades do bean de sessão.
- Para armazenar informações de sessão, adicione propriedades ao bean de sessão. Para fazer isso, declare uma variável do tipo apropriado no bean de sessão, então clique com o botão direito do mouse no nome da variável e escolha Refactor> Encapsulate Fields... para criar os métodos *get* e *set* apropriados.

Terminologia

`AbstractPageBean`, classe, 954
`AbstractSessionBean`, classe, 979
ação, 951
aplicativo *de n* camadas, 950
aplicativo de múltiplas camadas, 950
`ApplicationBean`, 952
árvore componente, 953
bean de página, 952
biblioteca de tags, 951
bibliotecas de tags personalizados, 951
Button, componente JSF, 964
cache, 949
camada de cliente, 950
camada de dados, 950
camada de informações, 950
camada em um aplicativo de múltiplas camada, 950
camada inferior, 950
camada intermediária, 950
camada superior, 950
ciclo de vida do processamento de evento, 956
`com.sun.rave.web.ui.appbase`, pacote, 954
`com.sun.webui.jsf.component`, pacote, 954
contêiner de servlets, 951
cookie, 971
dados de template fixa, 951
data de expiração de um cookie, 971
Design, modo no NetBeans, 956
diretiva, 951
diretório virtual, 948
DNS (*Domain Name System*), servidor, 948
Double Range Validator, componente JSF, 964
Drop Down List, componente JSF, 964
editor visual (NetBeans), 956
elemento de script, 951
endereço de IP, 948
`escape`, propriedade de um componente Static Text, 965
escopo de aplicativo, 952
escopo de sessão, 952
escopo de solicitação, 952
`f:view`, elemento, 953
framework, 952
`GET`, solicitação HTTP, 948
GlassFish, servidor de aplicativo, 947
Grid Panel, componente JSF, 963
handler do formulário do lado do servidor, 949
host, 948
hostname, 948
Hyperlink, componente JSF, 964
Image, componente JSF, 963
`init`, método, 956
JavaBean, 951
JavaServer Faces (JSF), 952
JavaServer Faces (JSF) Expression Language, 954
JavaServer Pages Standard Tag Library (JSTL), 951
`javax.faces.validators`, pacote, 964
`javax.servlet.http`, pacote, 951
`javax.servlet`, pacote, 951
JSF (JavaServer Faces), 952
JSP, contêiner, 951
`.jsp`, extensão de nome do arquivo, 952
JSP (JavaServer Pages), 951

JSTL (JavaServer Pages Standard Tag Library), 951
Label, componente JSF, 964
Length Validator, componente JSF, 964
lógica da apresentação, 950
lógica do controlador, 950
lógica do negócio, 950
Long Range Validator, componente JSF, 964
mecanismo de extensão de tags, 951
Message, componente JSF, 964
MIME (Multipurpose Internet Mail Extensions), 949
Multipurpose Internet Mail Extensions (MIME), 949
Navigator, janela no NetBeans, 960
NetBeans IDE, 947
Palette no NetBeans, 957
personalização, 970
pesquisa de DNS, 948
posicionamento absoluto, 959
postback, 956
`preprocess`, método (JSF), 956
`prerender`, método (JSF), 956
protocolo de estado (HTTP), 971
Radio Button Group, componente JSF, 964
refatorando, 959
regra de negócio, 950
`rendered`, propriedade de um componente JSF, 965
renderizando XHTML em um navegador da Web, 949
`RequestBean`, 952
`required`, propriedade de um componente JSF, 968
resposta de servidor, 949
`service`, método da interface `Servlet`, 951
servidor Web, 948
servlet, 951
`Servlet`, interface, 951
sessão, monitoramento, 971
`SessionBean`, 952
sistema de nome de domínio (*Domain Name System* — DNS), servidor, 948
solicitação, método de, 949
`span`, elemento, 954
Static Text, componente JSF, 953
string de consulta, 949
tags personalizados, 951
Text Area, componente JSF, 978
texto de template fixa, 951
Text Field, componente JSF, 964
`UIViewRoot`, classe, 953
validação, 964
`xmlns`, atributos, 953

Exercícios de autorrevisão

29.1 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Cada página Web JSP criada no NetBeans tem seus próprios arquivos `ApplicationBean`, `SessionBean` e `RequestBean`.

b) O método de ciclo de vida do processamento de eventos `init` é invocado toda vez que uma página carrega.

c) Cada componente em uma página Web JSP é vinculado a uma propriedade no arquivo de bean de página Java.

d) Um único componente JSF pode conter múltiplos componentes de validação.

e) Se nenhuma data de expiração for configurada para um cookie, esse cookie será destruído no fim da sessão do navegador.

f) Cada componente JSF mapeia para exatamente um único elemento XHTML correspondente.

g) Expressões na sintaxe da JSF Expression Language são delimitadas por `<!--` e `-->`.

h) O `SessionBean` só pode armazenar propriedades primitivas e propriedades do tipo `String`.

29.2 Preencha as lacunas em cada uma das seguintes afirmações:

a) Aplicativos Web contêm três camadas básicas: _____, _____ e _____.

b) O componente JSF _____ é utilizado para exibir mensagens de erro se a validação falhar.

c) Um componente que verifica a entrada em outro componente antes de submeter essa entrada ao servidor é chamado _____.

d) Cada classe de bean de página estende a classe _____.

e) Quando uma página é carregada pela primeira vez, o evento _____ ocorre primeiro, seguido pelo evento _____.

f) O arquivo _____ contém as funcionalidades de um JSP.

g) Um(a) _____ pode ser utilizado(a) em um método validador personalizado para validar o formato da entrada de usuário.

h) O array de objetos `Cookie` armazenados no cliente pode ser obtido chamando `getCookies` no objeto _____.

i) Em um aplicativo multicamadas, a camada _____ controla as interações entre os clientes do aplicativo e os dados do aplicativo.

Respostas dos exercícios de autorrevisão

29.1 a) Falso. Se um aplicativo contiver múltiplas JSPs, essas JSPs irão compartilhar os beans de dados com escopo. b) Falso. `init` é invocado na primeira vez que a página é solicitada, mas não nas atualizações de página. c) Falso. Você deve adicionar manualmente um atributo de vinculação ao componente JSF para que ele tenha uma propriedade no arquivo do bean de página. d) Verdadeiro. e) Verdadeiro. f) Falso. Um componente Web pode mapear para um grupo de elementos XHTML — JSPs podem gerar a marcação XHTML complexa a partir de componentes simples. g) Falso. `#{` e `}` delimitam instruções da JSF Expression Language. h) Falso. Os beans de dados com escopo podem armazenar qualquer tipo de propriedade.

29.2 a) inferior (dados/informações), intermediária (lógica do negócio), superior (cliente/interface com o usuário). b) Message. c) Validador. d) `AbstractPageBean`. e) `init`, `prerender`. f) Bean de página. g) Expressão regular. h) Solicitação (`HttpServletRequest`). i) Intermediária.

Exercícios

29.3 (***Modificação de WebTime***) Modifique o exemplo de `WebTime` para conter listas drop-down que permitem ao usuário modificar propriedades do componente Static Text como `background-color`, `color` e `font-size`. Quando o usuário faz uma seleção, a página deve atualizar e refletir as modificações especificadas nas propriedades do Static Text exibindo a hora.

29.4 (***Modificação do formulário de registro***) Modifique o aplicativo `WebComponents` para adicionar funcionalidade ao botão **Register**. Quando o usuário clicar em **Submit**, valide todos os campos de entrada para certificar-se de que o usuário preencheu o formulário completamente e inseriu um endereço de e-mail e número de telefone válidos. Em seguida, direcione o usuário a outra página que exibe uma mensagem indicando um registro bem-sucedido e ecoa de volta as informações de registro do usuário.

29.5 (***Contador de hits de página com cookies***) Crie um JSP que utiliza um cookie persistente (isto é, um cookie com uma data de expiração no futuro) para rastrear quantas vezes o computador cliente visitou a página. Utilize o método `setMaxAge` para fazer o cookie permanecer no computador do cliente por um mês. Exiba o número de hits de página (isto é, o valor do cookie) toda vez que a página for carregada.

29.6 (***Contador de hits de página com `ApplicationBean`***) Crie um JSP que utiliza o `ApplicationBean` para rastrear quantas vezes uma página foi visitada. [*Nota*: se você fosse implantar essa página na Web, ela contaria o número de vezes que qualquer computador solicitou a página, diferentemente do exercício anterior.] Exiba o número de hits de página (isto é, o valor de uma propriedade `int` no `ApplicationBean`) toda vez que a página for carregada.

30

O que quer que seja belo de alguma forma tem sua fonte de beleza em si mesmo, e é completo em si mesmo; o elogio não é parte dele.
— **Marco Aurélio Antonino**

There is something in a face, an air, and a peculiar grace, which boldest painters cannot trace. [Existe algo em uma face, um ar, e uma graça peculiar, que os maiores pintores não podem reproduzir.]
— **William Somerville**

Cato disse que a melhor maneira de manter as boas ações na memória era refrescá-las com novas.
— **Francis Bacon**

Nunca me esqueço de um rosto, mas em seu caso farei uma exceção.
— **Groucho Marx**

Aplicativos Web JavaServer™ Faces compatíveis com Ajax

Objetivos

Neste capítulo, você aprenderá:

- A utilizar provedores de dados para acessar bancos de dados de aplicativos Web incorporados em NetBeans.
- Os princípios básicos e as vantagens da tecnologia Ajax.
- A utilizar componentes JSF compatíveis com Ajax em um projeto de aplicativo Web baseado em NetBeans.
- A configurar formulários virtuais que permitem que subconjuntos de componentes de entrada de um formulário sejam enviados para o servidor.

Sumário

30.1 Introdução
30.2 Acessando bancos de dados em aplicativos Web
30.2.1 Criando um aplicativo Web que exibe dados a partir de um banco de dados
30.2.2 Modificando o arquivo Page Bean para o aplicativo `AddressBook`
30.3 Componentes JSF compatíveis com o Ajax
30.4 Criando um **Text Field** de autocompletamento e utilizando formulários virtuais
30.4.1 Configurando formulários virtuais
30.4.2 Arquivo JSP com formulários virtuais e Autocomplete `Text Field`
30.4.3 Fornecendo sugestões para `Text Field` de autocomplemento
30.4.4 Exibindo as informações do contato
30.5 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

30.1 Introdução

Este capítulo continua nossa discussão sobre desenvolvimento de aplicativo Web com vários conceitos avançados. Discutimos como acessar, atualizar e pesquisar bancos de dados em um aplicativo Web, adicionar formulários virtuais a páginas Web para ativar subconjuntos de componentes de entrada de um formulário a serem enviados para o servidor, e como utilizar as bibliotecas de componentes compatíveis com Ajax para aprimorar o desempenho do aplicativo e a responsividade do componente.

Apresentamos um aplicativo de um único catálogo de endereços desenvolvido em três etapas para ilustrar esses conceitos. O aplicativo é formado por um banco de dados Java DB para armazenar os nomes de contato e seus endereços.

O aplicativo de catálogo de endereços apresenta um formulário que permite ao usuário inserir um novo nome e endereço para armazenar no catálogo de endereços, e exibe o conteúdo do catálogo de endereços em formato de tabela. Ele também fornece um formulário de pesquisa que permite ao usuário procurar um contato e, se localizado, exibir o endereço do contato. A primeira versão desse aplicativo demonstra como adicionar contatos ao banco de dados e como exibir a lista de contatos em um componente JSF **Table**. Na segunda versão, adicionamos um componente Autocomplete **Text Field** compatível com Ajax e o habilitamos para que sugira uma lista de nomes de contato à medida que o usuário digita. Quando o usuário seleciona um contato, as informações do contato são exibidas.

Os exemplos deste capítulo, como os do Capítulo 29, foram desenvolvidos no NetBeans. Alguns componentes Woodstock que vêm com NetBeans, como o componente **Text Field**, são compatíveis com Ajax. Esses componentes utilizam o Dojo Toolkit no lado do cliente no navegador Web. Dojo é uma biblioteca JavaScript compatível com vários navegadores e plataformas para criar interfaces ricas com o usuário do lado do cliente e executar interações do Ajax com servidores Web. Para obter mais informações sobre esse conjunto de ferramentas, consulte nosso centro de recursos do Dojo em `www.deitel.com/dojo/`. [*Nota*: No futuro, os componentes Woodstock serão substituídos por IceFaces. Para obter informações sobre IceFaces, visite `icefaces.org`. O site IceFaces fornece um documento sobre como migrar do Woodstock para IceFaces em `www.icefaces.org/main/product/woodstock-migration.iface`.]

30.2 Acessando bancos de dados em aplicativos Web

Muitos aplicativos Web acessam bancos de dados para armazenar e recuperar dados persistentes. Nesta seção, construímos um aplicativo Web que utiliza um banco de dados Java DB para armazenar contatos no catálogo de endereços e exibir contatos do catálogo de endereços em uma página Web.

A página Web permite que o usuário insira novos contatos em um formulário. Esse formulário consiste em componentes **Text Field** para o nome, sobrenome, endereço da rua, cidade, estado e CEP do contato. O formulário também tem um botão **Submit** para enviar os dados para o servidor e um botão **Clear** para reinicializar os campos do formulário. O aplicativo armazena as informações de catálogo de endereços em um banco de dados chamado `AddressBook`, que tem uma única tabela chamada `Addresses`. (Fornecemos esse banco de dados no diretório de exemplos deste capítulo.) Esse exemplo também introduz o componente **Table** JSF, que exibe os endereços do banco de dados em formato tabular.

30.2.1 Criando um aplicativo Web que exibe dados a partir de um banco de dados

Agora explicamos como construir a GUI do aplicativo `AddressBook` e configurar uma vinculação de dados que permite que o componente **Table** exiba informações do banco de dados. Apresentamos o arquivo JSP gerado mais adiante na seção, e discutimos o arquivo de bean de página relacionado na Seção 30.2.2. Para construir o aplicativo `AddressBook`, siga estes passos:

Passo 1: criando o projeto

Em NetBeans, crie um novo projeto **Web Application** chamado `AddressBook`. Certifique-se de selecionar **Visual Web JavaServer Faces** como a estrutura, como você fez no Capítulo 29. Renomeie os arquivos JSP e arquivos de bean de página de `Page1` para `AddressBook` utilizando as ferramentas de refatoração.

Passo 2: criando o formulário de entrada de usuário

No modo **Design**, crie o formulário mostrado na Figura 30.1. Adicione um componente **Static Text** que contenha `"Add a contact to the address book":`. Utilize propriedade `style` do componente para configurar o tamanho da fonte com `18px`. Adicione seis pares de componentes **Label** e **Text Field** à página. Renomeie os **Text Fields** `firstNameTextField`, `lastNameTextField`, `streetTextField`, `cityTextField`, `stateTextField` e `zipTextField`. Configure a propriedade `required` de cada **Text Field** como `true` (marcada) selecionando o **Text Field** e clicando na caixa de seleção da propriedade `required`. Associe cada **Label** com seu **Text Field** correspondente mantendo as teclas *Ctrl* e *Shift* pressionadas e arrastando o rótulo ao **Text Field** correto. Adicione um atributo de vinculação ao bean de página para cada **Text Field** (clique com o botão direito do mouse no componente e selecione **Add Binding Attribute**). Por fim, adicione os botões **Submit** e **Clear** e configure-os com larguras fixas em pixels. Configure a propriedade **`primary`** do botão **Submit** como `true` (marcado) para que ele se destaque mais na página do que o botão **Clear** e para permitir ao usuário enviar um novo contato pressionando *Enter* em vez de clicar no botão. Configure a propriedade **`reset`** do botão **Clear** como `true` (marcado) para impedir a validação quando o usuário clicar no botão — já que estamos limpando os campos, não precisamos assegurar que eles contenham informações. Discutiremos o handler de ação do botão **Submit** quando apresentarmos o arquivo de bean de página. O botão **Clear** não precisa de um método de tratamento da ação — definir `reset` como `true` configura automaticamente o botão para redefinir todos os campos de entrada da página.

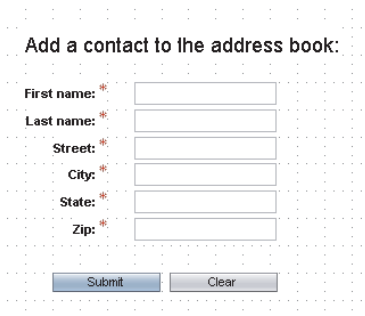


Figura 30.1 | Formulário do aplicativo `AddressBook` para adicionar um contato.

Passo 3: adicionando um componente `Table` à página

Arraste um componente **Table** da seção **Basic** da **Palette** à página e coloque-o logo abaixo dos dois componentes **Button**. Nomeie-o como `addressesTable`. O componente **Table** formata e exibe dados de tabelas de banco de dados. Na janela **Properties**, mude a propriedade `title` da **Table** para `Contacts`. A seguir mostramos como configurar a **Table** para interagir com o banco de dados `AddressBook`.

Passo 4: criando um banco de dados Java DB

Este exemplo utiliza um banco de dados chamado `AddressBook` para armazenar as informações de endereço. Para criar esse banco de dados, siga estes passos:

1. Na guia NetBeans **Services** (à direita das guias **Projects** e **Files**), expanda o nó **Databases**, clique com o botão direito do mouse **Java DB** e selecione **Create Database...**.
2. Digite o nome do banco de dados para criar (`AddressBook`), um nome de usuário (`test`) e uma senha (`test`).
3. Se desejar mudar o local em que o banco de dados está armazenado no sistema, clique no botão **Properties...** para especificar a nova localização.
4. Clique em **OK** para criar o banco de dados.

Na guia **Services**, os passos anteriores criam uma nova entrada no nó **Databases** que mostra o URL do banco de dados (`jdbc:derby://localhost:1527/AddressBook`). O novo banco de dados Java DB reside na máquina local e aceita conexões na porta 1527.

Passo 5: adicionando uma tabela e dados ao banco de dados `AddressBook`

Você pode utilizar a guia **Services** para criar tabelas e executar instruções SQL que preenchem o banco de dados com dados:

1. Na guia **Services**, expanda o nó **Databases**.

2. O NetBeans deve ser conectado ao banco de dados para executar as instruções SQL. Se o NetBeans já estiver conectado ao banco de dados, o ícone é exibido ao lado do URL do banco de dados (`jdbc:derby://localhost:1527/AddressBook`). Nesse caso, passe para o *Passo 3*. Se o NetBeans não estiver conectado ao banco de dados, o ícone aparece ao lado do URL do banco de dados. Nesse caso, clique com o botão direito do mouse no ícone e selecione **Connect...**. Uma vez conectado, o ícone muda para .
3. Expanda o nó para o banco de dados `AddressBook`, clique com o botão direito do mouse no nó **Tables** e escolha **Execute Command...** para abrir um editor **SQL Command** no NetBeans. Fornecemos o arquivo `AddressBook.sql` na pasta de exemplos deste capítulo. Abra esse arquivo em um editor de textos, copie as instruções SQL e cole-as no editor **SQL Command** no NetBeans. Então destaque todos os comandos SQL, clique com o botão direito do mouse dentro no editor **SQL Command** e selecione **Run Selection**. Isso criará a tabela `Addresses` com os dados de exemplo mostrados na Figura 30.2. [*Nota:* O script SQL tenta conectar-se ao banco de dados se ele já existir. Se não existir, você receberá uma mensagem de erro, mas o banco de dados ainda será criado corretamente.] Você pode precisar atualizar (clique com o botão direito do mouse e selecione **Refresh**) e expandir o nó **Tables** para ver a nova tabela. Você pode visualizar os dados na tabela expandindo o nó **Tables**, clicando com o botão direito do mouse em **ADDRESSES** e selecionando **View Data...**.

Nome	Sobrenome	Rua	Cidade	Estado	CEP
Bob	Green	5 Bay St.	San Francisco	CA	94133
Liz	White	100 5th Ave.	New York	NY	10011
Mike	Brown	3600 Delmar Blvd.	St. Louis	MO	63108
Mary	Green	300 Massachusetts Ave.	Boston	MA	02115
John	Gray	500 South St.	Philadelphia	PA	19147
Meg	Gold	1200 Stout St.	Denver	CO	80204
James	Blue	1000 Harbor Ave.	Seattle	WA	98116
Sue	Black	1000 Michigan Ave.	Chicago	IL	60605

Figura 30.2 | Dados da tabela `Addresses`.

Passo 6: vinculando o componente Table à tabela Addresses do banco de dados AddressBook

Agora que configuramos o banco de dados **AddressBook** e criamos a tabela `Addresses`, vamos configurar o componente **Table** para exibir os dados `AddressBook`. Arraste a tabela de banco de dados da guia **Services** e solte-a no componente **Table** para criar a vinculação.

Para selecionar colunas específicas a exibir, clique com o botão direito do mouse no componente **Table** e selecione **Bind to Data** para exibir o diálogo **Bind to Data** que contém a lista das colunas na tabela de banco de dados `Addresses` (Figura 30.3). Os itens abaixo do cabeçalho **Selected** serão exibidos no componente **Table**. Para remover uma coluna, selecione-a e clique no botão **<**. Queremos exibir todas as colunas neste exemplo, portanto, você deve simplesmente clicar em **OK** para fechar o diálogo.

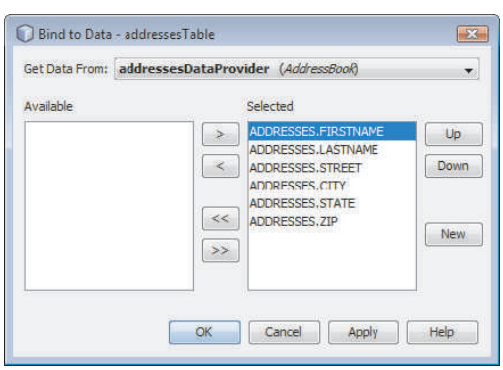


Figura 30.3 | Diálogo para vincular à tabela `Addresses`.

Por padrão, **Table** utiliza os nomes de coluna da tabela de banco de dados com todas as letras maiúsculas como cabeçalhos. Para mudar esses cabeçalhos, selecione uma coluna e edite sua propriedade `headerText` na janela **Properties**. Para selecionar uma coluna, clique no nome da coluna no modo **Design**. Também mudamos a propriedade `id` de cada coluna para tornar os nomes da variável no código mais

legíveis. No modo **Design**, os cabeçalhos de coluna da **Table** devem aparecer como os da Figura 30.4. Se alguns dos cabeçalhos da coluna recorrerem para duas linhas, você pode aumentar a largura da tabela.

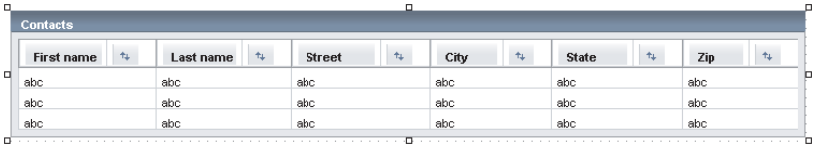


Figura 30.4 | O componente `Table` depois de vinculá-lo a uma tabela de banco de dados e editar os nomes de coluna com propósitos de visualização.

Um catálogo de endereços pode conter muitos contatos, portanto, gostaríamos de exibir apenas alguns deles por vez. Configure a propriedade **`paginationControls`** da tabela como `true` (marcada) e na janela **Properties** configure essa **Table** para paginação automática. Isso adiciona botões à parte inferior da **Table** para avançar e retroceder entre os grupos de contatos. Você pode utilizar a guia **Options** do diálogo **Table Layout** para selecionar o número de linhas a ser exibido por vez. Para fazer isso, clique com o botão direito do mouse no cabeçalho da **Table**, selecione **Table Layout...** e, então, clique na guia **Options**. Para esse exemplo, configuramos a propriedade **Page Size** como `5`. Clique no botão **OK** para aplicar as alterações.

Em seguida, configure a propriedade **`internalVirtualForm`** da `addressesTable` como `true` (marcada). Os formulários virtuais permitem que subconjuntos de componentes de entrada de um formulário sejam enviados para o servidor. Configurar essa propriedade impede que os botões de controle de paginação na **Table** enviem os **Text Fields** no formulário toda vez que o usuário quiser visualizar o grupo de contatos seguinte. Os formulários virtuais são discutidos na Seção 30.4.1.

Vincular o componente **Table** com um provedor de dados adicionou o objeto `addressesDataProvider` (uma instância da classe `CachedRowSetDataProvider`) ao nó **AddressBook** na janela **Navigator**. Um **`CachedRowSetDataProvider`** fornece um `RowSet` rolável que pode ser vinculado a um componente **Table** para exibir os dados do `RowSet`. Esse provedor de dados é um empacotador para um objeto `CachedRowSet`. Se clicar no nó **addressesDataProvider** na janela **Navigator**, pode ver na janela **Properties** que sua propriedade `CachedRowSet` está configurada como `addressesRowSet`, uma propriedade de bean de sessão que implementa a interface `CachedRowSet`.

Passo 7: modificando a instrução SQL do `addressesRowSet`

O objeto `CachedRowSet` empacotado por nosso `addressesDataProvider` é configurado por padrão para executar uma consulta SQL que seleciona todos os dados na tabela `Addresses` do banco de dados `AddressBook`. Você pode editar essa consulta SQL. Para tanto, assegure que você está no modo **Design** para o bean de página, então expanda o nó `SessionBean1` na janela **Navigator** e dê um clique duplo no elemento `addressesRowSet` para abrir a janela de editor de consulta (Figura 30.5). Gostaríamos de editar a instrução SQL para que os registros com sobrenomes duplicados sejam classificados por sobrenome e depois por nome. Para fazer isso, clique na coluna **Sort Type** ao lado da linha **LASTNAME** e selecione **Ascending**. Então repita isso para a linha **FIRSTNAME**. Note que a expressão

```
ORDER BY TEST.ADDRESSES.LASTNAME ASC, TEST.ADDRESSES.FIRSTNAME ASC
```

foi adicionada ao fim da instrução SQL na parte inferior do editor. Salve o aplicativo (**File> Save All**) e feche a guia de editor de consulta.

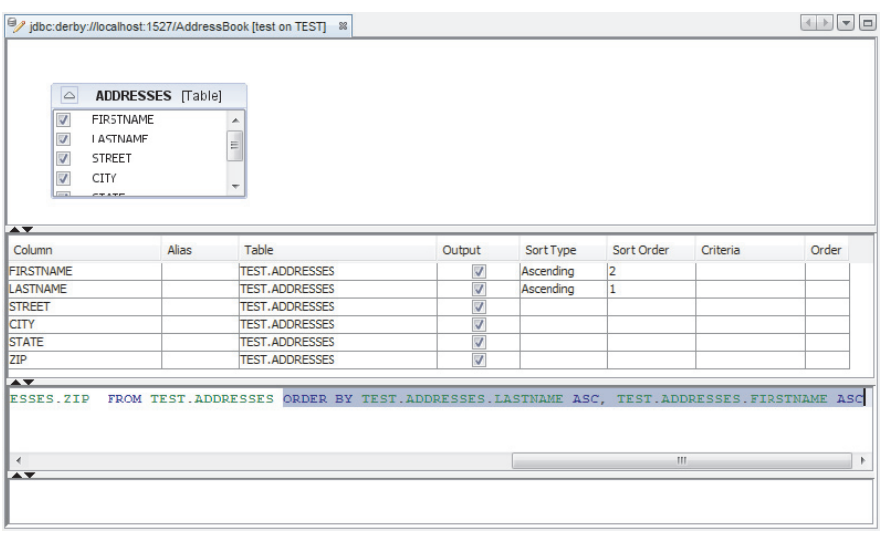


Figura 30.5 | Editando a instrução SQL do `addressesRowSet`.

Passo 8: adicionando a validação

É importante validar os dados de formulário nessa página para assegurar que os dados possam ser inseridos com sucesso no banco de dados `AddressBook`. Todas as colunas do banco de dados são do tipo `varchar` e têm restrições de tamanho. Por essa razão, você deve adicionar um **Length Validator** a cada componente **Text Field** ou configurar a propriedade `maxLength` de cada componente **Text Field**. Escolhemos fazer a última opção. Os componentes **Text Field** nome, sobrenome, rua, cidade, estado e CEP (zip) não podem exceder 30, 30, 150, 30, 2 e 5 caracteres, respectivamente.

Por fim, arraste um componente **Message Group** para a página à direita dos **Text Fields**. Esse componente exibe mensagens de sistema. Nós o utilizamos para exibir uma mensagem de erro quando uma tentativa de adicionar um contato ao banco de dados falha. Configure a propriedade `showGlobalOnly` do **Message Group** como `true` (marcada) para impedir que mensagens de erro de validação no nível do componente sejam exibidas aqui.

Revisando o documento JSP para uma página Web que interage com um banco de dados

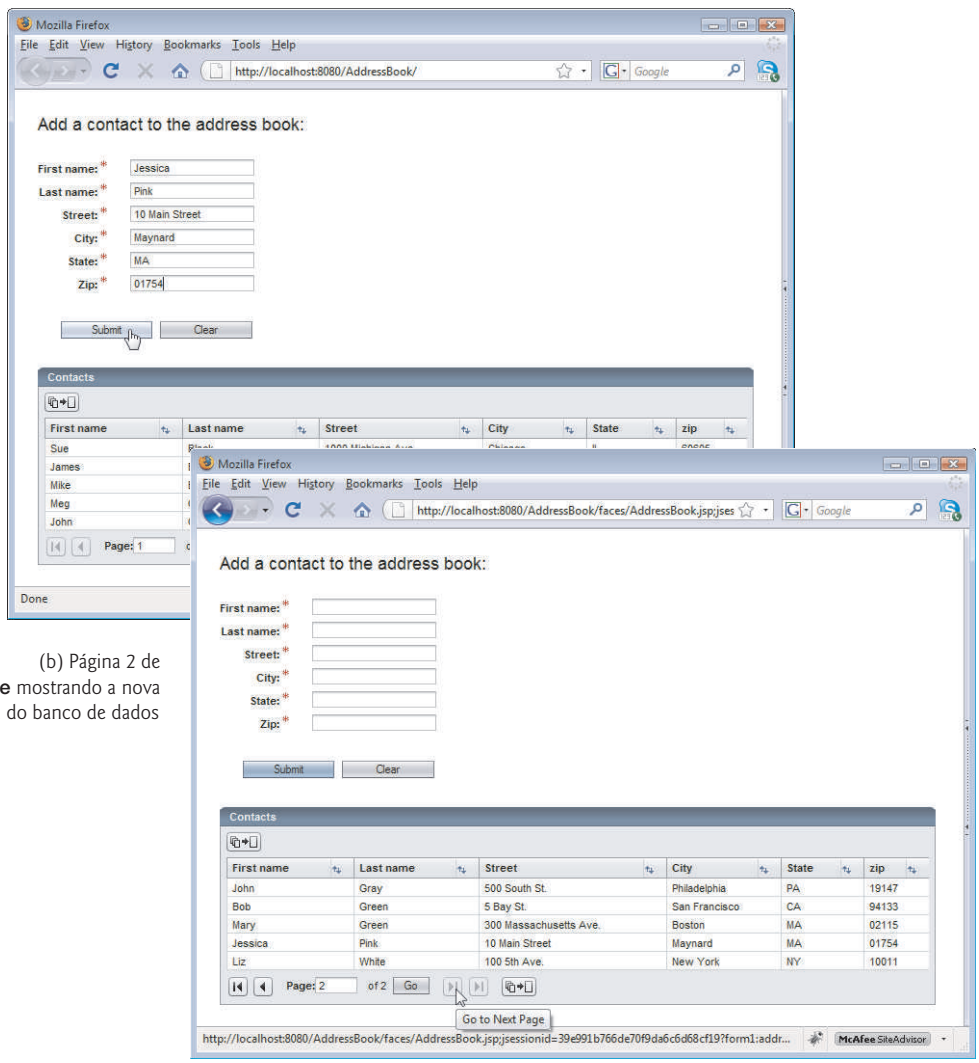
O arquivo JSP do aplicativo é mostrado na Figura 30.6. Esse arquivo contém uma grande quantidade da marcação gerada para componentes que você aprendeu no Capítulo 29. Discutimos apenas a marcação para os componentes que são novos nesse exemplo.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Figura 30.6: AddressBook.jsp -->
3  <!-- AddressBook JSP com um formulário de adição e um componente Table JSF. -->
4  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:jsp="http://java.sun.com/JSP/Page"
7     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
8     <jsp:directive.page contentType="text/html;charset=UTF-8"
9        pageEncoding="UTF-8"/>
10    <f:view>
11       <webuijsf:page id="page1">
12          <webuijsf:html id="html1">
13             <webuijsf:head id="head1">
14                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
15             </webuijsf:head>
16             <webuijsf:body id="body1" style="-rave-layout: grid">
17                <webuijsf:form id="form1">
18                   <webuijsf:staticText id="instructionText"
19                      style="font-size: 18px; left: 24px; top: 24px;
20                         position: absolute"
21                      text="Add a contact to the address book:"/>
22                   <webuijsf:label for="firstNameTextField"
23                      id="firstNameLabel" style="left: 24px; top: 72px;
24                         position: absolute" text="First name:"/>
25                   <webuijsf:textField
26                      binding="#{AddressBook.firstNameTextField}"
27                      id="firstNameTextField" maxLength="30"
28                      required="true" style="left: 120px; top: 72px;
29                         position: absolute"/>
30                   <webuijsf:label for="lastNameTextField"
31                      id="lastNameLabel" style="left: 25px; top: 96px;
32                         position: absolute" text="Last name:"/>
33                   <webuijsf:textField
34                      binding="#{AddressBook.lastNameTextField}"
35                      id="lastNameTextField" maxLength="30" required="true"
36                      style="left: 120px; top: 96px; position: absolute"/>
37                   <webuijsf:label for="streetTextField" id="streetLabel"
38                      style="left: 49px; top: 120px; position: absolute"
39                      text="Street:"/>
40                   <webuijsf:textField
41                      binding="#{AddressBook.streetTextField}"
42                      id="streetTextField" maxLength="150" required="true"
43                      style="left: 120px; top: 120px; position: absolute"/>
44                   <webuijsf:label for="cityTextField" id="cityLabel"
45                      style="left: 62px; top: 144px; position: absolute"
46                      text="City:"/>
47                   <webuijsf:textField
48                      binding="#{AddressBook.cityTextField}"
49                      id="cityTextField" maxLength="30" required="true"
50                      style="left: 120px; top: 144px; position: absolute"/>
```

```
51                <webuijsf:label for="stateTextField" id="stateLabel"
52                   style="left: 55px; top: 168px; position: absolute"
53                   text="State:"/>
54                <webuijsf:textField
55                   binding="#{AddressBook.stateTextField}"
56                   id="stateTextField" maxLength="2" required="true"
57                   style="left: 120px; top: 168px; position: absolute"/>
58                <webuijsf:label for="zipTextField" id="zipLabel"
59                   style="left: 66px; top: 192px; position: absolute"
60                   text="Zip:"/>
61                <webuijsf:textField
62                   binding="#{AddressBook.zipTextField}"
63                   id="zipTextField" maxLength="5" required="true"
64                   style="left: 120px; top: 192px; position: absolute"/>
65                <webuijsf:button
66                   actionExpression="#{AddressBook.submitButton_action}"
67                   id="submitButton" primary="true" style="left: 47px;
68                      top: 240px; position: absolute; width: 95px"
69                   text="Submit"/>
70                <webuijsf:button id="clearButton" reset="true"
71                   style="left: 150px; top: 240px; position: absolute;
72                      width: 96px" text="Clear"/>
73                <webuijsf:table augmentTitle="false"
74                   binding="#{AddressBook.addressesTable}"
75                   id="addressesTable" paginateButton="true"
76                   paginationControls="true" style="left: 24px;
77                      top: 288px; position: absolute"
78                   title="Contacts" width="744">
79                   <webuijsf:tableRowGroup id="tableRowGroup1" rows="5"
80                      sourceData="#{AddressBook.addressesDataProvider}"
81                      sourceVar="currentRow">
82                      <webuijsf:tableColumn headerText="First name"
83                         id="firstNameColumn"
84                         sort="ADDRESSES.FIRSTNAME">
85                         <webuijsf:staticText id="staticText1"
86                            text="#{currentRow.value[
87                               'ADDRESSES.FIRSTNAME']}"/>
88                      </webuijsf:tableColumn>
89                      <webuijsf:tableColumn headerText="Last name"
90                         id="lastNameColumn" sort="ADDRESSES.LASTNAME">
91                         <webuijsf:staticText id="staticText2"
92                            text="#{currentRow.value[
93                               'ADDRESSES.LASTNAME']}"/>
94                      </webuijsf:tableColumn>
95                      <webuijsf:tableColumn headerText="Street"
96                         id="streetColumn" sort="ADDRESSES.STREET">
97                         <webuijsf:staticText id="staticText3"
98                            text="#{currentRow.value[
99                               'ADDRESSES.STREET']}"/>
100                     </webuijsf:tableColumn>
101                     <webuijsf:tableColumn headerText="City"
102                        id="cityColumn" sort="ADDRESSES.CITY">
103                        <webuijsf:staticText id="staticText4"
104                           text="#{currentRow.value[
105                              'ADDRESSES.CITY']}"/>
106                     </webuijsf:tableColumn>
107                     <webuijsf:tableColumn headerText="State"
108                        id="stateColumn" sort="ADDRESSES.STATE">
109                        <webuijsf:staticText id="staticText5"
110                           text="#{currentRow.value[
111                              'ADDRESSES.STATE']}"/>
112                     </webuijsf:tableColumn>
113                     <webuijsf:tableColumn headerText="zip"
114                        id="zipColumn" sort="ADDRESSES.ZIP">
115                        <webuijsf:staticText id="staticText6"
116                           text="#{currentRow.value[
117                              'ADDRESSES.ZIP']}"/>
118                     </webuijsf:tableColumn>
119                  </webuijsf:tableRowGroup>
```

```
                    </webuijsf:table>
                    <webuijsf:messageGroup id="messageGroup1"
                       showGlobalOnly="true"
                       style="left: 264px; top: 72px; position: absolute"/>
                 </webuijsf:form>
              </webuijsf:body>
           </webuijsf:html>
        </webuijsf:page>
     </f:view>
  </jsp:root>
```

(a) Adicionando um novo contato ao banco de dados



(b) Página 2 de **Table** mostrando a nova entrada do banco de dados

Figura 30.6 | `AddressBook` JSP com um formulário de adição e um componente JSF **Table**.

As linhas 18–72 contêm os componentes JSF para o formulário que coleta a entrada do usuário. As linhas 73–120 definem o elemento **Table** (`webuijsf:table`) que exibe informações de endereço do banco de dados. **Tables** podem ter vários grupos de linhas que exibem dados diferentes. Esse elemento **Table** tem um `webuijsf:tableRowGroup` simples com um tag inicial nas linhas 79–81. O atributo `sourceData` do grupo de linhas está vinculado ao nosso `addressesDataProvider` no bean de página e recebeu o nome da variável `currentRow`. O grupo de linhas também define as colunas do elemento **Table**. Cada elemento `webuijsf:tableColumn` (por exemplo, as linhas 82–88) contém um elemento `webuijsf:staticText` com seu atributo `text` vinculado a uma coluna no provedor de dados `currentRow`. Esses elementos `webuijsf:staticText` permitem que **Table** exiba os dados de cada linha.

Bean de sessão para o aplicativo AddressBook

A Figura 30.7 exibe o arquivo `SessionBean1.java` gerado por NetBeans para o aplicativo `AddressBook`. O `CachedRowSet` que o provedor de dados do componente **Table** utiliza para acessar o banco de dados `AddressBook` é uma propriedade dessa classe (linhas 31–41).

O método `_init` (linhas 13–29) configura `addressesRowSet` para interagir com o banco de dados `AddressBook`. As linhas 15–16 conectam a linha configurada como o banco de dados. As linhas 17–27 configuram o comando SQL do `addressesRowSet` como a consulta definida na Figura 30.5. A linha 28 configura o nome da tabela `RowSet`.

```
1  // Figura 30.7: SessionBean1.java
2  // O bean de sessão inicializa a origem de dados para o
3  // banco de dados AddressBook.
4  package addressbook;
5
6  import com.sun.rave.web.ui.appbase.AbstractSessionBean;
7  import com.sun.sql.rowset.CachedRowSetXImpl;
8  import javax.faces.FacesException;
9
10 public class SessionBean1 extends AbstractSessionBean
11 {
12    // Definição de componente gerenciado
13    private void _init() throws Exception
14    {
15       addressesRowSet.setDataSourceName(
16          "java:comp/env/jdbc/TEST_ApacheDerby" );
17       addressesRowSet.setCommand(
18          "SELECT ALL " +
19             "TEST.ADDRESSES.FIRSTNAME, " +
20             "TEST.ADDRESSES.LASTNAME, " +
21             "TEST.ADDRESSES.STREET, " +
22             "TEST.ADDRESSES.CITY, " +
23             "TEST.ADDRESSES.STATE, " +
24             "TEST.ADDRESSES.ZIP " +
25             "FROM TEST.ADDRESSES " +
26          "ORDER BY TEST.ADDRESSES.LASTNAME ASC, " +
27             "TEST.ADDRESSES.FIRSTNAME ASC" );
28       addressesRowSet.setTableName( "ADDRESSES" );
29    } // fim do método _init
30
31    private CachedRowSetXImpl addressesRowSet = new CachedRowSetXImpl();
32
33    public CachedRowSetXImpl getAddressesRowSet()
34    {
35       return addressesRowSet;
36    } // fim do método getAddressesRowSet
37
38    public void setAddressesRowSet( CachedRowSetXImpl crsxi )
39    {
40       this.addressesRowSet = crsxi;
41    } // fim do método setAddressesRowSet
42
43    // Para economizar espaço, omitimos o código nas linhas 42-88. O código-fonte
44    // completo é fornecido com os exemplos deste capítulo.
89 } // fim da classe SessionBean1
```

Figura 30.7 | O bean de sessão inicializa a origem de dados para o banco de dados `AddressBook`.

30.2.2 Modificando o arquivo Page Bean para o aplicativo AddressBook

Depois de construir a página Web e configurar os componentes utilizados neste exemplo, dê um clique duplo no botão **Submit** a fim de criar seu handler de evento de ação (`submitButton_action`, linhas 179–223) no arquivo de bean de página (Figura 30.8). O código para inserir um contato no banco de dados é colocado nesse método. A Figura 30.8 mostra o bean de página com o handler de evento pronto.

```
1  // Figura 30.8: AddressBook.java
2  // Bean de página para adicionar um contato ao catálogo de endereços.
3  package addressbook;
4
5  import com.sun.data.provider.RowKey;
6  import com.sun.data.provider.impl.CachedRowSetDataProvider;
7  import com.sun.rave.web.ui.appbase.AbstractPageBean;
```

```
 8  import com.sun.webui.jsf.component.Table;
 9  import com.sun.webui.jsf.component.TextField;
10  import javax.faces.FacesException;
11
12  public class AddressBook extends AbstractPageBean
13  {
14     // Definição de componente gerenciado
15     private void _init() throws Exception
16     {
17        addressesTable.setInternalVirtualForm( true );
18        addressesDataProvider.setCachedRowSet(
19           ( javax.sql.rowset.CachedRowSet ) getValue(
20              "#{SessionBean1.addressesRowSet}" ) );
21     } // fim do método _init
22
23     private Table addressesTable = new Table();
24
25     public Table getAddressesTable()
26     {
27        return addressesTable;
28     } // fim do método getAddressesTable
29
30     public void setAddressesTable( Table t )
31     {
32        this.addressesTable = t;
33     } // fim do método setAddressesTable
34
35     private CachedRowSetDataProvider addressesDataProvider =
36        new CachedRowSetDataProvider();
37
38     public CachedRowSetDataProvider getAddressesDataProvider()
39     {
40        return addressesDataProvider;
41     } // fim do método getAddressesDataProvider
42
43     public void setAddressesDataProvider( CachedRowSetDataProvider crsdp )
44     {
45        this.addressesDataProvider = crsdp;
46     } // fim do método setAddressesDataProvider
47
48     // Para economizar espaço, omitimos o código nas linhas 48-149. O
49     // código-fonte completo é fornecido com os exemplos deste capítulo.
50
150    @Override
151    public void prerender()
152    {
153       addressesDataProvider.refresh();
154    } // fim do método prerender
155
156    @Override
157    public void destroy()
158    {
159       addressesDataProvider.close();
160    } // fim do método destroy
161
162    // Para economizar espaço, omitimos o código nas linhas 162-176. O
163    // código-fonte completo é fornecido com os exemplos deste capítulo.
164
177    // o handler de ação que adiciona um contato ao banco de dados AddressBook
178    // quando o usuário clicar em submit
179    public String submitButton_action()
180    {
181       // verifica se uma linha pode ser acrescentada
182       if (addressesDataProvider.canAppendRow() )
183       {
184          try
185          {
186             // acrescenta nova linha e move o cursor para essa linha
187             RowKey rk = addressesDataProvider.appendRow();
```

```
188                addressesDataProvider.setCursorRow( rk );
189
190                // configura valores para as colunas da nova linha
191                addressesDataProvider.setValue( "ADDRESSES.FIRSTNAME",
192                   firstNameTextField.getValue() );
193                addressesDataProvider.setValue( "ADDRESSES.LASTNAME",
194                   lastNameTextField.getValue() );
195                addressesDataProvider.setValue( "ADDRESSES.STREET",
196                   streetTextField.getValue() );
197                addressesDataProvider.setValue( "ADDRESSES.CITY",
198                   cityTextField.getValue() );
199                addressesDataProvider.setValue( "ADDRESSES.STATE",
200                   stateTextField.getValue() );
201                addressesDataProvider.setValue( "ADDRESSES.ZIP",
202                   zipTextField.getValue() );
203
204                // confirma as alterações no banco de dados
205                addressesDataProvider.commitChanges();
206
207                // redefine campos de texto
208                firstNameTextField.setValue( "" );
209                lastNameTextField.setValue( "" );
210                streetTextField.setValue( "" );
211                cityTextField.setValue( "" );
212                stateTextField.setValue( "" );
213                zipTextField.setValue( "" );
214             } // fim do try
215             catch ( Exception ex )
216             {
217                error( "The address book was not updated. " +
218                   ex.getMessage() );
219             } // fim do catch
220          } // fim do if
221
222          return null;
223       } // fim do método submitButton_action
224    } // fim da classe AddressBook
```

Figura 30.8 | Bean de página para adicionar um contato ao catálogo de endereços.

A linha 17 no método `_init` configura a propriedade `internalVirtualForm` do componente **Table** como `true`. Isso impede que os botões de controle de paginação no componente **Table** enviem os **Text Fields** do formulário toda vez que o usuário quiser examinar o grupo de contatos seguinte. As linhas 18–20 configuram o `CachedRowSet` do `addressesDataProvider`. Desse modo, as linhas de tabela podem ser preenchidas com o conteúdo do banco de dados. O `CachedRowSet` é obtido a partir do bean de sessão.

As linhas 179–223 contêm o código de tratamento do evento para o botão **Submit**. A linha 182 determina se uma nova linha pode ser acrescentada ao provedor de dados. Se puder, acrescentamos uma nova linha (linha 187). Cada linha em um `CachedRowSetDataProvider` tem sua própria tecla; o método **`appendRow`** retorna a chave para a nova linha como um objeto `RowKey`. A linha 188 configura o cursor do provedor de dados como a nova linha, para que qualquer alteração que fizermos no provedor de dados afete essa linha. As linhas 191–202 configuram cada uma das colunas da linha como os valores que o usuário inseriu nos **Text Fields** correspondentes. A linha 205 armazena o novo contato chamando o método **`commitChanges`** da classe `CachedRowSetDataProvider` para inserir a nova linha no banco de dados `AddressBook`.

As linhas 208–213 limpam os **Text Fields** do formulário. Se essas linhas forem omitidas, os campos conservarão seus valores atuais depois que o banco de dados for atualizado e a página recarregada. Além disso, o botão **Clear** não funcionará corretamente se os **Text Fields** não forem limpados. Em vez de limpar os **Text Fields**, o código os redefine com os valores que eles continham na última vez que o formulário foi enviado.

As linhas 215–219 capturam qualquer exceção que possa ocorrer ao atualizar o banco de dados `AddressBook`. Se ocorrer uma exceção, as linhas 217–218 exibem uma mensagem no componente `MessageGroup` da página indicando que o banco de dados não foi atualizado e mostrando a mensagem de erro da exceção.

No método `prerender`, adicionamos a linha 153 que chama o método `CachedRowSetDataProvider` **`refresh`** para reexecutar a instrução SQL do `CachedRowSet` empacotado e classifica novamente as linhas de **Table** para que a nova linha seja exibida na ordem correta. Se você não chamar `refresh`, o novo endereço é exibido no fim do componente **Table** (já que acrescentamos a nova linha no fim do provedor de dados). O IDE gerou automaticamente o código para liberar os recursos utilizados pelo provedor de dados (linha 159) no método `destroy`. Execute o aplicativo para testar sua funcionalidade.

30.3 Componentes JSF compatíveis com o Ajax

O termo **Ajax** — abreviação de **Asynchronous JavaScript and XML** — foi cunhado por Jesse James Garrett da Adaptive Path, Inc., em fevereiro de 2005 a fim de descrever uma série de tecnologias para desenvolver aplicativos Web altamente responsivos e dinâmicos. Os aplicativos Ajax incluem o Google Maps, o Yahoo's FlickR e muitos outros. O Ajax cria uma separação entre a parte da interação com usuário de um aplicativo e sua interação com o servidor, permitindo que ambos avancem paralelamente. Isso permite que os aplicativos Ajax baseados na Web executem em velocidades que se aproximam daquelas dos aplicativos desktop, reduzindo ou eliminando a vantagem de desempenho que os aplicativos desktop tinham tradicionalmente sobre os aplicativos baseados na Web. Isso tem enormes ramificações para a indústria de aplicativos desktop — a plataforma de aplicativos recomendada está começando a mudar de desktop para a Web. Muitas pessoas acreditam que a Web — especialmente no contexto de softwares de código-fonte aberto abundante, computadores baratos e crescente largura de banda de Internet — criará a próxima principal fase de crescimento para empresas de Internet.

O Ajax faz chamadas assíncronas ao servidor para trocar pequenas quantidades de dados a cada chamada. Onde normalmente a página inteira seria enviada e recarregada com cada interação de usuário em uma página Web, o Ajax permite que somente as partes necessárias da página sejam recarregadas, poupando tempo e recursos.

Em geral, os aplicativos Ajax utilizam tecnologias de scripting do lado do cliente, como o JavaScript para interagir com elementos de página. Eles utilizam o **objeto `XMLHttpRequest`** do navegador para realizar as trocas assíncronas com o servidor Web que tornam os aplicativos Ajax tão responsivos. Esse objeto pode ser utilizado pela maioria das linguagens de criação de scripts para passar dados XML do cliente para o servidor e processar dados XML enviados do servidor de volta para o cliente.

Usar tecnologias Ajax em aplicativos Web pode aprimorar muito o desempenho, mas a programação em Ajax é complexa e propensa a erros. Isso exige que os designers de página conheçam tanto a linguagem de criação de script como de marcação. As **bibliotecas do Ajax** — como o Dojo Toolkit utilizado por JSF — permitem obter os benefícios do Ajax sem o trabalho de escrever o Ajax "bruto". Essas bibliotecas fornecem elementos de página compatíveis com o Ajax que podem ser incluídos em páginas Web simplesmente adicionando tags definidos pela biblioteca à marcação da página. Limitamos nossa discussão sobre a criação aplicativos Ajax ao uso de componentes JSF Woodstock no NetBeans.

Aplicativos Web tradicionais

A Figura 30.9 apresenta as típicas interações entre o cliente e o servidor em um aplicativo Web tradicional, como o que utiliza um formulário de registro de usuário. O usuário primeiro preenche os campos do formulário e depois envia o formulário (Figura 30.9, *Passo 1*). O navegador gera uma solicitação ao servidor, que recebe a solicitação e a processa (*Passo 2*). O servidor gera e envia uma resposta que contém a página exata que o navegador exibirá (*Passo 3*), que faz o navegador carregar a nova página (*Passo 4*) e janela de navegador, temporariamente em branco. Observe que o cliente *espera* o servidor responder e *recarrega a página inteira* com os dados da resposta (*Passo 4*). Enquanto essa **solicitação síncrona** estiver sendo processada no servidor, o usuário não pode interagir com a página Web cliente. Períodos de espera longos e frequentes, possivelmente por causa do congestionamento da internet, levaram alguns usuários a chamar a World Wide Web como "Word Wide Wait". Se o usuário interagir com outro formulário e enviá-lo, o processo inicia novamente (*Passos 5–8*).

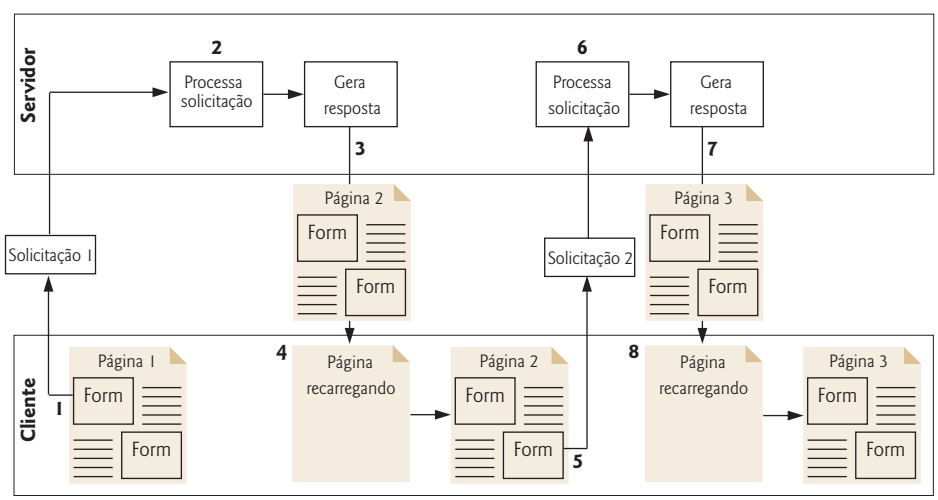


Figura 30.9 | O clássico aplicativo Web que recarrega a página para cada interação de usuário.

Esse modelo foi originalmente projetado para uma rede de documentos de hipertexto — o que algumas pessoas chamam "rede brochura". À medida que a rede se desenvolvia em uma plataforma de aplicativos em escala completa, o modelo mostrado na Figura 30.9 fazia com que o aplicativo apresentasse um desempenho muito inconstante. Toda atualização de página inteira exigia que usuários restabelecessem seu entendimento do conteúdo da página inteira. Os usuários começaram a exigir um modelo que produzisse a sensação responsiva dos aplicativos desktop.

Aplicativos Web baseados em Ajax

Os aplicativos Ajax adicionam uma camada entre o cliente e o servidor para gerenciar a comunicação entre os dois (Figura 30.10). Quando o usuário interage com a página, o cliente cria um objeto `XMLHttpRequest` para gerenciar uma solicitação (*Passo 1*). O objeto `XMLHttpRequest` envia a solicitação ao servidor (*Passo 2*) e espera a resposta. As solicitações são **assíncronas**, portanto, o usuário pode continuar a interação com o aplicativo no lado do cliente enquanto o servidor processa a solicitação anterior concorrentemente. Outras interações de usuário podem resultar em solicitações adicionais ao servidor (*Passos 3* e *4*). Assim que o servidor responde à solicitação original (*Passo 5*), o objeto `XMLHttpRequest` que emitiu a solicitação chama uma função de lado do cliente para processar os dados retornados pelo servidor. Essa função — conhecida como **função de retorno de chamada** — utiliza **atualizações de página parciais** (*Passo 6*) para exibir os dados na página Web existente *sem recarregar a página inteira*. Ao mesmo tempo, o servidor pode responder à segunda solicitação (*Passo 7*) e o lado do cliente pode começar a fazer outra atualização de página parcial (*Passo 8*). A função de retorno de chamada atualiza apenas uma parte designada da página. Essas atualizações de página parciais ajudam a tornar os aplicativos Web mais responsivos, fazendo-os parecer mais com aplicativos desktop. O aplicativo Web não carrega uma nova página enquanto o usuário interage com ela.

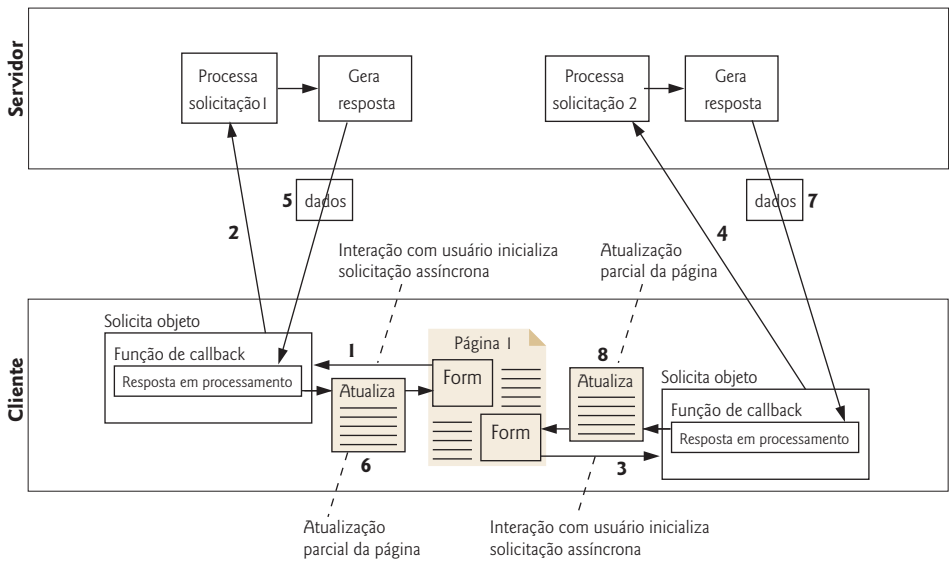


Figura 30.10 | Aplicativo Web compatível com Ajax que interage com o servidor assincronamente.

30.4 Criando um Text Field de autocompletamento e utilizando formulários virtuais

Agora modificamos o aplicativo `AddressBook` para fornecer um formulário de pesquisa que permite ao usuário localizar um contato e exibir suas informações. Tiramos proveito das capacidades predefinidas do Ajax do componente **Text Field** para fornecer a funcionalidade autocompletar — o **Text Field** fornece uma lista de sugestões à medida que o usuário digita. Essa funcionalidade obtém as sugestões de uma origem de dados, como um banco de dados ou serviço Web. Por fim, o aplicativo atualizado permitirá aos usuários pesquisar o catálogo de endereços por sobrenome. Se o usuário selecionar um contato, o aplicativo exibirá as informações do contato em um componente **Text Area**.

Adicionando componentes de pesquisa à página `AddressBook.jsp`

Utilizando o aplicativo `AddressBook` da Seção 30.2, adicione um componente **Static Text** chamado `searchText` abaixo da `addressesTable`. Mude seu texto para `"Search the address book by last name:"` e mude o tamanho da fonte para `18px`. Agora arraste um componente **Text Field** para página e nomeie-o `lastNameSearchTextField`. Configure as propriedades `required` e `autoComplete` desse campo como `true`. Adicione um **Label** chamado `lastNameSearchLabel` contendo o texto `"Last name:"` à esquerda do **Text Field** e o associe ao **Text Field**. Adicione um botão chamado `searchButton` com o texto `Find` à direita do **Text Field** e uma **Text Area** chamado `searchTextArea` à direita do botão. Dê um clique duplo no botão para criar seu handler de evento. Além disso, adicione atributos de vinculação ao bean de página ao **Text Field** e **Text Area** (clique com o botão direito do mouse em cada componente e selecione **Add Binding Attribute**).

30.4.1 Configurando formulários virtuais

Os **formulários virtuais** são utilizados quando você quer que o botão envie um subconjunto de campos de entrada da página para o servidor. Lembre-se de que anteriormente ativamos os formulários virtuais internos de **Table** para que a operação de clicar nos botões de paginação não enviasse os dados nos **Text Fields** utilizados para adicionar um contato ao banco de dados `AddressBook`. Os formulários virtuais são especialmente úteis para exibir vários formulários na mesma página. Eles permitem especificar um componente **submissor** e um ou vários componentes **participantes** para um formulário. Quando o componente submissor do formulário virtual é clicado, apenas os valores

dos seus componentes participantes são enviados para o servidor. Utilizamos formulários virtuais no nosso aplicativo AddressBook para criar uma separação entre o formulário para adicionar um contato, de um lado, e o formulário para pesquisar o banco de dados, de outro.

Para adicionar formulários virtuais à página, clique com o botão direito do mouse no botão **Submit** do formulário superior e selecione **Configure Virtual Forms...** para exibir o diálogo **Configure Virtual Forms**. Clique em **New** para adicionar um formulário virtual e, então, clique na coluna **Name** e mude o nome do novo formulário para `addForm`. Dê um clique duplo na célula abaixo da coluna **Submit** e mude opção para **Yes** a fim de indicar que esse botão deve ser utilizado para enviar o formulário virtual `addForm`. Clique em **OK** para fechar o diálogo. Depois, mantenha a tecla *Ctrl* e clique em cada um dos **Text Fields** utilizados para inserir informações de um contato no formulário superior. Clique com o botão direito do mouse um dos **Text Fields** selecionados e escolha **Configure Virtual Forms...**. Dê um clique duplo na célula abaixo da coluna **Participate** do `addForm` para mudar a opção para **Yes**, indicando que os valores nesses **Text Fields** devem ser enviados ao servidor quando o formulário for submetido. Clique em **OK** para fechar.

Repita o processo descrito para criar um segundo formulário virtual chamado `searchForm` para o formulário inferior. O `Button` **Find** deve enviar o `searchForm`, e `lastNameSearchTextField` e `searchTextArea` devem participar do `searchForm`. A Figura 30.11 mostra o diálogo **Configure Virtual Forms** depois que ambos os formulários virtuais foram adicionados.

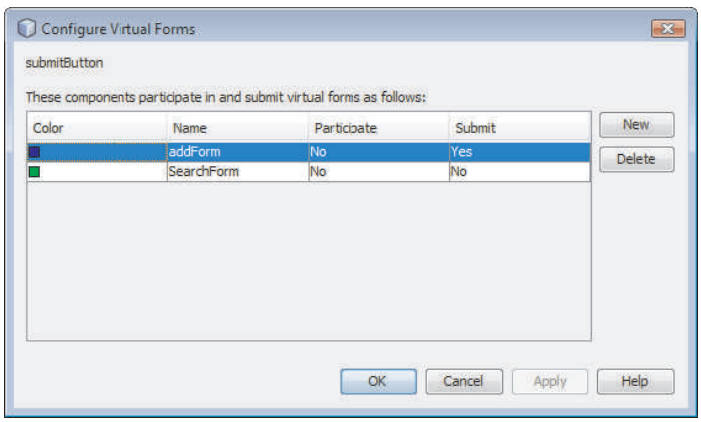


Figura 30.11 | Caixa de diálogo **Configure Virtual Forms**.

Depois, volte ao modo **Design** e clique no botão **Show Virtual Forms** ( ) na parte superior do painel Visual Designer para exibir uma legenda dos formulários virtuais na página. Os formulários virtuais devem ser configurados como os da Figura 30.12. Os **Text Fields** delineados no azul participam do formulário virtual `addForm`. Os delineados em verde participam do formulário virtual `searchForm`. Os componentes delineados com uma linha tracejada enviam seus respectivos formulários. Uma chave colorida é fornecida à direita da área **Design** para que você saiba os componentes que pertencem a cada formulário virtual.

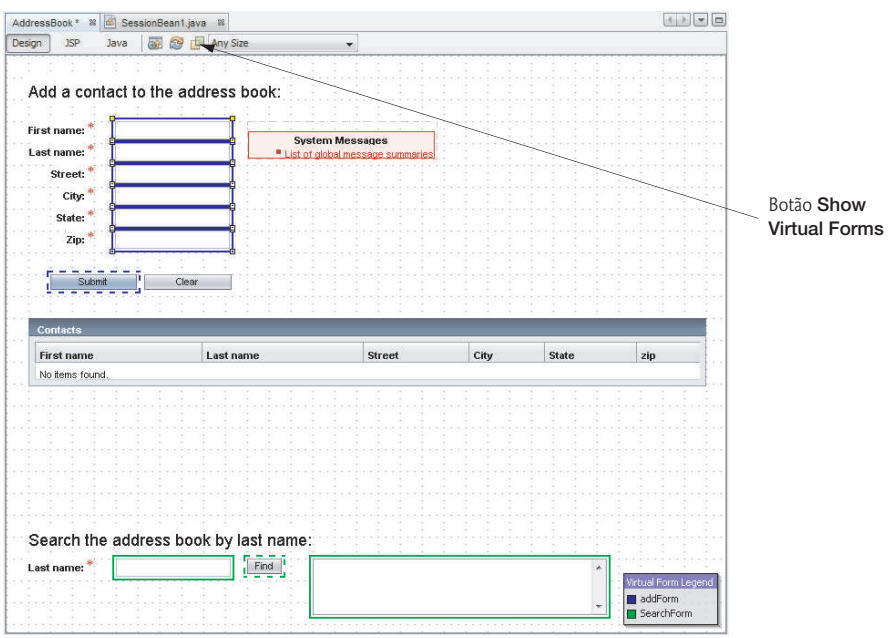


Figura 30.12 | Legenda de formulários virtuais.

30.4.2 Arquivo JSP com formulários virtuais e um Text Field de autocompletamento

A Figura 30.13 apresenta o arquivo JSP gerado por NetBeans para essa etapa do aplicativo `AddressBook`. Focalizamos apenas os novos recursos dessa JSP. Para ativar as capacidades do Ajax, você deve adicionar o atributo `webuiJsfx="true"` ao tag inicial do elemento `webuijsf:head` (como mostrado na linha 13).

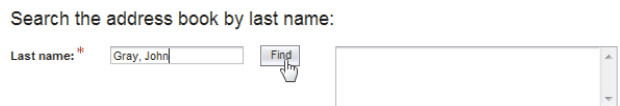
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Figura 30.13: AddressBook.jsp -->
3  <!-- AddressBook JSP com um formulário de adição e um componente Table JSF. -->
4  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:jsp="http://java.sun.com/JSP/Page"
7     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
8  <jsp:directive.page contentType="text/html;charset=UTF-8"
9     pageEncoding="UTF-8"/>
10    <f:view>
11       <webuijsf:page id="page1">
12          <webuijsf:html id="html1">
13             <webuijsf:head id="head1" webuiJsfx="true">
14                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
15             </webuijsf:head>
16             <webuijsf:body id="body1" style="-rave-layout: grid">
17                <webuijsf:form id="form1" virtualFormsConfig=
18                   "addForm | firstNameTextField lastNameTextField
19                   streetTextField cityTextField stateTextField
20                   zipTextField | submitButton ,
21                   SearchForm | lastNameSearchTextField searchTextArea |
22                   searchButton">
23                   <webuijsf:staticText id="instructionText"
24                      style="font-size: 18px; left: 24px; top: 24px;
25                         position: absolute"
26                      text="Add a contact to the address book:"/>
27                   <webuijsf:label for="firstNameTextField"
28                      id="firstNameLabel" style="left: 24px; top: 72px;
29                         position: absolute" text="First name:"/>
30                   <webuijsf:textField
31                      binding="#{AddressBook.firstNameTextField}"
32                      id="firstNameTextField" maxLength="30"
33                      required="true" style="left: 120px; top: 72px;
34                         position: absolute"/>
35                   <webuijsf:label for="lastNameTextField"
36                      id="lastNameLabel" style="left: 25px; top: 96px;
37                         position: absolute" text="Last name:"/>
38                   <webuijsf:textField
39                      binding="#{AddressBook.lastNameTextField}"
40                      id="lastNameTextField" maxLength="30" required="true"
41                      style="left: 120px; top: 96px; position: absolute"/>
42                   <webuijsf:label for="streetTextField" id="streetLabel"
43                      style="left: 49px; top: 120px; position: absolute"
44                      text="Street:"/>
45                   <webuijsf:textField
46                      binding="#{AddressBook.streetTextField}"
47                      id="streetTextField" maxLength="150" required="true"
48                      style="left: 120px; top: 120px; position: absolute"/>
49                   <webuijsf:label for="cityTextField" id="cityLabel"
50                      style="left: 62px; top: 144px; position: absolute"
51                      text="City:"/>
52                   <webuijsf:textField
53                      binding="#{AddressBook.cityTextField}"
54                      id="cityTextField" maxLength="30" required="true"
55                      style="left: 120px; top: 144px; position: absolute"/>
56                   <webuijsf:label for="stateTextField" id="stateLabel"
57                      style="left: 55px; top: 168px; position: absolute"
58                      text="State:"/>
59                   <webuijsf:textField
60                      binding="#{AddressBook.stateTextField}"
61                      id="stateTextField" maxLength="2" required="true"
62                      style="left: 120px; top: 168px; position: absolute"/>
```

```
63                   <webuijsf:label for="zipTextField" id="zipLabel"
64                      style="left: 66px; top: 192px; position: absolute"
65                      text="Zip:"/>
66                   <webuijsf:textField
67                      binding="#{AddressBook.zipTextField}"
68                      id="zipTextField" maxLength="5" required="true"
69                      style="left: 120px; top: 192px; position: absolute"/>
70                   <webuijsf:button
71                      actionExpression="#{AddressBook.submitButton_action}"
72                      id="submitButton" primary="true" style="left: 47px;
73                         top: 240px; position: absolute; width: 95px"
74                      text="Submit"/>
75                   <webuijsf:button id="clearButton" reset="true"
76                      style="left: 150px; top: 240px; position: absolute;
77                         width: 96px" text="Clear"/>
78                   <webuijsf:table augmentTitle="false"
79                      binding="#{AddressBook.addressesTable}"
80                      id="addressesTable" paginateButton="true"
81                      paginationControls="true" style="left: 24px;
82                         top: 288px; position: absolute"
83                      title="Contacts" width="744">
84                      <webuijsf:tableRowGroup id="tableRowGroup1" rows="5"
85                         sourceData="#{AddressBook.addressesDataProvider}"
86                         sourceVar="currentRow">
87                         <webuijsf:tableColumn headerText="First name"
88                            id="firstNameColumn"
89                            sort="ADDRESSES.FIRSTNAME">
90                            <webuijsf:staticText id="staticText1"
91                               text="#{currentRow.value[
92                                  'ADDRESSES.FIRSTNAME']}"/>
93                         </webuijsf:tableColumn>
94                         <webuijsf:tableColumn headerText="Last name"
95                            id="lastNameColumn" sort="ADDRESSES.LASTNAME">
96                            <webuijsf:staticText id="staticText2"
97                               text="#{currentRow.value[
98                                  'ADDRESSES.LASTNAME']}"/>
99                         </webuijsf:tableColumn>
100                        <webuijsf:tableColumn headerText="Street"
101                           id="streetColumn" sort="ADDRESSES.STREET">
102                           <webuijsf:staticText id="staticText3"
103                              text="#{currentRow.value[
104                                 'ADDRESSES.STREET']}"/>
105                        </webuijsf:tableColumn>
106                        <webuijsf:tableColumn headerText="City"
107                           id="cityColumn" sort="ADDRESSES.CITY">
108                           <webuijsf:staticText id="staticText4"
109                           text="#{currentRow.value[
110                              'ADDRESSES.CITY']}"/>
111                        </webuijsf:tableColumn>
112                        <webuijsf:tableColumn headerText="State"
113                           id="stateColumn" sort="ADDRESSES.STATE">
114                           <webuijsf:staticText id="staticText5"
115                           text="#{currentRow.value[
116                              'ADDRESSES.STATE']}"/>
117                        </webuijsf:tableColumn>
118                        <webuijsf:tableColumn headerText="zip"
119                           id="zipColumn" sort="ADDRESSES.ZIP">
120                           <webuijsf:staticText id="staticText6"
121                              text="#{currentRow.value[
122                                 'ADDRESSES.ZIP']}"/>
123                        </webuijsf:tableColumn>
124                     </webuijsf:tableRowGroup>
125                  </webuijsf:table>
126                  <webuijsf:messageGroup id="messageGroup1"
127                     showGlobalOnly="true"
128                     style="left: 264px; top: 72px; position: absolute"/>
129                  <webuijsf:staticText id="searchText"
130                     style="font-size: 18px; left: 24px; top: 516px;
131                        position: absolute"
```

```
                        text="Search the address book by last name:"/>
                    <webuijsf:label for="lastNameSearchTextField"
                        id="lastNameSearchLabel"
                        style="left: 24px; top: 552px; position: absolute"
                        text="Last name:"/>
                    <webuijsf:textField autoComplete="true"
                        autoCompleteExpression="#{SessionBean1.getOptions}"
                        binding="#{AddressBook.lastNameSearchTextField}"
                        id="lastNameSearchTextField" required="true"
                        style="left: 120px; top: 552px; position: absolute"/>
                    <webuijsf:button
                        actionExpression="#{AddressBook.searchButton_action}"
                        id="searchButton" style="left: 263px; top: 552px;
                           position: absolute; width: 38px" text="Find"/>
                    <webuijsf:textArea
                        binding="#{AddressBook.searchTextArea}" columns="50"
                        id="searchTextArea" rows="4" style="left: 336px;
                           top: 552px; position: absolute; z-index: 500"/>
                </webuijsf:form>
            </webuijsf:body>
        </webuijsf:html>
    </webuijsf:page>
</f:view>
</jsp:root>
```

(a) O usuário começa a digitar em **Text Field** e uma lista de sobrenomes que começam com o conteúdo de **Text Field** é exibida.

(b) O usuário seleciona um nome, que então é exibido em **Text Field**.



(c) O usuário clica em **Find** para localizar os dados do contato, que então são exibidos em **Text Area**.

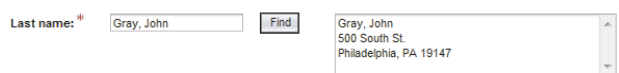


Figura 30.13 | JSP `AddressBook` com um componente **AutoComplete Text Field**.

As linhas 17–22 configuram os formulários virtuais dessa página. As linhas 137–141 definem o componente **Text Field** com a funcionalidade autocompletar. Note que o atributo **`autoComplete`** do **Text Field** é configurado como `"true"`, o que permite ao componente enviar solicitações do Ajax para o servidor. O atributo **`autoCompleteExpression`** do **Text Field** está vinculado ao método que será chamado (`SessionBean1.getOptions`) para fornecer a lista de opções que o componente **Text Field** deve sugerir. Você pode configurar essa vinculação selecionando o **Text Field** e configurando a `autoCompleteExpression` como `#{SessionBean1.getOptions}` na janela **Properties**. Discutimos como implementar o método `getOptions` na Seção 30.4.3.

30.4.3 Fornecendo sugestões para Text Field de autocompletamento

A Figura 30.14 exibe o arquivo de bean de sessão do aplicativo. Ela inclui o método **`getOptions`**, que fornece as sugestões para o **AutoComplete Text Field**. A maior parte do código nesse arquivo é idêntica ao código da Figura 30.7, portanto, discutimos apenas os recursos novos aqui.

```
1  // Figura 30.14: SessionBean1.java
2  // O bean de sessão inicializa a origem de dados do banco de dados AddressBook
3  // e fornece opções para a capacidade de autocompletar.
4  package addressbook;
5
6  import com.sun.data.provider.impl.CachedRowSetDataProvider;
7  import com.sun.rave.web.ui.appbase.AbstractSessionBean;
8  import com.sun.sql.rowset.CachedRowSetXImpl;
9  import com.sun.webui.jsf.model.AutoComplete;
10 import javax.faces.FacesException;
11 import com.sun.webui.jsf.model.Option;
```

```
12  import java.util.ArrayList;
13
14  public class SessionBean1 extends AbstractSessionBean
15     implements AutoComplete
16  {
17     // Para economizar espaço, omitimos o código nas linhas 17-85. O
18     // código-fonte completo é fornecido com os exemplos deste capítulo.
19
86     // pesquisa por sobrenomes que correspondem à string de filtro especificada
87     public Option[] getOptions( String filter )
88     {
89        // cria ArrayList para armazenar nomes correspondentes como opções
90        ArrayList< Option > list = new ArrayList< Option >();
91
92        // obtém addressesDataProvider do bean de página AddressBook
93        AddressBook addressBook = (AddressBook) getBean( "AddressBook" );
94        CachedRowSetDataProvider addressesDataProvider =
95           addressBook.getAddressesDataProvider();
96
97        try
98        {
99           // move-se para a primeira linha no banco de dados
100          boolean hasNext = addressesDataProvider.cursorFirst();
101
102          // enquanto houver registros
103          while ( hasNext )
104          {
105             // obtém um nome do banco de dados
106             String name =
107                (String) addressesDataProvider.getValue(
108                "ADDRESSES.LASTNAME" ) + ", " +
109                (String) addressesDataProvider.getValue(
110                "ADDRESSES.FIRSTNAME" );
111
112             // se o nome no banco de dados iniciar com o prefixo,
113             // adiciona-o à lista de sugestões
114             if (name.toLowerCase().startsWith( filter.toLowerCase() ) )
115             {
116                list.add( new Option( name, name ) );
117             } // fim do if
118             else
119             {
120                // termina o loop se o prefixo do filtro for
121                // alfabeticamente menor que os demais nomes
122                if ( filter.compareTo( name ) < 0 )
123                {
124                   break;
125                } // fim do if
126             } // fim de else
127
128             // move o cursor para a próxima linha do banco de dados
129             hasNext = addressesDataProvider.cursorNext();
130          } // fim do while
131       } // fim do try
132       catch ( Exception ex )
133       {
134          list.add( new Option( ex.getClass().getName(),
135             "Exception getting matching names." ) );
136       } // fim do catch
137
138       return list.toArray( new Option[ list.size() ] ); // retorna Options
139    } // fim do método getOptions
140 } // fim da classe SessionBean1
```

Figura 30.14 | O bean de sessão inicializa a origem de dados para o banco de dados `AddressBook` e fornece `Options` para a capacidade autocompletar.

O método `getOptions` (linhas 87–139) é invocado depois de cada pressionamento de tecla no AutoComplete Text Field para atualizar a lista de sugestões baseadas no que o usuário digitou. A classe em que você define `getOptions` deve implementar a interface `AutoComplete` (pacote `com.sun.webui.jsf.model`), que declara um método `getOptions` que recebe uma string (`filter`) contendo o texto que o usuário inseriu e retorna um array de `Options` (pacote `com.sun.webui.jsf.model`) a ser exibido pelo AutoComplete Text Field. O método itera pelo `addressesDataProvider`, recupera o nome de cada linha, verifica se o nome começa com as letras digitadas até o momento e, se começar, adiciona o nome à `list` (uma `ArrayList` de `Options` criadas na linha 90). As linhas 93–95 obtêm o bean `AddressBook` e o utilizam para obter o `addressesDataProvider`. A linha 100 configura o cursor como a primeira linha no provedor de dados. A linha 103 determina se há mais linhas no provedor de dados. Se houver, as linhas 106–110 recuperam os sobrenomes e nomes da linha atual e criam uma `String` no formato *sobrenome*, *nome*. A linha 114 compara as versões em letras minúsculas de `name` e `filter` para determinar se o `name` inicia com os caracteres em `filter` (isto é, os caracteres digitados até o momento). Se sim, o nome é uma correspondência, e a linha 116 o adiciona à `list`.

Lembre-se de que o provedor de dados empacota um objeto `CachedRowSet` que contém uma consulta SQL que retorna as linhas no banco de dados classificado por sobrenome e, então, por nome. Isso nos permite parar de iterar pelo provedor de dados assim que conseguirmos uma linha cujo nome vem alfabeticamente depois do texto inserido pelo usuário — os nomes nas linhas seguintes serão todos alfabeticamente maiores e, assim, não são correspondências potenciais. Se o `name` não corresponder ao texto inserido até o momento, a linha 122 testa se o nome atual é alfabeticamente maior que o `filter`. Se for, a linha 124 termina o loop.

Dica de desempenho 30.1

Ao utilizar colunas de banco de dados para fornecer sugestões no AutoComplete Text Field, classificar as colunas elimina a necessidade de verificar cada linha no banco de dados em busca de possíveis correspondências. Isso aprimora muito o desempenho do recurso autocompletar ao lidar com um banco de dados grande.

Se o nome não for uma correspondência nem for alfabeticamente maior do que `filter`, a linha 129 move o cursor para a linha seguinte no provedor de dados. Embora haja mais linhas, o loop verifica se o nome na linha seguinte corresponde ao `filter` e se deve ser adicionado à `list`.

As linhas 132–136 capturam qualquer exceção gerada ao pesquisar o banco de dados. As linhas 134–135 adicionam o texto à caixa de sugestão que indica o erro ao usuário.

Assim que o loop terminar, a linha 138 converte o `ArrayList` de `Options` em um array de `Options` e retorna o array. Os dados no array são enviados ao navegador Web cliente como a resposta à solicitação Ajax inicial e, então, utilizados para exibir uma lista drop-down das entradas de banco de dados correspondentes. Isso é tratado pelo código Dojo JavaScript que a JSF gera para exibir o componente Text Field no cliente. O usuário pode clicar em uma das entradas exibidas para selecioná-la e pressionar o botão Find para exibir as informações do contato selecionado. Discutimos o handler de evento do botão Find a seguir.

30.4.4 Exibindo as informações do contato

A Figura 30.15 exibe o arquivo de bean de página para a JSP na Figura 30.13. O método de tratamento de evento `searchButton_action` exibe as informações do contato selecionado. A maior parte do código nesse arquivo é idêntica ao código da Figura 30.8, portanto, mostramos aqui apenas os recursos novos.

```
1   // Figura 30.15: AddressBook.java
2   // Bean de página para adicionar um contato ao catálogo de endereços
3   // e exibir as informações de um contato
4   package addressbook;
5
6   import com.sun.data.provider.RowKey;
7   import com.sun.data.provider.impl.CachedRowSetDataProvider;
8   import com.sun.rave.web.ui.appbase.AbstractPageBean;
9   import com.sun.webui.jsf.component.Table;
10  import com.sun.webui.jsf.component.TextArea;
11  import com.sun.webui.jsf.component.TextField;
12  import javax.faces.FacesException;
13
14  public class AddressBook extends AbstractPageBean
15  {
16     // Para economizar espaço, omitimos o código nas linhas 16-121. O código-fonte
17     // completo é fornecido com os exemplos deste capítulo.
18
122    private TextField lastNameSearchTextField = new TextField();
123
124    public TextField getLastNameSearchTextField()
125    {
126       return lastNameSearchTextField;
```

```
127      }
128
129      public void setLastNameSearchTextField( TextField tf )
130      {
131         this.lastNameSearchTextField = tf;
132      }
133
134      private TextArea searchTextArea = new TextArea();
135
136      public TextArea getSearchTextArea()
137      {
138         return searchTextArea;
139      }
140
141      public void setSearchTextArea( TextArea ta )
142      {
143         this.searchTextArea = ta;
144      }
145
146      // Para economizar espaço, omitimos o código nas linhas 146-250. O código-fonte
147      // completo é fornecido com os exemplos deste capítulo.
148
251      // localiza e exibe as informações do contato selecionado
252      public String searchButton_action()
253      {
254         // obtém o nome selecionado e o divide em tokens
255         String name = lastNameSearchTextField.getText().toString();
256         String[] names = name.split( ", " );
257
258         // localiza a RowKey para a linha no banco de dados que contém esse nome
259         RowKey row = addressesDataProvider.findFirst(
260            new String[] { "ADDRESSES.LASTNAME", "ADDRESSES.FIRSTNAME" },
261            new String[] { names[0], names[1] } );
262
263         // move para a linha no banco de dados
264         addressesDataProvider.setCursorRow( row );
265
266         // obtém informações de contato
267         String result = name + "\n" +
268            addressesDataProvider.getValue( "ADDRESSES.STREET" ) + "\n" +
269            addressesDataProvider.getValue( "ADDRESSES.CITY" ) + ", " +
270            addressesDataProvider.getValue( "ADDRESSES.STATE" ) + " " +
271            addressesDataProvider.getValue( "ADDRESSES.ZIP" ) + "\n";
272
273         searchTextArea.setText( result ); // exibe as informações de contato
274         return null;
275      } // termina o método searchButton_action
276   } // fim da classe AddressBook
```

Figura 30.15 | O bean de página que sugere nomes no **Text Field AutoComplete**.

As linhas 252–275 definem o método `searchButton_action`. A linha 255 obtém o texto do `lastNameSearchTextField` e depois a linha 256 analisa sintaticamente o nome em um sobrenome e em um nome. Observe que o método `getText` (linha 255) retorna um `Object`, portanto, devemos invocar toString nesse `Object` para obter sua representação `String`. As linhas 259–261 utilizam o método `findFirst` do `addressDataProvider` para localizar a primeira entrada no banco de dados com o sobrenome e o nome especificados. Esse método retorna um `RowKey` que identifica exclusivamente essa linha na tabela de banco de dados. A linha 264 move o cursor para essa linha. As linhas 267–271 obtêm os dados restantes do contato e criam uma string de resultado, que é exibida na **Text Area** (linha 273).

30.5 Conclusão

Neste capítulo, apresentamos um estudo de caso sobre a criação de um aplicativo Web que interage com um banco de dados e fornece uma rica interação de usuário utilizando um componente JSF compatível com Ajax. Primeiro criamos um aplicativo `AddressBook` que permitiu que um usuário adicionasse e acessasse contatos. Você aprendeu a inserir a entrada do usuário em um banco de dados Java DB e a exibir o conteúdo de um banco de dados em uma página Web utilizando um componente JSF **Table**. Você também aprendeu que alguns dos componentes Woodstock JSF são compatíveis com Ajax. Estendemos o aplicativo `AddressBook` para incluir componente **AutoComplete Text Field**. Mostramos como utilizar um banco de dados para exibir sugestões enquanto o usuário digita no **Text Field**. Você também aprendeu a utilizar formulários virtuais para enviar subconjuntos de componentes de entrada de um formulário ao servidor para processamento. No Capítulo 31, você utilizará o NetBeans para criar serviços Web e consumi-los a partir de aplicativos desktop e aplicativos Web.

Resumo

Seção 30.2 Acessando bancos de dados em aplicativos Web

- Definir a propriedade `primary` de um **Button** como `true` (marcado) o destaca na página e permite ao usuário enviar um formulário pressionando *Enter* em vez de clicar no botão.
- Definir propriedade `reset` de um **Button** como `true` (marcado) impede a validação quando o usuário clica no botão e configura automaticamente o botão para redefinir todos os campos de entrada da página.
- O componente **Table** formata e exibe dados de tabelas de banco de dados.
- Para criar um banco de dados Java DB, na guia **Services** do NetBeans, expanda o nó **Databases** e clique com o botão direito do mouse em **Java DB** e selecione **Create Database...**. Depois de você criar um banco de dados, a guia **Services** terá uma nova entrada no nó **Databases** que mostra o URL do banco de dados.
- Você pode utilizar a guia **Services** para criar tabelas de banco de dados e executar instruções SQL que manipulam o banco de dados. O NetBeans deve ser conectado ao banco de dados para executar as instruções SQL. Se o NetBeans estiver conectado ao banco de dados, o ícone aparece ao lado do URL do banco de dados; caso contrário, o ícone aparece — nesse caso, clicar com o botão direito do mouse no ícone e selecione **Connect...**.
- Você pode visualizar os dados na tabela expandindo o nó **Tables**, clicando com o botão direito na tabela e selecionando **View Data...**.
- Para vincular uma tabela de banco de dados a um componente **Table**, arraste a tabela de banco de dados da guia **Services** e solte-a no componente **Table**.
- Para selecionar colunas específicas para a exibição, clique com o botão direito do mouse no componente **Table** e selecione **Bind to Data** para exibir o diálogo **Bind to Data** que contém a lista das colunas na tabela de banco de dados.
- Por padrão, **Table** utiliza os nomes de coluna da tabela de banco de dados com todas as letras maiúsculas como cabeçalhos. Para alterar esses cabeçalhos, selecione uma coluna no modo **Design** e edite sua propriedade `headerText` na janela **Properties**.
- Configurar a propriedade `paginationControls` de **Table** como `true` configura esse componente **Table** para a paginação automática e fornece botões para avançar e retroceder entre os registros.
- Configurar a propriedade `internalVirtualForm` de **Table** como `true` impede que os botões de paginação de **Table** enviem o formulário inteiro quando o usuário estiver navegando por páginas de registros.
- Um `CachedRowSetDataProvider` fornece um `RowSet` rolável que pode ser vinculado a um componente **Table** para exibir os dados do `RowSet`. Esse provedor de dados é um empacotador para um objeto `CachedRowSet`.
- Um componente **Message Group** exibe mensagens de sistema, como mensagens de erro quando uma tentativa de manipular um banco de dados falha. Configure a propriedade `showGlobalOnly` do **Message Group** como `true` para impedir que mensagens de erro de validação no nível do componente sejam exibidas aqui.
- Cada linha em um `CachedRowSetDataProvider` tem sua própria chave; o método `AppendRow` retorna a chave para a nova linha.
- Chamar o método `commitChanges` da classe `CachedRowSetDataProvider` atualiza o banco de dados baseado nas alterações no `CachedRowSetDataProvider`.
- O método `CachedRowSetDataProvider` `refresh` reexecuta a instrução SQL do `CachedRowSet`.

Seção 30.3 Componentes JSF compatíveis com o Ajax

- O termo Ajax — abreviação de Asynchronous JavaScript e XML — foi cunhado por Jesse James Garrett da Adaptive Path, Inc., em fevereiro de 2005 a fim de descrever uma série de tecnologias para desenvolver aplicativos Web altamente responsivos e dinâmicos.
- O Ajax cria uma separação entre a parte da interação com o usuário de um aplicativo e a sua interação com o servidor, permitindo que ambos prossigam assíncrona e paralelamente. Isso permite que aplicativos Ajax baseados na Web executem em velocidades que se aproximam daquelas de aplicativos desktop.
- O Ajax faz chamadas assíncronas ao servidor para trocar pequenas quantidades de dados a cada chamada. Onde normalmente a página inteira seria enviada e recarregada com cada interação do usuário em uma página Web, o Ajax recarrega apenas as partes necessárias da página, poupando tempo e recursos.
- Em geral, os aplicativos Ajax utilizam tecnologias de scripting do lado do cliente, como o JavaScript para interagir com elementos de página. Eles utilizam o objeto `XMLHttpRequest` do navegador para realizar as trocas assíncronas com o servidor Web que tornam os aplicativos Ajax tão responsivos.
- As bibliotecas do Ajax — como o Dojo Toolkit utilizado por JSF — permitem obter os benefícios do Ajax sem o trabalho de escrever o Ajax "bruto".
- Em um aplicativo Web tradicional, o usuário preenche campos de um formulário e envia o formulário. O navegador gera uma solicitação ao servidor, que a recebe e a processa. O servidor gera e envia uma resposta que contém a página exata que o navegador exibirá. O navegador carrega a nova página, fazendo sua janela ficar temporariamente em branco. O cliente *espera* o servidor responder e *recarrega a página inteira* com os dados da resposta. Enquanto essa solicitação síncrona estiver sendo processada no servidor, o usuário não pode interagir com a página Web. Esse modelo resulta em um desempenho inconstante do aplicativo.
- Em um aplicativo Ajax, quando o usuário interage com a página, o cliente cria um objeto `XMLHttpRequest` para gerenciar uma solicitação. Esse objeto envia a solicitação ao servidor e espera a resposta. As solicitações são assíncronas, portanto, o usuário pode interagir com o aplicativo no lado do cliente enquanto o servidor processa a solicitação anterior concorrentemente. Outras interações de usuário poderiam resultar em solicitações adicionais ao servidor. Assim que o servidor responde à solicitação original, o objeto `XMLHttpRequest` que emitiu a solicitação chama uma função do lado do cliente

para processar os dados retornados pelo servidor. Essa função de retorno de chamada utiliza atualizações de página parciais para exibir os dados na página Web existente *sem recarregar a página inteira*. Ao mesmo tempo, o servidor pode responder à segunda solicitação e o lado do cliente pode começar a fazer outra atualização de página parcial.

- As atualizações de página parciais ajudam a tornar os aplicativos Web mais responsivos, fazendo-os se comportarem mais como aplicativos desktop.

Seção 30.4 Criando um AutoComplete `Text Field` utilizando formulários virtuais

- Os formulários virtuais são utilizados quando você quer que um botão envie um subconjunto dos campos de entrada da página ao servidor e para exibir vários formulários na mesma página. Você especifica um componente submissor e um ou vários componentes participantes de um formulário. Quando o componente submissor é clicado, somente os valores dos seus componentes participantes serão enviados ao servidor.
- Para adicionar formulários virtuais à página, clique com o botão direito do mouse Clique em **New** para adicionar um formulário virtual, então clique na coluna **Name** e mude o nome do novo formulário. Dê um clique duplo na célula abaixo da coluna **Submit** e mude a opção para **Yes** para indicar que esse botão deve enviar o formulário virtual. Clique em **OK** para fechar o diálogo. Depois, mantenha a tecla *Ctrl* pressionada e clique em cada componente que deve ser enviado como parte do formulário virtual. Clique com o botão direito do mouse em um dos componentes selecionados e escolha **Configure Virtual Forms...**. Dê um clique duplo na célula abaixo da coluna **Participate** do formulário virtual para mudar a opção para **Yes**, indicando que os valores nesses componentes devem ser enviados ao servidor quando o formulário for enviado. Clique em **OK** para fechar.
- No modo **Design**, clique no botão **Show Virtual Forms** ( ) para exibir uma legenda dos formulários virtuais na página. Os componentes delineados com uma linha tracejada enviam seus respectivos formulários. Uma chave colorida à direita da área **Design** mostra os componentes que pertencem a cada formulário virtual.
- Configure o atributo `autoComplete` de um **Text Field** como `"true"` para permitir que o componente envie solicitações Ajax ao servidor. O atributo `autoCompleteExpression` do **Text Field** é vinculado ao método que será chamado para fornecer a lista de opções que o componente **Text Field** deve sugerir.
- O método `getOptions` é invocado depois de cada pressionamento de tecla no **AutoComplete Text Field** para atualizar a lista de sugestões baseadas no que o usuário digitou. A classe em que você define `getOptions` deve implementar a interface `AutoComplete` (pacote `com.sun.webui.jsf.model`), que declara o método `getOptions`, recebe uma string que contém o texto que o usuário inseriu e retorna um array de `Options` (pacote `com.sun.webui.jsf.model`) para ser exibido pelo **Text Field** autocompletar.

Terminologia

Ajax (Asynchronous JavaScript and XML), 1005
`appendRow`, método da classe `CachedRowSetDataProvider`, 1004
atualização de página parcial, 1006
`autoComplete`, atributo de um **Text Field**, 1010
`autoCompleteExpression`, atributo de um **Text Field**, 1010
bibliotecas do Ajax, 1005
`CachedRowSetDataProvider`, classe, 998
`commitChanges`, método da classe `CachedRowSetDataProvider`, 1004
formulários virtuais, 1006
função de retorno de chamada, 1006
`getOptions`, método da interface `AutoComplete`, 1010
`internalVirtualForms`, propriedade de um **Table**, 998
JavaScript assíncrono e XML (Ajax), 1005
Message Group, componente JSF, 999
`paginationControls`, propriedade de um **Table**, 998
participante de um formulário virtual, 1007
`primary`, propriedade de um JSF **Button**, 996
`refresh`, método da classe `CachedRowSetDataProvider`, 1004
`reset`, propriedade de um JSF **Button**, 996
solicitação assíncrona, 1006
solicitação síncrona, 1005
submissor em um formulário virtual, 1006
Table, componente JSF, 995
`XMLHttpRequest`, objeto, 1005

Exercícios de autorrevisão

30.1 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) O componente JSF **Table** permite diagramar outros componentes e texto no formato de tabela.

b) Os formulários virtuais permitem que múltiplos formulários, cada um com o seu próprio botão **Submit**, sejam exibidos na mesma página Web.

c) Um `CachedRowSetDataProvider` é armazenado na `SessionBean` e executa consultas SQL para fornecer componentes `Table` com dados para exibição.

d) O objeto `XMLHttpRequest` fornece acesso ao objeto de solicitação de uma página.

e) Você configura o atributo `autoComplete` de um **Text Field** como `"true"` para permitir ao componente enviar solicitações Ajax ao servidor.

f) Um provedor de dados reexecuta automaticamente seu comando SQL para fornecer informações de banco de dados atualizadas a cada atualização de página.

30.2 Preencha as lacunas em cada uma das seguintes afirmações.

a) Ajax é um acrônimo de _____.

b) O método _____ da classe _____ atualiza um banco de dados para refletir qualquer alteração feita no provedor de dados do banco de dados.

c) Um formulário virtual especifica que certos componentes JSF são _____ cujos dados serão enviados quando o componente submissor for clicado.

Respostas dos exercícios de autorrevisão

30.1 a) Falso. Componentes `Table` são utilizados para exibir dados de bancos de dados. b) Verdadeiro. c) Falso. `CachedRowSetDataProvider` é uma propriedade do bean de página. Ela empacota um `CachedRowSet`, que é armazenado na `SessionBean` e executa consultas SQL. d) Falso. O objeto `XMLHttpRequest` é um objeto que permite trocas assíncronas com um servidor Web. e) Verdadeiro. f) Falso. Você deve chamar o método `refresh` no provedor de dados para reexecutar o comando SQL.

30.2 a) Asynchronous JavaScript and XML. b) `commitChanges`, `CachedRowSetDataProvider`. c) participantes.

Exercícios

30.3 (***Aplicativo de livro de visitas***) Crie uma página Web JSF que permite que os usuários assinem e visualizem um livro de visitas. Utilize o banco de dados `Guestbook` para armazenar as entradas do livro de visitas. [*Nota*: um script SQL para criar o banco de dados `Guestbook` é fornecido no diretório de exemplos deste capítulo.] O banco de dados `Guestbook` tem uma única tabela, `Messages`, que tem quatro colunas: `Data`, `Name`, `Email` e `Message`. O banco de dados já contém algumas entradas de exemplo. Na página Web, forneça Text Fields para o nome e e-mail do usuário eletrônico e um Text Area para a mensagem. Adicione um Submit Button e um componente Table e configure Table para exibir as entradas do livro de visitas. Utilize o método de handler de ação do botão Submit para inserir uma nova linha que contém a entrada do usuário e data de hoje no banco de dados `Guestbook`.

30.4 (***Modificação do aplicativo AddressBook***) Modifique o aplicativo `AddressBook` para que os usuários insiram pesquisas no Text Field autocompletar no formato *sobrenome nome*. Você precisará adicionar um novo provedor de dados (ou modificar o existente) para classificar as linhas no banco de dados `AddressBook` por nome e sobrenome.

Um cliente é para mim uma mera unidade, um fator em um problema.
— Sir Arthur Conan Doyle

Eles também servem aos que somente ficam parados esperando.
— John Milton

... se as coisas mais simples da natureza tiverem uma mensagem que você entende, alegre-se, sua alma está viva.
— Eleonora Duse

Protocolo é tudo.
— Françoise Giuliani

31

Serviços Web

Objetivos

Neste capítulo, você aprenderá:

- O que é um serviço Web.
- Como publicar e consumir serviços Web no NetBeans.
- Como a XML, JSON, Simple Object Access Protocol (SOAP) baseado em XML e a arquitetura Representational State Transfer (REST) possibilitam serviços Web Java.
- Os elementos que compreendem os serviços.
- Como criar aplicativos clientes para desktop e Web que consomem serviços Web.
- Como utilizar o rastreamento de sessão nos serviços Web para manter informações de estado de cliente.
- Como conectar-se a bancos de dados a partir de serviços Web.
- Como passar objetos de tipos definidos pelo usuário e retorná-los de um serviço Web.

Sumário

31.1 Introdução
31.2 Fundamentos do serviço Web
31.3 Simple Object Access Protocol (SOAP)
31.4 Representational State Transfer (REST)
31.5 JavaScript Object Notation (JSON)
31.6 Publicando e consumindo serviços Web baseados em SOAP
- 31.6.1 Criando um projeto de aplicativo Web e adicionando uma classe de serviço Web no NetBeans
- 31.6.2 Definindo o serviço Web `WelcomeSOAP` no NetBeans
- 31.6.3 Publicando o serviço Web `WelcomeSOAP` a partir do NetBeans
- 31.6.4 Testando o serviço Web `WelcomeSOAP` com página Web `Tester` do servidor de aplicativo GlassFish
- 31.6.5 Descrevendo um serviço Web com a Web Service Description Language (WSDL)
- 31.6.6 Criando um cliente para consumir o serviço Web `WelcomeSOAP`
- 31.6.7 Consumindo o serviço Web `WelcomeSOAP`

31.7 Publicando e consumindo serviços Web XML baseados em REST
- 31.7.1 Criando um serviço Web XML baseado em REST
- 31.7.2 Consumindo um serviço Web XML baseado em REST

31.8 Publicando e consumindo serviços Web JSON baseados em REST
- 31.8.1 Criando um serviço Web JSON baseado em REST
- 31.8.2 Consumindo um serviço Web JSON baseado em REST

31.9 Rastreamento de sessão em um serviço Web baseado em SOAP
- 31.9.1 Criando um serviço Web `Blackjack`
- 31.9.2 Consumindo o serviço Web `Blackjack`

31.10 Consumindo um serviço Web baseado em SOAP orientado a banco de dados
- 31.10.1 Criando o banco de dados `Reservation`
- 31.10.2 Criando um aplicativo Web para interagir com o serviço `Reservation`

31.11 Gerador de equação: retornando tipos definidos pelo usuário
- 31.11.1 Criando o serviço Web XML baseado em REST `EquationGenerator`
- 31.11.2 Consumindo o serviço Web XML baseado em REST `EquationGenerator`
- 31.11.3 Criando o serviço Web JSON baseado em REST `EquationGenerator`
- 31.11.4 Consumindo o serviço Web JSON baseado em REST `EquationGenerator`

31.12 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios | Fazendo a diferença

31.1 Introdução

Este capítulo introduz os serviços Web, que promovem a portabilidade e reusabilidade de softwares nos aplicativos que operam pela Internet. Um **serviço Web** é um componente de software armazenado em um computador que pode ser acessado por um aplicativo (ou outro componente de software) em outro computador por uma rede. Serviços Web se comunicam utilizando tecnologias como XML, JSON e HTTP. Neste capítulo, utilizamos duas Java APIs que facilitam os serviços Web. A primeira, **JAX-WS**, é baseada no **Simple Object Access Protocol (SOAP)** — um protocolo baseado em XML que permite a serviços Web e clientes se comunicarem, mesmo que o cliente e o serviço Web sejam escritos em linguagens diferentes. A segunda, **JAX-RS**, utiliza o **Representational State Transfer (REST)** — uma arquitetura de rede que utiliza os mecanismos de solicitação/resposta tradicionais da Web como solicitações `GET` e `POST`. Para obter informações adicionais sobre serviços Web baseados em SOAP e REST, visite nossos Web Services Resource Centers:

```
www.deitel.com/WebServices/
www.deitel.com/RESTWebServices/
```

Esses Resource Centers incluem informações sobre a criação e implementação de serviços Web em muitas linguagens e sobre os serviços Web oferecidos por empresas como Google, Amazon e eBay. Você também encontrará muitas ferramentas adicionais para publicar e consumir serviços Web. Para informações adicionais sobre serviços Web Java baseados em REST, confira o projeto Jersey:

```
jersey.dev.java.net/
```

A XML utilizada neste capítulo é criada e manipulada por meio das APIs, portanto, você não precisa conhecer os detalhes da XML para utilizá-la aqui. Se quiser aprender mais sobre a XML, leia os seguintes tutoriais:

```
www.deitel.com/articles/xml_tutorials/20060401/XMLBasics/
www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData/
```

e visite nosso XML Resource Center:

```
www.deitel.com/XML/
```

Transações de empresa para empresa

Em vez de contar com aplicativos proprietários, as empresas podem conduzir transações via serviços Web padronizados, amplamente disponíveis. Isso tem implicações importantes para **transações business-to-business (B2B)**. Serviços Web são independentes de plataforma e linguagem, permitindo que empresas colaborem sem se preocupar com a compatibilidade de hardwares, softwares e tecnologias de comunicação. Empresas, como a Amazon, Google, eBay, PayPal e muitas outras tiram proveito disponibilizando seus aplicativos no lado do servidor a parceiros via serviços Web.

Comprando alguns serviços Web e utilizando outros gratuitos que são relevantes para seus negócios, o desenvolvimento de aplicativos para as empresas pode ser mais rápido e elas podem criar novos serviços que são mais inovadores. Sites de comércio eletrônico, por exemplo, podem fornecer aos seus clientes experiências de compra aprimoradas. Considere uma loja de músicas on-line. O site Web da loja oferece links para informações sobre vários CDs, permitindo aos usuários comprá-los, conhecer artistas, localizar mais títulos desses artistas, encontrar músicas de outros artistas que talvez eles possam gostar etc. O site Web da loja também pode oferecer links para o site de uma empresa que vende entradas para shows e fornecer um serviço Web que exibe datas de futuros concertos para vários artistas, permitindo aos usuários comprar entradas. Consumindo o serviço Web de entrada para shows no seu site, a loja de músicas on-line pode fornecer um serviço adicional aos seus clientes, aumentar o tráfego do site e talvez ganhar uma comissão sobre as vendas de entrada para shows. A empresa que vende entrada para shows também se beneficia da relação comercial vendendo mais bilhetes e possivelmente recebendo receitas da loja de músicas on-line pelo uso do serviço Web.

Qualquer programador Java com conhecimento de serviços Web pode escrever aplicativos que "consomem" serviços Web. Os aplicativos resultantes invocariam serviços Web em execução nos servidores que poderiam estar a milhares de quilômetros de distância.

NetBeans

O NetBeans é uma entre várias ferramentas que permitem "publicar" e/ou "consumir" serviços Web. Demonstramos como utilizar o NetBeans para implementar serviços Web utilizando as APIs do JAX-WS e JAX-RS e como invocá-los a partir de aplicativos clientes. Para cada exemplo, fornecemos o código do serviço Web e então apresentamos um aplicativo cliente que utiliza o serviço Web. Nossos primeiros exemplos constroem serviços Web e aplicativos clientes simples no NetBeans. Demonstramos então serviços Web que utilizam recursos mais sofisticados, como manipulação de bancos de dados com o JDBC e manipulação de objetos de classes. Para obter informações sobre como fazer o download e instalar o NetBeans e o servidor GlassFish v2 UR2, consulte a Seção 29.1.

31.2 Fundamentos do serviço Web

A máquina na qual um serviço Web reside é chamada **host de serviço Web**. O aplicativo cliente envia uma solicitação por uma rede ao host do serviço Web, que processa a solicitação e retorna uma resposta pela rede ao aplicativo. Esse tipo de computação distribuída beneficia os sistemas de vários modos. Por exemplo, um aplicativo sem acesso direto a dados em outro sistema poderia ser capaz de recuperar os dados via um serviço Web. De maneira semelhante, um aplicativo que não tem capacidade de processamento suficiente para realizar cálculos específicos poderia utilizar um serviço Web para tirar vantagem dos recursos superiores de outro sistema.

No Java, um serviço Web é implementado como uma classe. Nos capítulos anteriores, todas as partes de um aplicativo residiam em uma máquina. A classe que representa o serviço Web reside em um servidor — ela não é parte do aplicativo cliente. Disponibilizar um serviço Web para receber solicitações de cliente é conhecido como **publicar um serviço Web**; utilizar um serviço Web a partir de um aplicativo cliente é conhecida como **consumir um serviço Web**.

31.3 Simple Object Access Protocol (SOAP)

O Simple Object Access Protocol (SOAP) é um protocolo independente de plataforma que utiliza a XML para fazer chamadas de procedimento remoto, geralmente sobre o HTTP. Você pode visualizar a especificação SOAP em `www.w3.org/TR/soap/`. Cada solicitação e resposta são empacotadas em uma **mensagem SOAP** — marcação XML que contém as informações que um serviço Web exige para processar a mensagem. Mensagens SOAP são escritas em XML para que sejam legíveis por computadores e humanos e para que sejam independentes de plataforma. A maioria dos **firewalls** — as barreiras de segurança que restringem a comunicação entre redes — permite que o tráfego HTTP passe para que os clientes possam navegar pela Web enviando solicitações e recebendo respostas de servidores Web. Assim, serviços baseados em SOAP podem enviar e receber mensagens SOAP por meio de conexões HTTP com poucas limitações.

O SOAP suporta um extenso conjunto de tipos. O **formato wire** utilizado para transmitir solicitações e respostas deve suportar todos os tipos passados entre os aplicativos. Tipos SOAP incluem os tipos primitivos (por exemplo, `int`), bem como `DateTime`, `XmlNode`- e outros. O SOAP também pode transmitir arrays desses tipos.

Quando um programa invoca um método de um serviço Web SOAP, a solicitação e todas as informações relevantes são empacotadas em uma mensagem SOAP colocadas em um **envelope SOAP** e enviadas ao servidor no qual o serviço Web reside. Quando o serviço Web recebe essa mensagem SOAP, ele analisa a XML que representa a mensagem e então processa o conteúdo da mensagem. A mensagem especifica o método que o cliente deseja executar e os argumentos que o cliente passou para esse método. Em seguida, o serviço Web chama o método com os argumentos especificados (se houver um) e envia a resposta de volta ao cliente em outra mensagem SOAP. O cliente analisa a resposta para recuperar o resultado do método. Na Seção 31.6, você construirá e consumirá um serviço Web SOAP básico.

31.4 Representational State Transfer (REST)

O Representational State Transfer (REST) refere-se a um estilo arquitetônico de implementar serviços Web. Esses serviços Web costumam ser chamados de **serviços Web RESTful**. Embora o próprio REST não seja um padrão, serviços Web RESTful são implementados utilizando padrões Web. Cada método em um serviço Web RESTful é identificado por um URL único. Assim, quando o servidor recebe uma solicitação, ele sabe imediatamente que operação realizar. Esses serviços Web podem ser utilizados em um programa ou diretamente de um navegador Web. Os resultados de uma determinada operação podem ser armazenados em cache localmente pelo navegador quando o serviço é invocado com uma solicitação `GET`. Isso pode tornar solicitações subsequentes à mesma operação mais rápidas carregando o resultado diretamente do cache do navegador. Os serviços Web da Amazon (`aws.amazon.com`) são RESTful, assim como vários outros.

Serviços Web RESTful são alternativas àqueles implementados com o SOAP. Diferentemente dos serviços Web baseados em SOAP, a solicitação e resposta dos serviços REST não são empacotados em envelopes. O REST também não está limitado a retornar dados no formato XML. Ele pode utilizar vários formatos, como XML, JSON, HTML, texto sem formatação e arquivos de mídia. Nas seções 31.7–31.8, você construirá e consumirá serviços Web RESTful básicos.

31.5 JavaScript Object Notation (JSON)

A **JavaScript Object Notation (JSON)** é uma alternativa à XML para representar dados. JSON é um formato de troca de dados baseado em texto utilizado para representar objetos em JavaScript como coleções de pares nome/valor representados como `Strings`. Ele é comumente utilizado em aplicativos Ajax. JSON é um formato simples que facilita a leitura, criação e análise de objetos e, como é muito menos prolixo que a XML, permite que os programas transmitam dados eficientemente pela internet. Cada objeto JSON é representado como uma lista de nomes e valores de propriedade entre colchetes, no seguinte formato:

```
{ NomeDaPropriedade1 : valor1, NomeDaPropriedade2 : valor2 }
```

Arrays são representados no JSON com colchetes no seguinte formato:

```
[ valor1, valor2, valor3 ]
```

Cada valor em um array pode ser uma string, um número, um objeto JSON, `true`, `false` ou `null`. Para apreciar a simplicidade dos dados JSON, examine esta representação de um array de entradas de catálogo de endereços:

```
[ { first: 'Cheryl', last: 'Black' },
  { first: 'James', last: 'Blue' },
  { first: 'Mike', last: 'Brown' },
  { first: 'Meg', last: 'Gold' } ]
```

Muitas linguagens de programação agora suportam o formato de dados JSON. Uma extensa lista de bibliotecas JSON classificadas por linguagem pode ser localizada em `www.json.org`.

31.6 Publicando e consumindo serviços Web baseados em SOAP

Esta seção apresenta nosso primeiro exemplo da publicação (permissão de acesso de um cliente) e consumo (utilização) de um serviço Web. Começamos com um serviço Web baseado em SOAP.

31.6.1 Criando um projeto de aplicativo Web e adicionando uma classe de serviço Web no NetBeans

Ao criar um serviço Web no NetBeans, você focaliza a lógica e deixa o IDE tratar sua infraestrutura. Para criar um serviço Web no NetBeans, você primeiro cria um **projeto Web Application**. O NetBeans utiliza esse tipo de projeto para serviços Web que são invocados por outros aplicativos.

Criando um projeto Web Application no NetBeans

Para criar um aplicativo Web, siga estes passos:

1. Escolha File> New Project... para abrir a caixa de diálogo New Project.
2. Selecione Java Web na lista Categories da caixa de diálogo e então selecione Web Application na lista Projects. Clique em Next>.

3. Especifique o nome do seu projeto (`WelcomeSOAP`) no campo **Project Name** e especifique onde você gostaria de armazenar o projeto no campo **Project Location**. Clique no botão **Browse** para selecionar a localização. Clique em **Next>**.
4. Escolha **GlassFish v2** na lista drop-down **Server** e **Java EE 5** na lista drop-down **Java EE Version**.
5. Clique em **Finish** para fechar a caixa de diálogo **New Project**.

Isso cria um aplicativo Web que executará em um navegador Web, semelhante aos projetos utilizados nos capítulos 29 e 30.

Adicionando uma classe de serviço Web a um projeto Web Application

Siga estes passos para adicionar uma classe de serviço Web ao projeto:

1. Na guia **Projects** no NetBeans, clique com o botão direito do mouse no nó **WelcomeSOAP** do projeto e escolha **New > Web Service...** para abrir a caixa de diálogo **New Web Service**.
2. Especifique `WelcomeSOAP` no campo **Web Service Name**.
3. Especifique `com.deitel.welcomesoap` no campo **Package**.
4. Clique em **Finish** para fechar a caixa de diálogo **New Web Service**.

O IDE gera uma classe de serviço Web de exemplo com o nome que você especificou no *Passo 2*. Você pode localizar essa classe na guia **Projects** sob o nó **Web Services** do projeto. Nessa classe, você definirá os métodos que seu serviço Web disponibiliza para aplicativos clientes. Quando por fim você compilar seu aplicativo, o IDE irá gerar outros arquivos de suporte (que discutiremos mais adiante) para seu serviço Web.

31.6.2 Definindo o serviço Web `WelcomeSOAP` no NetBeans

A Figura 31.1 contém o código do serviço Web `WelcomeSOAP`. Você mesmo pode implementar esse código no arquivo `WelcomeSOAP.java` criado na Seção 31.6.1, ou pode simplesmente substituir o código no `WelcomeSOAP.java` por uma cópia do nosso código na pasta desse exemplo. Você pode encontrar esse arquivo na pasta `src\java\com\deitel\welcomesoap` do projeto. Os exemplos do livro podem ser baixados de `www.deitel.com/books/jhtp8/` ou `www.prenhall.com/deitel_br`.

As linhas 5–7 importam as anotações utilizadas nesse exemplo. Por padrão, cada nova classe de serviço Web criada com as APIs JAX-WS é um **POJO (Plain Old Java Object)**, significando que — diferentemente das APIs dos serviços Web prévios — você não precisa estender uma classe ou implementar uma interface para criar um serviço Web.

Ao implantar um aplicativo Web contendo uma classe que utiliza a anotação `@WebService`, o servidor (GlassFish no nosso caso) reconhece que a classe implementa um serviço Web e cria todos os **artefatos no lado do servidor** que suportam o serviço Web — isto é, o framework que permite ao serviço Web esperar solicitações de clientes e responder a essas solicitações depois que ele é implantado em um servidor de aplicativos. Alguns servidores de aplicativo populares de código-fonte aberto que suportam serviços Web Java incluem GlassFish (`glassfish-.dev.java.net`), Apache Tomcat (`tomcat.apache.org`) e o servidor de aplicativos JBoss (`www.jboss.com/products/platforms/application`).

```
1  // Figura 31.1: WelcomeSOAP.java
2  // Serviço Web que retorna uma mensagem de boas-vindas via SOAP.
3  package com.deitel.welcomesoap;
4
5  import javax.jws.WebService; // o programa utiliza a anotação @WebService
6  import javax.jws.WebMethod; // o programa utiliza a anotação @WebMethod
7  import javax.jws.WebParam; // o programa utiliza a anotação @WebParam
8
9  @WebService( // anota a classe como um serviço Web
10    name = "WelcomeSOAP", // configura o nome de interface de extremidade de serviço
11    serviceName = "WelcomeSOAPService" ) // configura o nome de serviço
12 public class WelcomeSOAP
13 {
14    // WebMethod que devolve a mensagem boas-vindas
15    @WebMethod( operationName = "welcome" )
16    public String welcome( @WebParam( name = "name" ) String name )
17    {
18       return "Welcome to JAX-WS web services with SOAP, " + name + "!";
19    } // fim do método welcome
20 } // fim da classe WelcomeSOAP
```

Figura 31.1 | Serviço Web que retorna uma mensagem de boas-vindas via SOAP.

As linhas 9–11 contêm uma anotação `@WebService` (importada na linha 5) com as propriedades `name` e `serviceName`. A **anotação `@WebService`** indica que a classe `WelcomeSOAP` implementa um serviço Web. A anotação é seguida por parênteses contendo atributos de

anotação opcionais. O **atributo `name`** (linha 10) especifica o nome da classe de interface de ponto de extremidade de serviço que será gerada para o cliente. Uma classe de **interface de ponto de extremidade de serviço (Service Endpoint Interface — SEI)** (às vezes chamada **classe proxy**) é utilizada para interagir com o serviço Web — um aplicativo cliente consome o serviço Web invocando métodos no objeto interface de extremidade de serviço. O **atributo `serviceName`** (linha 11) especifica o nome de serviço, que também é o nome da classe que o cliente utiliza para obter um objeto de interface de extremidade de serviço. Se o atributo `serviceName` não for especificado, supõe-se que o serviço Web seja o nome da classe Java seguido pela palavra `Service`. O NetBeans insere a anotação `@WebService` no início de cada nova classe de serviço Web que você cria. Você pode então adicionar as propriedades `name` e `serviceName` entre parênteses depois da anotação.

O serviço `WelcomeSOAP` tem um único método, `welcome` (linhas 15–19), que recebe o nome do usuário como uma `String` e retorna uma `String` que contém uma mensagem de boas-vindas. Esse método é marcado com a **anotação `@WebMethod`** para indicar que ele pode ser chamado remotamente. Quaisquer métodos que não são marcados com `@WebMethod` não são acessíveis aos clientes que consomem o serviço Web. Esses métodos costumam ser métodos utilitários dentro da classe de serviço Web. Observe que a anotação `@WebMethod` utiliza o atributo **`operationName`** para especificar o nome do método que é exibido ao cliente do serviço Web. Se `operationName` não for especificado, ele será configurado como o nome do método Java real.

Erro comum de programação 31.1

Não conseguir exibir um método como um método Web declarando-o com a anotação `@WebMethod` impede que os clientes do serviço Web acessem o método. Há uma exceção — se nenhum dos métodos da classe for declarado com a anotação `@WebMethod`, então todos os métodos `public` da classe serão exibidos como métodos Web.

Erro comum de programação 31.2

Métodos com a anotação `@WebMethod` não podem ser `static`. Deve existir um objeto da classe de serviço Web para que um cliente acesse os métodos Web do serviço.

O parâmetro `name` para `welcome` é anotado com a **anotação `@WebParam`** (linha 16). O nome opcional de atributo **`name`** `@WebParam` indica o nome de parâmetro que é exibido aos clientes do serviço Web. Se você não especificar o nome, o nome do parâmetro real será usado.

31.6.3 Publicando o serviço Web `WelcomeSOAP` a partir do NetBeans

Agora que criamos a classe de serviço Web `WelcomeSOAP`, utilizaremos o NetBeans para construir e publicar (isto é, implantar) o serviço Web para que os clientes possam consumir seus serviços. O NetBeans trata todos os detalhes da criação e implantação de um serviço Web para você. Isso inclui criar o framework necessário para suportar o serviço Web. Clique com o botão direito do mouse no nome do projeto (`WelcomeSOAP`) na guia **Projects** do NetBeans para exibir o menu pop-up na Figura 31.2. Para determinar se há algum erro de compilação no seu projeto, escolha **Build**. Quando o projeto é compilado com sucesso, você pode selecionar **Deploy** para implantá-lo no servidor que você selecionou ao configurar o aplicativo da Web na Seção 31.6.1. Se o código no projeto foi alterado desde a última construção, selecionando **Deploy** também constrói o projeto. Selecionar **Run** executa o aplicativo Web. Se o aplicativo Web não foi anteriormente construído ou implantado, essa opção primeiro executa essas tarefas. Observe que tanto as opções **Deploy** como **Run** também iniciam o servidor de aplicativos (no nosso caso GlassFish) se ele ainda não estiver em execução. Para assegurar que todos os arquivos de código-fonte em um projeto sejam recompilados durante a próxima operação de construção, utilize as opções **Clean** ou **Clean and Build**. Se ainda não fez isso, escolha **Deploy** agora. [*Nota*: Você também pode publicar um serviço Web utilizando um aplicativo Java SE 6 padrão. Para informações adicionais, consulte o artigo `today.java.net/pub/a/today-/2007/07/03/jax-ws-web-services-without-ee-containers.html`.]

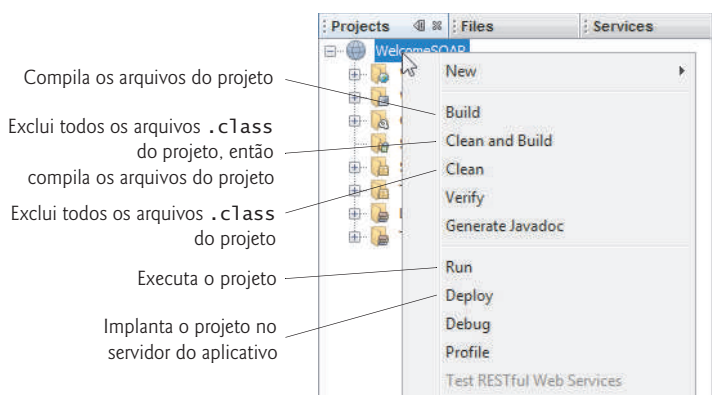


Figura 31.2 | Uma parte do menu pop-up que aparece ao clicar com o botão direito do mouse em um nome de projeto na guia **Projects** do NetBeans.

31.6.4 Testando o serviço Web `WelcomeSOAP` com página Web `Tester` do servidor de aplicativo GlassFish

O próximo passo é testar o serviço Web `WelcomeSOAP`. Anteriormente selecionamos o servidor de aplicativos GlassFish para executar esse aplicativo Web. Esse servidor pode criar dinamicamente uma página Web para testar os métodos de um serviço Web em um navegador Web. Para utilizar esta capacidade:

1. Expanda o **Web Services** do projeto na guia **Projects** do NetBeans.
2. Clique com o botão direito do mouse no nome da classe de serviço Web (`WelcomeSOAP`) e escolha **Test Web Service**.

O servidor de aplicativos GlassFish constrói a página Web `Tester` e carrega-a no navegador Web. A Figura 31.3 mostra a página Web `Tester` do serviço Web `WelcomeSOAP`.

Figura 31.3 | Página Web `Tester` criada pelo GlassFish para o serviço Web `WelcomeSOAP`.

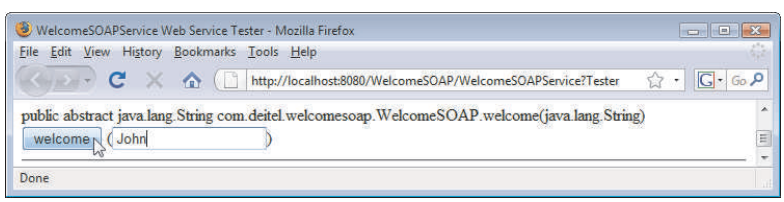
Depois de implantar o serviço Web, você também pode digitar o URL

```
http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?Tester
```

no navegador Web para visualizar a página Web `Tester`. Observe que `WelcomeSOAPService` é o nome (especificado na linha 11 da Figura 31.1) que os clientes utilizam para acessar o serviço Web.

Para testar o método Web `WelcomeSOAP`, digite seu nome no campo de texto à direita do botão **welcome** e então clique no botão para invocar o método e ver o resultado. A Figura 31.4 mostra os resultados de invocar o método `welcome` do `WelcomeSOAP` com o valor `John`.

(a) Invocando o método `welcome` do serviço Web `WelcomeSOAP`



(b) Resultados da chamada do método `welcome` com `"John"`

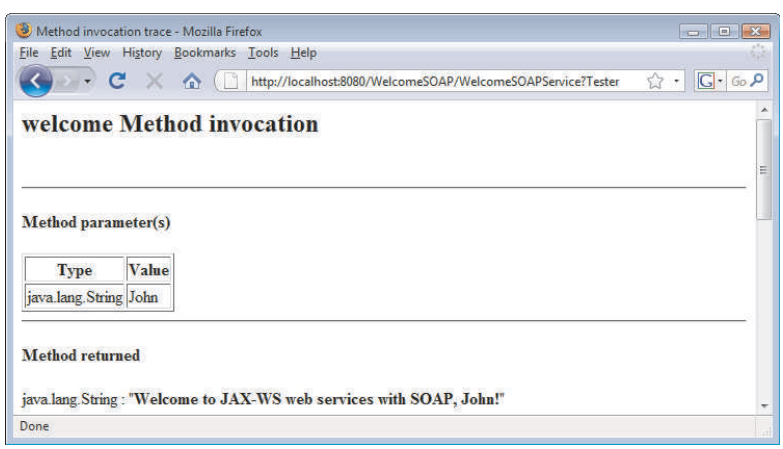


Figura 31.4 | Testando o método `welcome` do `WelcomeSOAP`.

Observe que você só pode acessar o serviço Web quando o servidor de aplicativos estiver em execução. Se NetBeans inicializar o servidor de aplicativos para você, ele o desativará automaticamente quando você fechar o NetBeans. Para manter o servidor de aplicativos pronto e em execução, inicialize-o independentemente do NetBeans antes de implantar ou execute os aplicativos Web no NetBeans. Consulte GlassFish Quick Start em `glassfish.dev.java.net/downloads/quickstart/index.html` para obter informações sobre como iniciar e parar o servidor manualmente.

Testando o Serviço Web `WelcomeSOAP` a partir de outro computador

Se seu computador estiver conectado a uma rede e permitir solicitações HTTP, você poderá testar o serviço Web a partir de outro computador na rede digitando o seguinte URL (onde *host* é o hostname ou o endereço IP do computador no qual o serviço Web é implantado) em um navegador em outro computador:

```
http://host:8080/WelcomeSOAP/WelcomeSOAPService?Tester
```

Nota para usuários do Windows XP e Windows Vista

Por razões de segurança, os computadores em execução no Windows XP ou Windows Vista não permitem solicitações HTTP de outros computadores por padrão. Para permitir que outros computadores se conectem ao seu computador utilizando HTTP, siga estes passos no Windows XP:

1. Selecione Start> Control Panel para abrir janela Control Panel do sistema, clique duas vezes em Windows Firewall para visualizar a caixa de diálogo de configurações Windows Firewall.
2. No diálogo Windows Firewall, clique na guia Exceptions e então clique em Add Port... e adicione a porta 8080 com o nome GlassFish.
3. Clique em OK para fechar a caixa de diálogo de configurações Windows Firewall.

Para permitir que outros computadores se conectem ao seu computador Windows Vista utilizando HTTP, siga estes passos:

1. Abra o Control Panel, alterne para Classic View e dê um clique duplo em Windows Firewall para abrir a caixa de diálogo Windows Firewall.
2. Na caixa de diálogo Windows Firewall, clique no link Change Settings....
3. Na caixa de diálogo Windows Firewall, clique na guia Exceptions e então clique e então clique em Add Port... e adicione a porta 8080 com o nome GlassFish.
4. Clique em OK para fechar a caixa de diálogo de configurações Windows Firewall.

Talvez você também precise desbloquear `java.exe` na primeira vez que executar o servidor GlassFish.

31.6.5 Descrevendo um serviço Web com a Web Service Description Language (WSDL)

Para consumir um serviço Web, um cliente deve determinar sua funcionalidade e como utilizá-lo. Para esse objetivo, os serviços Web normalmente contêm uma **descrição de serviço**. Isso é um documento XML que segue a **Web Service Description Language** (**WSDL**) — um vocabulário XML que define os métodos que um serviço Web disponibiliza e como os clientes interagem com eles. O documento WSDL também especifica informações de baixo nível que talvez os clientes precisem, como formatos necessários para solicitações e respostas.

Documentos WSDL ajudam os aplicativos a determinam como interagir com os serviços Web descritos nos documentos. Não é necessário entender a WSDL para tirar vantagem dela — o servidor de aplicativos GlassFish gera a WSDL de um serviço Web dinamicamente para você, e as ferramentas cliente podem analisar a WSDL para ajudar a criar a classe SEI no lado do cliente que um cliente utiliza para acessar o serviço Web. Como o GlassFish (e a maioria dos outros servidores) gera a WSDL dinamicamente, os clientes sempre recebem a descrição mais atualizada de um serviço Web implantado. Para visualizar a WSDL do serviço Web `WelcomeSOAP`, digite o seguinte URL no navegador:

```
http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

ou clique no link WSDL File na página Web `Tester` (mostrada na Figura 31.3).

Acessando a WSDL do serviço Web `WelcomeSOAP` a partir de outro computador

Por fim, você irá querer que clientes em outros computadores utilizem seu serviço Web. Esses clientes precisam de acesso à WSDL do serviço Web, que eles acessariam com o seguinte URL:

```
http://host:8080/WelcomeSOAP/WelcomeSOAPService?WSDL
```

onde *host* é o hostname ou endereço IP do servidor que hospeda o serviço Web. Como discutido na Seção 31.6.4, isso só funciona se seu computador permitir conexões HTTP de outros computadores — como é o caso para servidores Web publicamente e servidores de aplicativo.

31.6.6 Criando um cliente para consumir o serviço Web `WelcomeSOAP`

Agora que você definiu e implantou o serviço Web, vamos consumi-lo a partir de um aplicativo cliente. Um cliente de serviço Web pode ser qualquer tipo do aplicativo ou até mesmo outro serviço Web. Você permite que um aplicativo cliente baseado em Java consuma um serviço Web **adicionando uma referência de serviço Web** ao aplicativo cliente. Esse processo define a classe SEI que permite ao cliente acessar o serviço Web.

Um aplicativo que consome um serviço Web consiste em um objeto de uma classe de SEI (às vezes chamada classe proxy) que é utilizada para interagir com o serviço Web e um aplicativo cliente que consome o serviço Web invocando métodos no objeto SEI. O código cliente invoca os métodos no objeto SEI, que trata os detalhes da passagem de argumentos de método e recebimento de valores de retorno do serviço Web em nome do cliente. Essa comunicação pode ocorrer por uma rede local, pela Internet ou até mesmo com um serviço Web no mesmo computador. O serviço Web executa a tarefa correspondente e retorna os resultados ao objeto SEI, que então retorna os resultados ao código cliente. A Figura 31.5 representa as interações entre o código cliente, o objeto SEI e o serviço Web. Como você verá a seguir, o NetBeans cria essas classes de SEI para você.

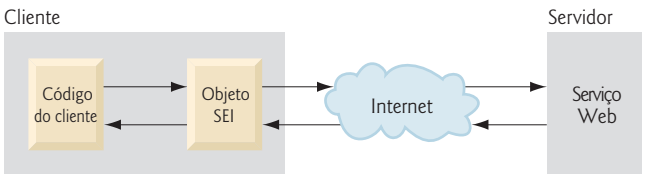


Figura 31.5 | Interação entre um cliente de serviço Web e um serviço Web.

Solicitações e respostas de serviços Web criados com o **JAX-WS** (um dos vários diferentes frameworks de serviço Web) costumam ser transmitidas via SOAP. Qualquer cliente capaz de gerar e processar mensagens SOAP pode interagir com um serviço Web, independentemente da linguagem na qual o serviço Web é escrito.

Agora utilizamos o NetBeans para criar um aplicativo GUI desktop Java cliente. Você então adicionará uma referência de serviço Web ao projeto para que o cliente possa acessar o serviço Web. Ao adicionar a referência, o IDE cria e compila os **artefatos no lado do cliente** — o framework do código Java que suporta a classe SEI no lado do cliente. O cliente chama então métodos em um objeto da classe SEI, que utiliza os demais artefatos para interagir com o serviço Web.

Passo 1: criando um projeto de aplicativo desktop no NetBeans

Antes de seguir os passos nesta seção, certifique-se de que o serviço Web `WelcomeSOAP` foi implantado e que o servidor de aplicativos GlassFish está em execução (ver a Seção 31.6.3). Siga estes passos para criar um aplicativo desktop Java cliente no NetBeans:

1. Escolha **File> New Project...** para abrir a caixa de diálogo **New Project**.
2. Escolha **Java** na lista **Categories** e **Java Application** na lista **Projects**, clique em **Next>**.
3. Especifique o nome `WelcomeSOAPClient` no campo **Project Name** e desmarque a caixa de seleção **Create Main Class**. Mas adiante, você adicionará uma subclasse de `JFrame` que contém um método `main`.
4. Clique em **Finish** para criar o projeto.

Passo 2: adicionando uma referência de serviço Web a um aplicativo

Em seguida, você adicionará uma referência de serviço Web ao seu aplicativo para que ele possa interagir com o serviço Web `WelcomeSOAP`. Para adicionar uma referência de serviço Web, siga os passos abaixo.

1. Clique com o botão direito do mouse no nome do projeto (`WelcomeSOAPClient`) na guia NetBeans **Projects** e selecione **New> Web Service Client...** no menu pop-up para exibir o diálogo **New Web Service Client** (Figura 31.6).

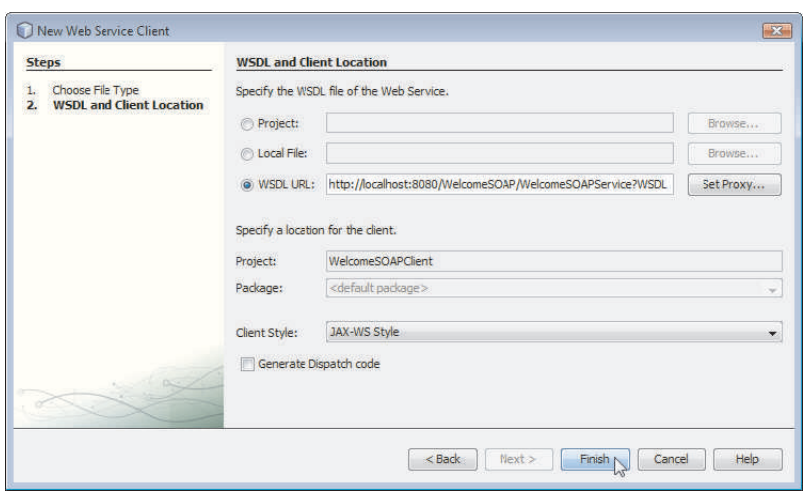


Figura 31.6 | Caixa de diálogo **New Web Service Client**.

2. No campo **WSDL URL**, especifique o URL `http://localhost:8080/WelcomeSOAP/WelcomeSOAPService?WSDL` (Figura 31.6). Esse URL instrui o IDE onde localizar a descrição WSDL do serviço Web. [*Nota*: Se o servidor de aplicativos GlassFish estiver localizado em um computador diferente, substitua `localhost` pelo hostname ou endereço IP desse computador.] O IDE utiliza essa descrição WSDL para gerar os artefatos no lado do cliente que compõem e suportam a SEI. Observe que o diálogo **New Web Service Client** permite procurar serviços Web em várias localizações. Muitas empresas simplesmente distribuem os URLs WSDL exatos para seus serviços Web, que você pode inserir no campo **WSDL URL**.
3. Clique em **Finish** para criar a referência de serviço Web e fechar o diálogo **New Web Service Client**.

Na guia NetBeans **Projects**, o projeto `WelcomeSOAPClient` agora contém uma pasta **Web Service References** com a SEI do serviço Web `WelcomeSOAP` (Figura 31.7). Note que o nome da SEI é listado como `WelcomeSOAPService`, como especificado na linha 11 da Figura 31.1.

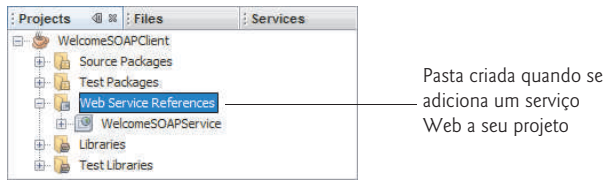


Figura 31.7 | A guia NetBeans **Project** depois de adicionar uma referência de serviço Web ao projeto.

Ao especificar o serviço Web que você quer consumir, o NetBeans acessa e copia as informações sobre a WSDL para um arquivo no seu projeto (nomeado `WelcomeSOAPService.wsdl` nesse exemplo). Você pode visualizar esse arquivo clicando duas vezes no nó `WelcomeSOAPService` na pasta **Web Service References** do projeto. Se o serviço Web mudar, os artefatos no lado do cliente e a cópia do arquivo WSDL do cliente podem ser regenerados clicando com o botão direito do mouse no nó `WelcomeSOAPService` mostrado na Figura 31.7 e selecionando **Refresh Client**.

Você pode examinar os artefatos no lado do cliente gerados pelo IDE selecionando a guia NetBeans **Files** guia e expandindo a pasta **build** do projeto `WelcomeSOAPClient`, como mostrado na Figura 31.8.

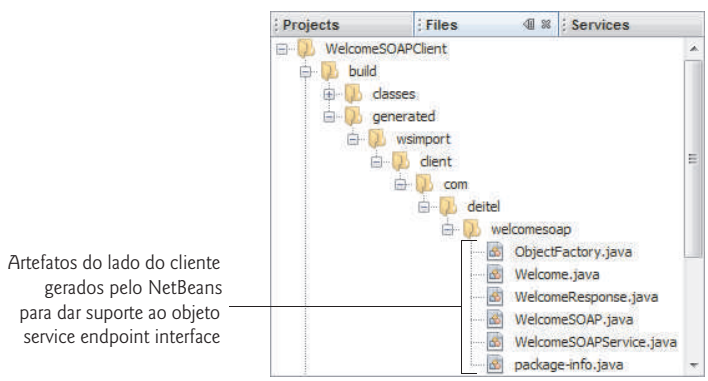


Figura 31.8 | Os artefatos no lado do cliente do serviço Web `WelcomeSOAP` gerados pelo NetBeans.

31.6.7 Consumindo o serviço Web `WelcomeSOAP`

Para esse exemplo, utilizamos um aplicativo GUI para interagir com o serviço Web `WelcomeSOAP`. Para construir a GUI do aplicativo cliente, você primeiro deve adicionar uma subclasse de `JFrame` ao projeto. Para fazer isso, siga estes passos:

1. Clique com o botão direito do mouse no nome do projeto (`WelcomeSOAPClient`) na guia NetBeans **Project** e escolha **New> JFrame Form...** para exibir o diálogo **New JFrame Form**.
2. Especifique `WelcomeSOAPClientJFrame` no campo **Class Name**.
3. Especifique `com.deitel.welcomesoapclient` no campo **Package**.
4. Clique em **Finish** para fechar o diálogo **New JFrame Form**.

Em seguida, utilize as ferramentas de design de interface do NetBeans para construir a interface mostrada nas capturas de tela de exemplo no fim da Figura 31.9. A interface consiste em um rótulo, um campo de texto e um botão.

O aplicativo na Figura 31.9 usa o serviço Web `WelcomeSOAP` para exibir uma mensagem de boas-vindas ao usuário. Para economizar espaço, não mostramos o método `initComponents` autogerado pelo NetBeans, que contém o código que cria os componentes da interface, posiciona-os e registra seus handlers de evento. Para ver o código-fonte completo, abra o arquivo `WelcomeSOAPClientJFrame.java` na pasta desse exemplo sob `src\java\com\deitel\welcomesoapclient`. O NetBeans insere as declarações de variável de instância de componente GUI no fim da classe (linhas 113–115). O Java permite que variáveis de instância sejam declaradas em qualquer lugar no corpo de uma classe desde que sejam inseridas fora dos métodos da classe. Continuamos a declarar nossas variáveis de instância no topo da classe.

```
1   // Figura 31.9: WelcomeSOAPClientJFrame.java
2   // Aplicativo desktop cliente do serviço Web WelcomeSOAP.
3   package com.deitel.welcomesoapclient;
4
5   import com.deitel.welcomesoap.WelcomeSOAP;
6   import com.deitel.welcomesoap.WelcomeSOAPService;
7   import javax.swing.JOptionPane;
8
9   public class WelcomeSOAPClientJFrame extends javax.swing.JFrame
10  {
11     // refere-se ao objeto SEI (isto é, o proxy)
12     private WelcomeSOAP welcomeSOAPProxy;
13
14     // construtor sem argumento
15     public WelcomeSOAPClientJFrame()
16     {
17        initComponents();
18
19        try
20        {
21           // cria os objetos para acessar o serviço Web WelcomeSOAP
22           WelcomeSOAPService service = new WelcomeSOAPService();
23           welcomeSOAPProxy = service.getWelcomeSOAPPort();
24        } // fim do try
25        catch ( Exception exception )
26        {
27           exception.printStackTrace();
28           System.exit( 1 );
29        } // fim do catch
30     } // fim do construtor WelcomeSOAPClientJFrame
31
32     // O método initComponents é autogerado pelo NetBeans e é chamado
33     // a partir do construtor para inicializar a GUI. Esse método não é mostrado
34     // aqui para economizar espaço. Abra WelcomeSOAPClientJFrame.java na
35     // pasta desse exemplo para examinar o código gerado completo.
36
86     // chama o serviço Web com o nome fornecido e exibe a mensagem
87     private void submitJButtonActionPerformed(
88        java.awt.event.ActionEvent evt )
89     {
90        String name = nameJTextField.getText(); // obtém o nome a partir de JTextField
91
92        // recupera a string welcome do serviço Web
93        String message = welcomeSOAPProxy.welcome( name );
94        JOptionPane.showMessageDialog( this, message,
95           "Welcome", JOptionPane.INFORMATION_MESSAGE );
96     } // fim do método submitJButtonActionPerformed
97
98     // método main inicia a execução
99     public static void main( String args[] )
100    {
101       java.awt.EventQueue.invokeLater(
102          new Runnable()
103          {
104             public void run()
105             {
106                new WelcomeSOAPClientJFrame().setVisible( true );
107             } // fim do método run
108          } // fim da classe interna anônima
109       ); // fim da chamada para java.awt. EventQueue.invokeLater
110    } // fim de main
111
112    // Declaração de variáveis - não modifique
113    private javax.swing.JLabel nameJLabel;
114    private javax.swing.JTextField nameJTextField;
115    private javax.swing.JButton submitJButton;
116    // Fim da declaração de variáveis
117 } // fim da classe WelcomeSOAPClientJFrame
```

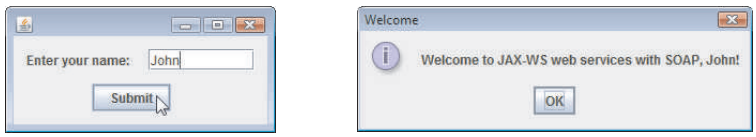


Figura 31.9 | Aplicativo desktop cliente do serviço Web `WelcomeSOAP`.

As linhas 5–6 importam as classes `WelcomeSOAP` e `WelcomeSOAPService` que permitem ao aplicativo cliente interagir com o serviço Web. Note que não temos declarações `import` para a maioria dos componentes GUI utilizados nesse exemplo. Ao criar uma GUI no NetBeans, ele utiliza nomes de classe totalmente qualificados (como `javax.swing.JFrame` na linha 9), portanto as declarações `import` são desnecessárias.

A linha 12 declara uma variável do tipo `WelcomeSOAP` que irá se referir ao objeto SEI. A linha 22 no construtor cria um objeto do tipo `WelcomeSOAPService`. A linha 23 utiliza método `getWelcomeSOAPPort` desse objeto para obter o objeto SEI `WelcomeSOAP` que o aplicativo utiliza para invocar os métodos do serviço Web.

O handler de evento para o botão **Submit** (linhas 87–96) primeiro recupera o nome do usuário inserido a partir de `nameJTextField`. Ele então chama o método `welcome` no objeto SEI (linha 93) para recuperar a mensagem de boas-vindas a partir do serviço Web. Esse objeto se comunica com o serviço Web em nome do cliente. Depois de a mensagem ser recuperada, as linhas 94–95 a exibem em uma caixa de mensagem chamando método `showMessageDialog` de `JOptionPane`.

31.7 Publicando e consumindo serviços Web XML baseados em REST

A seção anterior utilizou um objeto SEI (proxy) para passar dados para e de um serviço Web Java utilizando o protocolo SOAP. Agora, acessaremos um serviço Web Java utilizando a arquitetura REST. Recriamos o exemplo `WelcomeSOAP` para retornar dados no formato XML simples. É possível criar um projeto **Web Application** como você fez na Seção 31.6.

31.7.1 Criando um serviço Web XML baseado em REST

O plug-in **RESTful Web Services** para o NetBeans fornece várias templates para criar serviços Web RESTful, incluindo aquelas que podem interagir com bancos de dados no nome do cliente. Neste capítulo, focalizamos serviços Web RESTful simples. Para criar um serviço Web RESTful:

1. Clique com o botão direito do mouse no nó **WelcomeRESTXML** na guia **Projects** e escolha **New> Other...** para exibir o diálogo **New File**.
2. Selecione **Web Services** sob **Categories** então escolha **RESTful Web Services from Patterns** e clique em **Next>**.
3. Sob **Select Pattern**, certifique-se de que **Singleton** está selecionado e clique em **Next>**.
4. Configure o **Resource Package** como `com.deitel.welcomerestxml`, o **Path** como `welcome` e o **Class Name** como `WelcomeRESTXMLResource`. Deixe o **MIME Type** e **Representation Class** configurados como `application/xml` e `java.lang.String`, respectivamente. A configuração correta é mostrada na Figura 31.10.
5. Clique em **Finish** para criar o serviço Web.

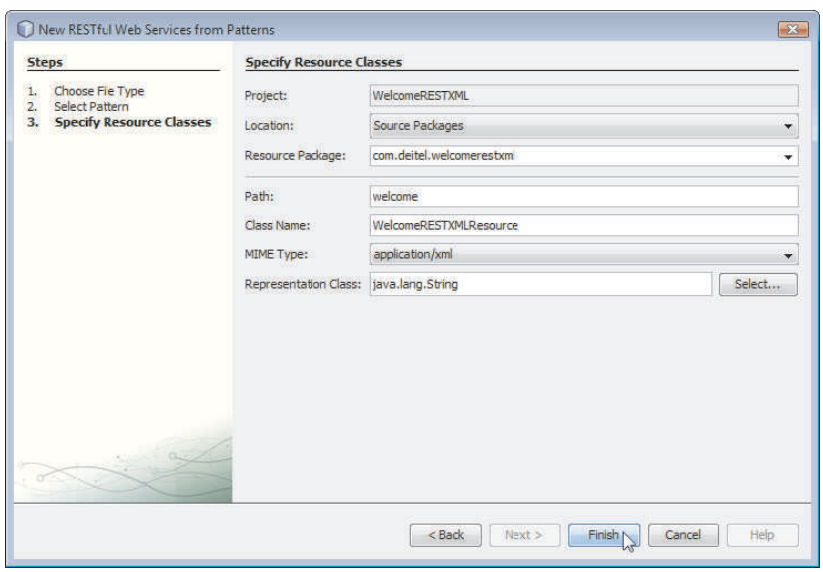


Figura 31.10 | Criando o serviço Web RESTful `WelcomeRESTXML`.

O NetBeans gera a classe e configura as anotações apropriadas. A classe é inserida na pasta Services RESTful Web do projeto. O código do serviço concluído é mostrado na Figura 31.11. Você notará que o código concluído não inclui parte do código gerado pelo NetBeans. Removemos as partes desnecessárias para esse serviço Web simples. O método `putXml` autogerado não é necessário, porque esse exemplo não modifica o estado no servidor. A variável de instância `UriInfo` não é necessária, porque não utilizamos parâmetros de consulta HTTP ou nenhum outro processamento avançado no URI de solicitação. Também removemos o construtor autogerado, porque não temos nenhum código para inserir.

```
1  // Figura 31.11: WelcomeRESTXMLResource.java
2  // O serviço Web REST que retorna uma mensagem de boas-vindas como XML.
3  package com.deitel.welcomerestxml;
4
5  import java.io.StringWriter;
6  import javax.ws.rs.GET; // anotação para indicar o método utiliza HTTP GET
7  import javax.ws.rs.Path; // anotação para especificar o caminho do recurso
8  import javax.ws.rs.PathParam; // anotação para obter parâmetros do URI
9  import javax.ws.rs.Produces; // anotação para especificar o tipo de dados
10 import javax.xml.bind.JAXB; // classe utilitária para operações JAXB comuns
11
12 @Path( "welcome" ) // URI utilizado para acessar o recurso
13 public class WelcomeRESTXMLResource
14 {
15    // recupera a mensagem welcome
16    @GET // trata solicitações HTTP GET
17    @Path( "{name}" ) // componente URI com parâmetro
18    @Produces( "application/xml" ) // resposta formatada como XML
19    public String getXml( @PathParam( "name" ) String name )
20    {
21       String message = "Welcome to JAX-RS web services with REST and " +
22          "XML, " + name + "!"; // nossa mensagem welcome
23       StringWriter writer = new StringWriter();
24       JAXB.marshal( message, writer ); // empacota string como XML
25       return writer.toString(); // retorna XML como String
26    } // fim do método getXml
27 } // fim da classe WelcomeRESTXMLResource
```

Figura 31.11 | Serviço Web REST que retorna uma mensagem de boas-vindas como XML.

As linhas 6–9 contêm as `imports` para as anotações JAX-RS que ajudam a definir o serviço Web RESTful. A **anotação `@Path`** na classe `WelcomeRESTXMLResource` (linha 12) indica o URI para acessar o serviço Web. Esse URI é anexado à Web URL do projeto de aplicativo para invocar o serviço.

Métodos da classe também podem utilizar a anotação `@Path` (linha 17). Partes do caminho especificado entre colchetes indicam parâmetros — eles são um espaço reservado para valores que são passados para o serviço Web como parte do caminho. O caminho de base do serviço é o diretório `resources` do projeto. Por exemplo, para obter uma mensagem de boas-vindas para alguém chamado John, o URL completo é

```
http://localhost:8080/WelcomeRESTXML/resources/welcome/John
```

Os argumentos em um URL podem ser utilizados como argumentos para um método de serviço Web. Para fazer isso, vincule os parâmetros especificados na especificação `@Path` aos parâmetros do método de serviço Web com a **anotação `@PathParam`**, como mostrado na linha 19. Quando a solicitação é recebida, o servidor passa o(s) argumento(s) no URL para o(s) parâmetro(s) apropriado(s) no método do serviço Web.

A **anotação `@GET`** denota que esse método é acessado via uma solicitação HTTP `GET`. Observe que o método `putXml` que o IDE criou para nós tinha uma anotação `@PUT`, que indica que o método é acessado utilizando o método `PUT` HTTP. Há anotações semelhantes para solicitações `POST`, `DELETE` e `HEAD` HTTP.

A **anotação `@Produces`** denota o tipo de conteúdo retornado ao cliente. É possível ter múltiplos métodos com o mesmo método HTTP e o mesmo caminho, mas diferentes anotações `@Produces` e o JAX-RS chamará o método que corresponde ao tipo de conteúdo solicitado pelo cliente. As regras de sobrecarga de método Java padrão são aplicadas; portanto, esses métodos devem ter nomes diferentes. A **anotação `@Consumes`** para o método `putXml` autogerado (que excluímos) restringe o tipo de conteúdo que o serviço Web aceitará de uma operação `PUT`.

A linha 10 importa a **classe `JAXB`** do pacote `javax.xml.bind`. JAXB (**Java Architecture for XML Binding**) é um conjunto de classes para converter POJOs em e da XML. Há muitas classes relacionadas no mesmo pacote que implementam as serializações que realizamos, mas a classe `JAXB` contém empacotadores fáceis de usar para operações comuns. Depois de criar a mensagem de boas-vindas (linhas 21–22),

criamos um `StringWriter` (linha 23) para o qual o JAXB irá gerar a XML. A linha 24 chama o método `JAXB` `static` **`marshal`** da classe para converter a `String` que contém nossa mensagem para o formato XML. A linha 25 chama método `toString` de `StringWriter` para recuperar o texto XML para retornar ao cliente.

Testando serviços Web RESTful

A Seção 31.6.4 demonstrou como testar um serviço SOAP utilizando a página `Tester` do GlassFish. O GlassFish não fornece um recurso de teste para serviços RESTful, mas o NetBeans gera automaticamente uma página de teste que pode ser acessada dando um clique com o botão direito do mouse no nó **WelcomeRESTXML** na guia **Projects** e selecionando **Test RESTful Web Services**. Isso irá compilar e implantar o serviço Web, se você ainda não fez isso, e então abrirá a página de teste. É provável que seu navegador exija que você reconheça uma potencial questão de segurança antes de permitir que a página de teste realize as tarefas. A página de teste é carregada a partir do sistema de arquivos local do seu computador, *não* do servidor GlassFish. Navegadores consideram o sistema de arquivos local e o GlassFish como dois servidores diferentes, embora eles estejam no computador local. Por razões de segurança, navegadores não permitem o chamado script cruzado (*cross-site scripting*) no qual uma página Web tenta interagir com um servidor diferente daquele que disponibilizou a página.

Na página de teste (Figura 31.12), expanda o elemento **welcome** na coluna esquerda e escolha **{name}**. O formulário no lado direito da página exibe um formulário que permite escolher o tipo MIME dos dados (`application/xml` por padrão) e também inserir o valor do parâmetro `name`. Clique no botão **Test** para invocar o serviço Web e exibir a XML retornada.

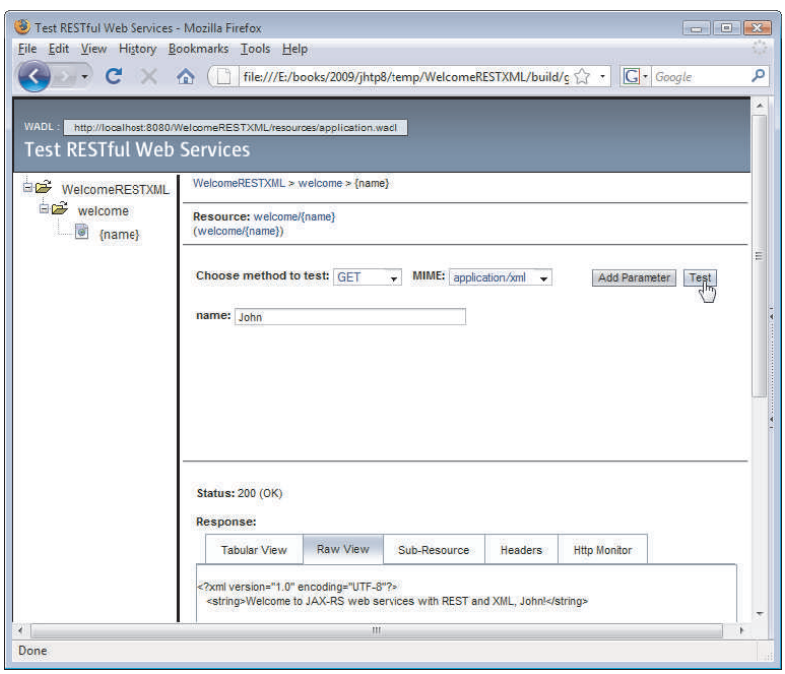


Figura 31.12 | Página de teste para o serviço Web `WelcomeRESTXML`.

A página de teste mostra várias guias que contêm os resultados e várias outras informações. A guia **Raw Text** mostra a resposta XML real. A guia **Headers** mostra os cabeçalhos HTTP retornados pelo servidor. A guia **Http Monitor** mostra um registro em log das transações HTTP que aconteceram para completar a solicitação e resposta.

A página de teste fornece as funcionalidades lendo um arquivo WADL do servidor — você pode ver o URL do arquivo WADL no canto superior esquerdo da página de teste. **WADL (Web Application Description Language)** tem objetivos de design semelhantes à WSDL, mas descreve serviços RESTful em vez de serviços SOAP.

31.7.2 Consumindo um serviço Web XML baseado em REST

Como fizemos com o SOAP, criamos um aplicativo Java que recupera a mensagem de boas-vindas do serviço Web e a exibe para o usuário. Primeiro, crie um aplicativo Java com o nome `WelcomeRESTXMLClient`. Serviços Web RESTful não exibem referências de serviço Web, assim você pode começar a criar a GUI imediatamente criando um formulário `JFrame` chamado `WelcomeRESTXMLClientJFrame` e inserindo-o no pacote `com.deitel.welcomerestxmlclient`. A GUI é idêntica àquela na Figura 31.9, incluindo os nomes dos elementos GUI. Para criar a GUI rapidamente, simplesmente copie e cole a GUI na visualização **Design** da classe `WelcomeSOAPClientJFrame` e cole-a na visualização **Design** da classe `WelcomeRESTXMLClientJFrame`. A Figura 31.13 contém o código completado.

```
1   // Figura 31.13: WelcomeRESTXMLClientJFrame.java
2   // O cliente que consome o serviço WelcomeRESTXML.
3   package com.deitel.welcomerestxmlclient;
4
5   import javax.swing.JOptionPane;
6   import javax.xml.bind.JAXB; // classe utilitária para operações JAXB comuns
7
8   public class WelcomeRESTXMLClientJFrame extends javax.swing.JFrame
9   {
10     // construtor sem argumento
11     public WelcomeRESTXMLClientJFrame()
12     {
13        initComponents();
14     } // fim do construtor
15
16     // O método initComponents é autogerado pelo NetBeans e é chamado
17     // a partir do construtor para inicializar a GUI. Esse método não é mostrado
18     // aqui para economizar espaço. Abra WelcomeRESTXMLClientJFrame.java nesse
19     // pasta desse exemplo para examinar o código gerado completo.
20
72     // chama o serviço Web com o nome fornecido e exibe a mensagem
73     private void submitJButtonActionPerformed(
74        java.awt.event.ActionEvent evt)
75     {
76        String name = nameJTextField.getText(); // obtém o nome a partir de JTextField
77
78        // URL para o serviço REST
79        String url =
80           "http://localhost:8080/WelcomeRESTXML/resources/welcome/" + name;
81
82        // lê do URL e converte da XML em string Java
83        String message = JAXB.unmarshal( url, String.class );
84
85        // exibe a mensagem ao usuário
86        JOptionPane.showMessageDialog( this, message,
87              "Welcome", JOptionPane.INFORMATION_MESSAGE );
88     } // fim do método submitJButtonActionPerformed
89
90     // método main inicia a execução
91     public static void main( String args[] )
92     {
93        java.awt.EventQueue.invokeLater(
94           new Runnable()
95           {
96              public void run()
97              {
98                 new WelcomeRESTXMLClientJFrame().setVisible( true );
99              } // fim do método run
100          } // fim da classe interna anônima
101       ); // fim da chamada a java.awt. EventQueue.invokeLater
102    } // fim de main
103
104    // Declaração de variáveis - não modifique
105    private javax.swing.JLabel nameJLabel;
106    private javax.swing.JTextField nameJTextField;
107    private javax.swing.JButton submitJButton;
108    // Fim da declaração de variáveis
109 } // fim da classe WelcomeRESTXMLClientJFrame
```

Figura 31.13 | O cliente que consome o serviço `WelcomeRESTXML`.

Você pode acessar um serviço Web RESTful com classes da API do Java. Como no serviço Web XML RESTful, utilizamos a biblioteca JAXB. A classe JAXB (importada na linha 6) tem um método `static` **`unmarshal`** que recebe como argumentos um nome de arquivo ou URL como uma `String`, e um objeto `Class<T>` indicando a classe Java na qual a XML será convertida (linha 83). Nesse exemplo, a XML contém um objeto `String`, portanto utilizamos o atalho de compilador Java `String.class` para criar o objeto `Class<String>` que precisamos como o segundo argumento. A `String` retornada da chamada ao método `unmarshal` é então exibida para o usuário via método `showMessageDialog` de `JOPTIONPANE` (linhas 86–87), como ocorreu com o serviço SOAP. Observe que o URL utilizado nesse exemplo para extrair os dados do serviço Web corresponde ao URL utilizado pela página de teste.

31.8 Publicando e consumindo serviços Web JSON baseados em REST

Embora a XML tenha sido projetada principalmente como um formato de troca de documentos, JSON é projetado como um formato de troca *de dados*. As estruturas de dados na maioria das linguagens de programação não mapeiam diretamente para construções XML — por exemplo, a distinção entre elementos e atributos não estão presentes nas estruturas de dados de linguagem de programação. O JSON é um subconjunto da linguagem de programação JavaScript, e os seus componentes — objetos, arrays, strings, números — podem ser facilmente mapeados para construções no Java e em outras linguagens de programação.

Atualmente as bibliotecas Java padrão não fornecem capacidades para trabalhar com o JSON, mas há várias bibliotecas JSON de código-fonte aberto para Java e outras linguagens; você pode localizar uma lista delas em `json.org`. Escolhemos a biblioteca Gson em `code.google.com/p/google-gson/`, que fornece um modo simples de converter POJOs em JSON e vice-versa.

31.8.1 Criando um serviço Web JSON baseado em REST

Para começar, crie um aplicativo Web WelcomeRESTJSON e então crie o serviço Web seguindo os passos na Seção 31.7.1. No *Passo 4*, altere o **Resource Package** para `com.deitel.welcomerestjson`, o **Class Name** para `WelcomeRESTJSONResource` e o **MIME Type** para `application/json`. Além disso, você precisa baixar o arquivo JAR da biblioteca Gson e então adicioná-lo ao projeto como uma biblioteca. Para fazer isso, clique com o botão direito do mouse na pasta **Libraries** do seu projeto, escolha **Add JAR/Folder...** para localizar o arquivo JAR da biblioteca Gson baixado e clique em **Open**. O código completo do serviço pode ser encontrado na Figura 31.14.

```
1  // Figura 31.14: WelcomeRESTJSONResource.java
2  // O serviço Web REST que retorna uma mensagem de boas-vindas como JSON.
3  package com.deitel.welcomerestjson;
4
5  import com.google.gson.Gson; // converte POJO em JSON e de volta novamente
6  import javax.ws.rs.GET; // anotação para indicar o método utiliza HTTP GET
7  import javax.ws.rs.Path; // anotação para especificar o caminho do recurso
8  import javax.ws.rs.PathParam; // anotação para obter parâmetros do URI
9  import javax.ws.rs.Produces; // anotação para especificar o tipo de dados
10
11 @Path( "welcome" ) // caminho utilizado para acessar o recurso
12 public class WelcomeRESTJSONResource
13 {
14    // recupera a mensagem welcome
15    @GET // trata solicitações HTTP GET
16    @Path( "{name}" ) // recebe o nome como um parâmetro de caminho
17    @Produces( "application/json" ) // resposta formatada como JSON
18    public String getJson( @PathParam( "name" ) String name )
19    {
20       // adiciona a mensagem de boas-vidnas ao campo do objeto TextMessage
21       TextMessage message = new TextMessage(); // cria o objeto empacotador
22       message.setMessage( String.format( "%s, %s!",
23          "Welcome to JAX-RS web services with REST and JSON", name ) );
24
25       return new Gson().toJson( message ); // retorna a mensagem empacotada pelo JSON
26    } // fim do método getJson
27 } // fim da classe WelcomeRESTJSONResource
28
29 // classe privada que contém a mensagem que desejamos enviar
30 class TextMessage
31 {
32    private String message; // mensagem que estamos enviando
33
34    // retorna a mensagem
35    public String getMessage()
36    {
37       return message;
38    } // fim do método getMessage
39
40    // configura a mensagem
41    public void setMessage( String value )
42    {
43       message = value;
44    } // fim do método setMessage
45 } // fim da classe TextMessage
```

Figura 31.14 | Serviço Web REST que retorna uma mensagem de boas-vindas como JSON.

Todas as anotações e a estrutura básica da classe `WelcomeRESTJSONResource` são as mesmas que aquelas no exemplo do REST XML. Observe que o argumento para o atributo `@Produces` (linha 19) é `"application/json"`. A classe `TextMessage` (linhas 30–45) resolve uma diferença entre JSON e XML. O JSON não permite que strings ou números permaneçam isolados — eles devem ser encapsulados em um tipo de dados composto. Desse modo, criamos a classe `TextMessage` para encapsular a `String` que representa a mensagem.

Quando um cliente invoca esse serviço Web, a linha 21 cria o objeto `TextMessage`, então as linhas 22–23 configuram a mensagem contida. Em seguida, a linha 25 cria um objeto **`Gson`** (do pacote `com.google.gson.Gson`) e chama seu método **`toJson`** para converter a `TextMessage` na sua representação `String` JSON. Retornamos essa `String`, que é então reenviada ao cliente na resposta do serviço Web. Há múltiplas sobrecargas do método `toJson`, como aquela que envia a saída a um `Writer` em vez de retornar uma `String`.

Serviços RESTful que retornam JSON podem ser testados do mesmo modo que aqueles que retornam XML. Siga o procedimento delineado na Seção 31.7.1, mas certifique-se de alterar o tipo MIME para `application/json` na página Web de teste; do contrário, o serviço Web retornará um erro declarando que ele não pode produzir a resposta desejada.

31.8.2 Consumindo um serviço Web JSON baseado em REST

Agora criamos um aplicativo Java que recupera a mensagem de boas-vindas do serviço Web e a exibe ao usuário. Primeiro, crie um aplicativo Java com o nome `WelcomeRESTJSONClient`. Crie então um formulário `JFrame` chamado `WelcomeRESTXMLClientJFrame` e insira-o no pacote `com.deitel.welcomerestjsonclient`. A GUI é idêntica àquela na Figura 31.9. Para criar a GUI rapidamente, copie-a da visualização **Design** da classe `WelcomeSOAPClientJFrame` e cole-a na visualização **Design** da classe `WelcomeRESTJSONClient-JFrame`. A Figura 31.13 contém o código completado.

```
1  // Figura 31.15: WelcomeRESTJSONClientJFrame.java
2  // Cliente que consome o serviço WelcomeRESTJSON.
3  package com.deitel.welcomerestjsonclient;
4
5  import com.google.gson.Gson; // converte POJO em JSON e de volta novamente
6  import java.io.InputStreamReader;
7  import java.net.URL;
8  import javax.swing.JOptionPane;
9
10 public class WelcomeRESTJSONClientJFrame extends javax.swing.JFrame
11 {
12    // construtor sem argumento
13    public WelcomeRESTJSONClientJFrame()
14    {
15       initComponents();
16    } // fim do construtor
17
18    // O método initComponents é autogerado pelo NetBeans e é chamado
19    // a partir do construtor para inicializar a GUI. Esse método não é mostrado
20    // aqui para economizar espaço. Abre WelcomeRESTJSONClientJFrame.java nesse
21    // pasta desse exemplo para examinar o código gerado completo.
22
74    // chama o serviço Web com o nome fornecido e exibe a mensagem
75    private void submitJButtonActionPerformed(
76       java.awt.event.ActionEvent evt )
77    {
78       String name = nameJTextField.getText(); // obtém o nome a partir de JTextField
79
80       // recupera a string welcome do serviço Web
81       try
82       {
83          // URL do serviço Web
84          String url = "http://localhost:8080/WelcomeRESTJSON/" +
85             "resources/welcome/" + name;
86
87          // abre o URL, utilizando um Reader para converter bytes em chars
88          InputStreamReader reader =
89             new InputStreamReader( new URL( url ).openStream() );
90
91          // analisa o JSON de volta em uma TextMessage
92          TextMessage message =
93             new Gson().fromJson( reader, TextMessage.class );
```

```
94
95             // exibe a mensagem ao usuário
96             JOptionPane.showMessageDialog( this, message.getMessage(),
97                "Welcome", JOptionPane.INFORMATION_MESSAGE );
98          } // fim do try
99          catch( Exception exception )
100         {
101            exception.printStackTrace(); // mostra os detalhes de exceção
102         } // fim do catch
103      } // fim do método submitJButtonActionPerformed
104
105      // método main inicia a execução
106      public static void main( String args[] )
107      {
108         java.awt.EventQueue.invokeLater(
109            new Runnable()
110            {
111               public void run()
112               {
113                  new WelcomeRESTJSONClientJFrame().setVisible( true );
114               } // fim do método run
115            } // fim da classe interna anônima
116         ); // fim da chamada a java.awt. EventQueue.invokeLater
117      } // fim de main
118
119      // Declaração de variáveis - não modifique
120      private javax.swing.JLabel nameJLabel;
121      private javax.swing.JTextField nameJTextField;
122      private javax.swing.JButton submitJButton;
123      // Fim da declaração de variáveis
124   } // fim da classe WelcomeRESTJSONClientJFrame
125
126   // a classe privada que contém a mensagem que estamos recebendo
127   class TextMessage
128   {
129      private String message; // mensagem que estamos recebendo
130
131      // retorna a mensagem
132      public String getMessage()
133      {
134         return message;
135      } // fim do método getMessage
136
137      // configura a mensagem
138      public void setMessage( String value )
139      {
140         message = value;
141      } // fim do método setMessage
142   } // fim da classe TextMessage
```

Figura 31.15 | O cliente que consome o serviço `WelcomeRESTJSON`.

As linhas 84–85 criam `String` do URL que é utilizado para invocar o serviço Web. As linhas 88–89 criam um objeto `URL` utilizando essa `String`, depois chamam o método **`openStream`** `URL` para invocar o serviço Web e obter um `InputStream` a partir do qual o cliente pode ler a resposta. O `InputStream` é empacotado em um `InputStreamReader` para que possa ser passado como o primeiro argumento para o método **`fromJson`** da classe `Gson`. Esse método é sobrecarregado. A versão que utilizamos recebe como argumentos um `Reader`, que contém uma `String` JSON, e um objeto `Class<T>`, que indica a classe Java em que a `String` JSON será convertida (linha 93). Nesse exemplo, a `String` JSON contém um objeto `TextMessage`, assim utilizamos o atalho de compilador Java `TextMessage.class` para criar o objeto `Class<TextMessage>` que precisemos como o segundo argumento. As linhas 96–97 exibem a mensagem no objeto `TextMessage`.

Observe que as classes `TextMessage` no cliente e serviço Web não estão relacionadas. Tecnicamente, o cliente pode ser escrito em qualquer linguagem de programação, portanto a maneira como uma resposta é processada pode variar significativamente. Como o nosso cliente é escrito em Java, duplicamos a classe `TextMessage` no cliente para poder converter facilmente o objeto JSON de volta ao Java.

31.9 Rastreamento de sessão em um serviço Web baseado em SOAP

A Seção 29.7 descreveu as vantagens do uso do rastreamento de sessão para manter informações de estado de cliente a fim de personalizar as experiências de navegação dos usuários. Agora incorporaremos o rastreamento de sessão a um serviço Web. Suponha que um aplicativo cliente precise chamar vários métodos a partir do mesmo serviço Web, possivelmente várias vezes cada um. Nesse caso, pode ser vantajoso que o serviço Web mantenha informações de estado do cliente, eliminando assim a necessidade de passar informações do cliente entre o cliente e o serviço Web diversas vezes. Por exemplo, um serviço Web que fornece avaliações de restaurantes locais poderia armazenar o endereço residencial do usuário cliente durante a solicitação inicial e então utilizá-lo para retornar resultados personalizados e localizados nas solicitações subsequentes. O armazenamento de informações de sessão também permite que um serviço Web diferencie seus clientes.

31.9.1 Criando um serviço Web `Blackjack`

Nosso próximo exemplo é um serviço Web que ajuda a desenvolver o jogo de cartas vinte-e-um. O serviço Web `Blackjack` (Figura 31.16) fornece métodos Web para embaralhar as cartas, distribuir as cartas e avaliar uma mão de cartas. Depois de apresentar o serviço Web, iremos utilizá-lo para servir como um carteador para uma partida de vinte-e-um (Figura 31.17). O serviço Web `Blackjack` utiliza um objeto `HttpSession` para manter um baralho único de cartas para cada aplicativo cliente. Vários clientes podem utilizar o serviço ao mesmo tempo, mas as chamadas de método Web feitas por um cliente específico utilizam apenas as cartas armazenadas na sessão desse cliente. Nosso exemplo utiliza as seguintes regras do jogo vinte-e-um:

> *Duas cartas são distribuídas ao carteador e ao jogador. As cartas do jogador são distribuídas viradas para cima. Somente a primeira carta do carteador é distribuída virada para cima. Cada carta tem um valor. Uma carta numerada de 2 a 10 vale seu valor de face. Valetes, rainhas e reis cada um valem 10. Ases podem valer 1 ou 11 — aquele que for mais vantajoso para o jogador (como veremos a seguir). Se a soma de duas cartas iniciais do jogador for 21 (isto é, o jogador recebeu uma carta com valor de 10 e um ás, que conta como 11 nessa situação), o jogador conseguiu um "vinte-e-um" e ganha imediatamente o jogo — se o carteador também não tiver um vinte-e-um (o que resultaria em um "push" — isto é, um empate). Do contrário, o jogador pode começar a pedir cartas adicionais uma após a outra. Essas cartas são distribuídas viradas para cima, e o jogador decide quando parar de pedir cartas. Se o jogador "estourar" (isto é, a soma das cartas do jogador exceder 21), o jogo termina, e o jogador perde. Quando o jogador está satisfeito com o conjunto atual de cartas, ele "fica como está" (stand, isto é, para de pedir cartas), e a carta oculta do carteador é exibida. Se o total do carteador for 16 ou menos, ele deve bater na mesa (hit), isto é, pedir outra carta; caso contrário, o carteador deve ficar como está. O carteador deve continuar a pedir cartas até que a soma das cartas dele seja maior ou igual a 17. Se o carteador exceder 21, o jogador ganha. Do contrário, a mão com número total mais alto de pontos ganha. Se o carteador e o jogador tiverem o mesmo número total de pontos, o jogo empata, e ninguém ganha. Observe que o valor de um ás para um carteador depende da(s) outra(s) carta(s) do carteador e das regras do cassino. Em geral, um carteador deve continuar batendo na mesa enquanto tiver um total de 16 ou menos e deve ficar como está se tiver 17 ou mais pontos. Mas para um "soft 17" — mão com um total de 17 com um ás contado como 11 — alguns cassinos exigem que o carteador peça mais cartas e outros exigem que o carteador fique como está (em nosso exemplo, exigimos que o carteador fique como está). Essa mão é conhecida como "soft 17" porque pedir outra carta não pode estourar a mão.*

O serviço Web (Figura 31.16) armazena cada carta como uma `String` consistindo em um número, `1`–`13`, representando a face da carta (de ás a rei, respectivamente), seguido por um espaço e um dígito, `0`–`3`, representando o naipe da carta (copas, ouros, paus ou espadas, respectivamente). Por exemplo, o valete de paus é representado como `"11 2"` e o dois de copas como `"2 0"`. Para criar e implantar esse serviço Web, siga os passos que apresentamos nas seções 31.6.2–31.6.3 para o serviço `WelcomeSOAP`.

```
1  // Figura 31.16: Blackjack.java
2  // Serviço Web Blackjack que distribui cartas e avalia mãos
3  package com.deitel.java.blackjack;
4
5  import java.util.ArrayList;
6  import java.util.Random;
7  import javax.annotation.Resource;
8  import javax.jws.WebMethod;
9  import javax.jws.WebParam;
10 import javax.jws.WebService;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpSession;
13 import javax.xml.ws.WebServiceContext;
14 import javax.xml.ws.handler.MessageContext;
15
16 @WebService( name = "Blackjack", serviceName = "BlackjackService" )
17 public class Blackjack
18 {
```

```
19       // use @Resource para criar um WebServiceContext para o rastreamento de sessão
20       private @Resource WebServiceContext webServiceContext;
21       private MessageContext messageContext; // utilizado no rastreamento de sessão
22       private HttpSession session; // armazena atributos da sessão
23
24       // distribui um card
25       @WebMethod( operationName = "dealCard" )
26       public String dealCard()
27       {
28          String card = "";
29
30          ArrayList< String > deck =
31             ( ArrayList< String > ) session.getAttribute( "deck" );
32
33          card = deck.get( 0 ); // obtém a carta no topo do baralho
34          deck.remove( 0 ); // remove a carta no topo do baralho
35
36          return card;
37       } // fim do método WebMethod dealCard
38
39       // embaralha as cartas
40       @WebMethod( operationName = "shuffle" )
41       public void shuffle()
42       {
43          // obtém o objeto HttpSession para armazenar o baralho para o cliente atual
44          messageContext = webServiceContext.getMessageContext();
45          session = ( ( HttpServletRequest ) messageContext.get(
46             MessageContext.SERVLET_REQUEST ) ).getSession();
47
48          // preenche o baralho de cartas
49          ArrayList< String > deck = new ArrayList< String >();
50
51          for ( int face = 1; face <= 13; face++ ) // faz um loop pelas faces das cartas
52             for ( int suit = 0; suit <= 3; suit++ ) // faz um loop pelos naipes
53                deck.add( face + " " + suit ); // adiciona carta ao deck
54
55          String tempCard; // mantém a carta temporariamente durante a troca
56          Random randomObject = new Random(); // gera números aleatórios
57          int index; // índice da carta selecionada aleatoriamente
58
59          for ( int i = 0; i < deck.size() ; i++ ) // shuffle
60          {
61             index = randomObject.nextInt( deck.size() - 1 );
62
63             // troca a carta na posição i por uma carta selecionada aleatoriamente
64             tempCard = deck.get( i );
65             deck.set( i, deck.get( index ) );
66             deck.set( index, tempCard );
67          } // for final
68
69          // adicione esse baralho à sessão do usuário
70          session.setAttribute( "deck", deck );
71       } // fim do shuffle WebMethod
72
73       // determine o valor de uma mão
74       @WebMethod( operationName = "getHandValue" )
75       public int getHandValue( @WebParam( name = "hand" ) String hand )
76       {
77          // divide a mão em cartas
78          String[] cards = hand.split( "\t" );
79          int total = 0; // valor total das cartas na mão
80          int face; // face da carta atual
81          int aceCount = 0; // número de ases na mão
82
83          for ( int i = 0; i < cards.length; i++ )
84          {
85             // analisa a string e obtêm o primeiro int na String
86             face = Integer.parseInt(
87                cards[ i ].substring( 0, cards[ i ].indexOf( " " ) ) );
```

```
88
89             switch ( face )
90             {
91                case 1: // se ás, incrementa aceCount
92                   ++aceCount;
93                   break;
94                case 11: // valete
95                case 12: // rainha
96                case 13: // rei
97                   total += 10;
98                   break;
99                default: // do contrário, adiciona a face da carta
100                  total += face;
101                  break;
102            } // fim do switch
103         } // for final
104
105         // calcula o uso ótimo dos ases
106         if ( aceCount > 0 )
107         {
108            // se possível, conta um ás como 11
109            if ( total + 11 + aceCount - 1 <= 21 )
110               total += 11 + aceCount - 1;
111            else // do contrário, conta todos os ases como 1
112               total += aceCount;
113         } // fim do if
114
115         return total;
116      } // fim de WebMethod getHandValue
117   } // fim da classe Blackjack
```

Figura 31.16 | Serviço Web `Blackjack` que distribui cartas e avalia mãos.

Rastreamento de sessão nos serviços Web

Um cliente primeiro chama o método `shuffle` de um serviço Web `Blackjack` (linhas 40–71) para embaralhar as cartas e inseri-las em um objeto `HttpSession` que é específico a esse cliente. Para utilizar o rastreamento de sessão em um serviço Web, você deve incluir o código para os recursos que mantêm as informações de estado de sessão. No passado, você tinha de escrever código tedioso para criar esses recursos. Isso é tratado para você via **anotação `@Resource`**. Essa anotação permite a ferramentas como o NetBeans "injetar" objetos de suporte na sua classe, permitindo assim que você se concentre na lógica do negócio em vez de no código de suporte. O servidor é responsável por inicializar os objetos de suporte. O conceito do uso de anotações para adicionar código que suporta suas classes é conhecido como **injeção de dependência**. Anotações como `@WebService`, `@WebMethod` e `@WebParam` também executam a injeção de dependência.

A linha 20 injeta um objeto **`WebServiceContext`** na sua classe. Esse objeto permite a um serviço Web acessar e manter informações, como estado de sessão de uma solicitação específica. Ao examinar o código na Figura 31.16, você notará que nós nunca criamos explicitamente o objeto `WebServiceContext` — esse código é injetado na classe pelo servidor de aplicativos como resultado da anotação `@Resource`. A linha 21 declara uma variável de tipo de interface **`MessageContext`** que o serviço Web utilizará para obter um objeto `HttpSession` para o cliente atual. A linha 22 declara a variável `HttpSession` que o serviço Web usará para manipular as informações de estado de sessão.

A linha 44 no método `shuffle` utiliza o objeto `WebServiceContext` que foi injetado na linha 20 para obter um objeto `MessageContext`. As linhas 45–46 utilizam então o método `get` do objeto `MessageContext` para obter o objeto `HttpSession` para o cliente atual. O método `get` recebe uma constante indicando o que obter do `MessageContext`. Nesse caso, a constante `MessageContext.SERVLET_REQUEST` indica que queremos obter o objeto `HttpServletRequest` para o cliente atual. Chamamos então o método `HttpServletRequest` **`getSession`** para obter o objeto `HttpSession`.

As linhas 49–70 geram um `ArrayList` representando um baralho de cartas, embaralham as cartas e as armazenam no cliente objeto `session`. As linhas 51–53 geram `String`s na forma "*face naipe* " para representar cada possível carta no baralho. As linhas 59–67 embaralham as cartas trocando cada carta por outra selecionada aleatoriamente. A linha 70 utiliza método `HttpSession` **`setAttribute`** para inserir o `ArrayList` no objeto `session` a fim de manter o baralho entre chamadas de método de um determinado cliente.

As linhas 25–37 definem o método `dealCard` como um método Web. As linhas 30–31 utilizam o objeto `session` para obter o atributo de sessão `"deck"` que foi armazenado na linha 70 do método `shuffle`. O método **`getAttribute`** recebe como um parâmetro uma `String` que identifica o `Object` a obter do estado de sessão. O `HttpSession` pode armazenar muitos `Objeto`s, desde que cada um tenha um identificador único. Observe que o método `shuffle` deve ser chamado antes que o método `dealCard` seja inicialmente chamado para um cliente — do contrário, uma exceção ocorre na linha 33, porque `getAttribute` retorna `null` nas linhas 30–31. Depois de obter o baralho do usuário, `dealCard` obtém a carta no topo do baralho (linha 33), remove-a do baralho (linha 34) e retorna o valor da carta como uma

`String` (linha 36). Sem utilizar o rastreamento de sessão, o baralho de cartas precisaria ser passado entre cada chamada de método. O rastreamento de sessão torna mais fácil chamar o método `dealCard` (ele não requer nenhum argumento) e elimina o overhead de enviar o baralho pela rede várias vezes.

O método `getHandValue` (linhas 74–116) determina o valor total das cartas em uma mão tentando alcançar a pontuação mais alta possível sem exceder 21. Lembre-se de que um ás pode ser contado como ou 1 ou 11, e todas as figuras contam como 10. Esse método não utiliza o objeto `session`, porque o baralho de cartas não é utilizado nesse método.

Como veremos mais adiante, o aplicativo cliente mantém uma mão das cartas como uma `String` em que cada carta é separada por um caractere de tabulação. A linha 78 tokeniza a mão de cartas (representada por `hand`) em cartas individuais chamando o método `split` `String` e passando para ele uma `String` contendo os caracteres delimitadores (neste caso, apenas uma tabulação). O método `split` utiliza os caracteres delimitadores para separar tokens em `String`s. As linhas 83–103 contam o valor de cada carta. As linhas 86–87 recuperam o primeiro número inteiro — a face — e usam esse valor na instrução `switch` (linhas 89–102). Se o cartão for um ás, o método incrementará a variável `aceCount`. Discutimos como essa variável é utilizada a seguir. Se a carta for um 11, 12 ou 13 (valete, rainha ou rei), o método adiciona 10 ao valor total da mão (linha 97). Se a carta for qualquer outra coisa, o método aumentará o total por esse valor (linha 100).

Como um ás pode ter dois valores, a lógica adicional é necessária para processar ases. As linhas 106–113 do método `getHandValue` processam os ases depois de todas as outras cartas. Se uma mão contiver vários ases, somente um ás poderá ser contado como 11. A condição na linha 109 determina se a contagem de um ás como 11 e o resto como 1 resultará em um total que não excede 21. Se isso for possível, a linha 110 ajusta o total correspondentemente. Caso contrário, a linha 112 ajusta o total, contando cada ás como 1.

O método `getHandValue` maximiza o valor das cartas atuais sem exceder 21. Imagine, por exemplo, que o carteador tem um 7 e recebe um ás. O novo total pode ser ou 8 ou 18. Mas `getHandValue` sempre maximiza o valor das cartas sem ultrapassar 21, assim o novo total é 18.

31.9.2 Consumindo o serviço Web `Blackjack`

O aplicativo de vinte-e-um na Figura 31.17 monitora as cartas do jogador e as do carteador, e o serviço Web monitora as cartas que foram distribuídas. O construtor (linhas 34–83) configura a GUI (linha 36), altera a cor de fundo da janela (linha 40) e cria o objeto SEI do serviço Web `Blackjack` (linhas 46–47). Na GUI, cada jogador tem 11 `JLabel`s — o número máximo de cartas que pode ser distribuído sem automaticamente exceder 21 (isto é, quatro ases, quatro dois e três três). Esses `JLabel`s são inseridos em um ArrayList de `Label`s (linhas 59–82), assim podemos indexar o `ArrayList` durante o jogo para determinar o `JLabel` que exibirá a imagem de uma carta em particular.

```
1  // Figura 31.17: BlackjackGameJFrame.java
2  // Jogo de vinte-e-um que usa o serviço web Blackjack.
3  package com.deitel.java.blackjackclient;
4
5  import com.deitel.java.blackjack.Blackjack;
6  import com.deitel.java.blackjack.BlackjackService;
7  import java.awt.Color;
8  import java.util.ArrayList;
9  import javax.swing.ImageIcon;
10 import javax.swing.JLabel;
11 import javax.swing.JOptionPane;
12 import javax.xml.ws.BindingProvider;
13
14 public class BlackjackGameJFrame extends javax.swing.JFrame
15 {
16    private String playerCards;
17    private String dealerCards;
18    private ArrayList<JLabel> cardboxes; // lista de JLabels da imagem da carta
19    private int currentPlayerCard; // o número da carta atual do jogador
20    private int currentDealerCard; // o número da carta atual do blackjackProxy
21    private BlackjackService blackjackService; // utilizado para obter o proxy
22    private Blackjack blackjackProxy; // utilizado para acessar o serviço Web
23
24    // enumeração dos estados do jogo
25    private enum GameStatus
26    {
27       PUSH, // o jogo termina em um empate
28       LOSE, // o jogador perde
29       WIN, // o jogador ganha
30       BLACKJACK // o jogador fez vinte-e-um
31    } // fim do enum GameStatus
32
33    // construtor sem argumento
34    public BlackjackGameJFrame()
35    {
```

```
36          initComponents();
37
38          // por causa de um bug no NetBeans, precisamos alterar o fundo do JFRAME
39          // usa cores aqui em vez de no designer
40          getContentPane().setBackground( new Color( 0, 180, 0 ) );
41
42          // inicializa o proxy blackjack
43          try
44          {
45             // cria os objetos para acessar o serviço Web Blackjack
46             blackjackService = new BlackjackService();
47             blackjackProxy = blackjackService.getBlackjackPort();
48
49             // permite rastreamento de sessão
50             ( (BindingProvider) blackjackProxy ).getRequestContext().put(
51                BindingProvider.SESSION_MAINTAIN_PROPERTY, true );
52          } // fim do try
53          catch ( Exception e )
54          {
55             e.printStackTrace();
56          } // fim do catch
57
58          // adiciona JLabels a ArrayList cardBoxes para manipulação programática
59          cardboxes = new ArrayList<JLabel>();
60
61          cardboxes.add( dealerCard1JLabel );
62          cardboxes.add( dealerCard2JLabel );
63          cardboxes.add( dealerCard3JLabel );
64          cardboxes.add( dealerCard4JLabel );
65          cardboxes.add( dealerCard5JLabel );
66          cardboxes.add( dealerCard6JLabel );
67          cardboxes.add( dealerCard7JLabel );
68          cardboxes.add( dealerCard8JLabel );
69          cardboxes.add( dealerCard9JLabel );
70          cardboxes.add( dealerCard10JLabel );
71          cardboxes.add( dealerCard11JLabel );
72          cardboxes.add( playerCard1JLabel );
73          cardboxes.add( playerCard2JLabel );
74          cardboxes.add( playerCard3JLabel );
75          cardboxes.add( playerCard4JLabel );
76          cardboxes.add( playerCard5JLabel );
77          cardboxes.add( playerCard6JLabel );
78          cardboxes.add( playerCard7JLabel );
79          cardboxes.add( playerCard8JLabel );
80          cardboxes.add( playerCard9JLabel );
81          cardboxes.add( playerCard10JLabel );
82          cardboxes.add( playerCard11JLabel );
83       } // fim do construtor
84
85       // joga a mão do carteador
86       private void dealerPlay()
87       {
88          try
89          {
90             // enquanto o valor da mão do carteador está abaixo de 17
91             // o carteador deve continuar a pedir cartas
92             String[] cards = dealerCards.split( "\t" );
93
94             // exibe as cartas do carteador
95             for ( int i = 0; i < cards.length; i++ )
96             {
97                displayCard( i, cards[i] );
98             }
99
100            while ( blackjackProxy.getHandValue( dealerCards ) < 17 )
101            {
102               String newCard = blackjackProxy.dealCard(); // distribui novo card
103               dealerCards += "\t" + newCard; // distribui novo card
104               displayCard( currentDealerCard, newCard );
```

```
105             ++currentDealerCard;
106             JOptionPane.showMessageDialog( this, "Dealer takes a card",
107                "Dealer's turn", JOptionPane.PLAIN_MESSAGE );
108          } // fim do while
109
110          int dealersTotal = blackjackProxy.getHandValue( dealerCards );
111          int playersTotal = blackjackProxy.getHandValue( playerCards );
112
113          // se o carteador estourou, o jogador ganha
114          if ( dealersTotal > 21 )
115          {
116             gameOver( GameStatus.WIN );
117             return;
118          } // fim do if
119
120          // se o carteador e o jogador estão abaixo de 21
121          // o escore mais alto ganha, escores iguais são um empate
122          if ( dealersTotal > playersTotal )
123          {
124             gameOver( GameStatus.LOSE );
125          }
126          else if ( dealersTotal < playersTotal )
127          {
128             gameOver( GameStatus.WIN );
129          }
130          else
131          {
132             gameOver( GameStatus.PUSH );
133          }
134       } // fim do try
135       catch ( Exception e )
136       {
137          e.printStackTrace();
138       } // fim do catch
139    } // fim do método dealerPlay
140
141    // exibe a carta representada por cardValue no JLabel especificado
142    private void displayCard( int card, String cardValue )
143    {
144       try
145       {
146          // recupere o JLabel correto de cardBoxes
147          JLabel displayLabel = cardboxes.get( card );
148
149          // se a string representando a carta estiver vazia, exibe a parte de trás da carta
150          if ( cardValue.equals( "" ) )
151          {
152             displayLabel.setIcon( new ImageIcon( getClass().getResource(
153                "/com/deitel/java/blackjackclient/" +
154                "blackjack_images/cardback.png" ) ) );
155             return;
156          } // fim do if
157
158          // recupere o valor de face da carta
159          String face = cardValue.substring( 0, cardValue.indexOf( " " ) );
160
161          // recupere o naipe da carta
162          String suit =
163             cardValue.substring( cardValue.indexOf( " " ) + 1 );
164
165          char suitLetter; // letra de naipe utilizada para formar o arquivo de imagem
166
167          switch ( Integer.parseInt( suit ) )
168          {
169             case 0: // copas
170                suitLetter = 'h';
171                break;
172             case 1: // ouros
173                suitLetter = 'd';
```

```
174                  break;
175               case 2: // paus
176                  suitLetter = 'c';
177                  break;
178               default: // espadas
179                  suitLetter = 's';
180                  break;
181            } // fim do switch
182
183            // configura a imagem para displayLabel
184            displayLabel.setIcon( new ImageIcon( getClass().getResource(
185               "/com/deitel/java/blackjackclient/blackjack_images/" +
186               face + suitLetter + ".png" ) ) );
187         } // fim do try
188         catch ( Exception e )
189         {
190            e.printStackTrace();
191         } // fim do catch
192      } // fim do método displayCard
193
194      // exibe todas as cartas do jogador e mostra uma mensagem apropriada
195      private void gameOver( GameStatus winner )
196      {
197         String[] cards = dealerCards.split( "\t" );
198
199         // exibe cartas do blackjackProxy
200         for ( int i = 0; i < cards.length; i++ )
201         {
202            displayCard( i, cards[i] );
203         }
204
205         // exibe a imagem de status apropriada
206         if ( winner == GameStatus.WIN )
207         {
208            statusJLabel.setText( "You win!" );
209         }
210         else if ( winner == GameStatus.LOSE )
211         {
212            statusJLabel.setText( "You lose." );
213         }
214         else if ( winner == GameStatus.PUSH )
215         {
216            statusJLabel.setText( "It's a push." );
217         }
218         else // vinte-e-um
219         {
220            statusJLabel.setText( "Blackjack!" );
221         }
222
223         // exibe escores finais
224         int dealersTotal = blackjackProxy.getHandValue( dealerCards );
225         int playersTotal = blackjackProxy.getHandValue( playerCards );
226         dealerTotalJLabel.setText( "Dealer: " + dealersTotal );
227         playerTotalJLabel.setText( "Player: " + playersTotal );
228
229         // reinicializa um novo jogo
230         standJButton.setEnabled( false );
231         hitJButton.setEnabled( false );
232         dealJButton.setEnabled( true );
233      } // fim do método gameOver
234
235      // O método initComponents é autogerado pelo NetBeans e é chamado
236      // a partir do construtor para inicializar a GUI. Esse método não é mostrado
237      // aqui para economizar espaço. Abra BlackjackGameJFrame.java nessa
238      // pasta de exemplo para visualizar o código gerado completo (linhas 235-541)
239
542      // trata o clique em dealJButton
543      private void dealJButtonActionPerformed(
544         java.awt.event.ActionEvent evt )
545      {
546         String card; // armazena uma carta temporariamente até ela ser adicionada a uma mão
```

```
547
548         // limpa imagens da carta
549         for ( int i = 0; i < cardboxes.size(); i++ )
550         {
551            cardboxes.get( i ).setIcon( null );
552         }
553
554         statusJLabel.setText( "" );
555         dealerTotalJLabel.setText( "" );
556         playerTotalJLabel.setText( "" );
557
558         // cria um novo baralho embaralhado na máquina remota
559         blackjackProxy.shuffle();
560
561         // distribui duas cartas ao jogador
562         playerCards = blackjackProxy.dealCard(); // adiciona a primeira carta à mão
563         displayCard( 11, playerCards ); // exibe a primeira carta
564         card = blackjackProxy.dealCard(); // distribui a segunda carta
565         displayCard( 12, card ); // exibe a segunda carta
566         playerCards += "\t" + card; // adiciona a segunda carta à mão
567
568         // distribui duas cartas para blackjackProxy, mas só mostra a primeira
569         dealerCards = blackjackProxy.dealCard(); // adiciona a primeira carta à mão
570         displayCard( 0, dealerCards ); // exibe a primeira carta
571         card = blackjackProxy.dealCard(); // distribui a segunda carta
572         displayCard( 1, "" ); // exibe a parte de trás da carta
573         dealerCards += "\t" + card; // adiciona a segunda carta à mão
574
575         standJButton.setEnabled( true );
576         hitJButton.setEnabled( true );
577         dealJButton.setEnabled( false );
578
579         // determina o valor das duas mãos
580         int dealersTotal = blackjackProxy.getHandValue( dealerCards );
581         int playersTotal = blackjackProxy.getHandValue( playerCards );
582
583         // se as duas mãos forem iguais a 21, isso é um empate
584         if ( playersTotal == dealersTotal && playersTotal == 21 )
585         {
586            gameOver( GameStatus.PUSH );
587         }
588         else if ( dealersTotal == 21 ) // blackjackProxy tem um vinte-e-um
589         {
590            gameOver( GameStatus.LOSE );
591         }
592         else if ( playersTotal == 21 ) // vinte-e-um
593         {
594            gameOver( GameStatus.BLACKJACK );
595         }
596
597         // a próxima carta para blackjackProxy tem o índice 2
598         currentDealerCard = 2;
599
600         // a próxima carta para o jogador tem o índice 13
601         currentPlayerCard = 13;
602      } // fim do método dealJButtonActionPerformed
603
604      // trata o clique em standJButton
605      private void hitJButtonActionPerformed(
606         java.awt.event.ActionEvent evt )
607      {
608         // obtém outra carta do jogador
609         String card = blackjackProxy.dealCard(); // distribui novo card
610         playerCards += "\t" + card; // adiciona carta à mão
611
612         // atualiza a GUI para exibir a nova carta
613         displayCard( currentPlayerCard, card );
614         ++currentPlayerCard;
615
616         // determine o novo valor da mão do jogador
617         int total = blackjackProxy.getHandValue( playerCards );
```

```
618
619         if ( total > 21 ) // o jogador estoura
620         {
621            gameOver( GameStatus.LOSE );
622         }
623         else if ( total == 21 ) // o jogador não pode pedir mais cartas
624         {
625            hitJButton.setEnabled( false );
626            dealerPlay();
627         } // fim do if
628      } // fim do método hitJButtonActionPerformed
629
630      // trata o clique em standJButton
631      private void standJButtonActionPerformed(
632         java.awt.event.ActionEvent evt )
633      {
634         standJButton.setEnabled( false );
635         hitJButton.setEnabled( false );
636         dealJButton.setEnabled( true );
637         dealerPlay();
638      } // fim do método standJButtonActionPerformed
639
640      // inicia a execução do aplicativo
641      public static void main( String args[] )
642      {
643         java.awt.EventQueue.invokeLater(
644            new Runnable()
645            {
646               public void run()
647               {
648                  new BlackjackGameJFrame().setVisible( true );
649               }
650            }
651         ); // fim da chamada para java.awt. EventQueue.invokeLater
652      } // fim de main
653
654      // Declaração de variáveis - não modifique
655      private javax.swing.JButton dealJButton;
656      private javax.swing.JLabel dealerCard10JLabel;
657      private javax.swing.JLabel dealerCard11JLabel;
658      private javax.swing.JLabel dealerCard1JLabel;
659      private javax.swing.JLabel dealerCard2JLabel;
660      private javax.swing.JLabel dealerCard3JLabel;
661      private javax.swing.JLabel dealerCard4JLabel;
662      private javax.swing.JLabel dealerCard5JLabel;
663      private javax.swing.JLabel dealerCard6JLabel;
664      private javax.swing.JLabel dealerCard7JLabel;
665      private javax.swing.JLabel dealerCard8JLabel;
666      private javax.swing.JLabel dealerCard9JLabel;
667      private javax.swing.JLabel dealerJLabel;
668      private javax.swing.JLabel dealerTotalJLabel;
669      private javax.swing.JButton hitJButton;
670      private javax.swing.JLabel playerCard10JLabel;
671      private javax.swing.JLabel playerCard11JLabel;
672      private javax.swing.JLabel playerCard1JLabel;
673      private javax.swing.JLabel playerCard2JLabel;
674      private javax.swing.JLabel playerCard3JLabel;
675      private javax.swing.JLabel playerCard4JLabel;
676      private javax.swing.JLabel playerCard5JLabel;
677      private javax.swing.JLabel playerCard6JLabel;
678      private javax.swing.JLabel playerCard7JLabel;
679      private javax.swing.JLabel playerCard8JLabel;
680      private javax.swing.JLabel playerCard9JLabel;
681      private javax.swing.JLabel playerJLabel;
682      private javax.swing.JLabel playerTotalJLabel;
683      private javax.swing.JButton standJButton;
684      private javax.swing.JLabel statusJLabel;
685      // Fim da declaração de variáveis
686   } // fim da classe BlackjackGameJFrame
```

a) A mão de quem dá as cartas e a mão do jogador depois que o usuário clica no JButton Deal.

b) A mão de quem dá as cartas e a do jogador depois que o usuário clica duas vezes em Hit e, então, clica em Stand. Nesse caso, o resultado é um empate.

c) A mão de quem dá as cartas e a mão do jogador depois que o usuário clica em Stand com base na mão inicial. Nesse caso, o jogador vence.

d) A mão de quem dá as cartas e a mão do jogador depois que o usuário fez o 21.

e) A mão de quem dá as cartas e a mão do jogador depois que o primeiro fez o 21.

Figura 31.17 | Jogo de vinte-e-um que utiliza o serviço Web `Blackjack`.

Com o JAX-WS, o aplicativo cliente deve indicar se ele quer permitir que o serviço Web mantenha informações de sessão. As linhas 50–51 no construtor realizam essa tarefa. Primeiro fazemos uma coerção do objeto SEI para o tipo de interface `BindingProvider`. Um **`BindingProvider`** permite ao cliente manipular as informações de solicitação que serão enviadas ao servidor. Essas informações são armazenadas em um objeto que implementa a interface **`RequestContext`**. `BindingProvider` e `RequestContext` são partes do framework que é criado pelo IDE quando você adiciona um cliente de serviço Web ao aplicativo. Em seguida, invocamos o **método `getRequestContext`** de `BindingProvider` para obter o objeto `RequestContext`. Então chamamos o **método `put`** de `RequestContext` para configurar a propriedade

```
BindingProvider.SESSION_MAINTAIN_PROPERTY
```

como `true`. Isso ativa o lado cliente do mecanismo de rastreamento de sessão para que o serviço Web saiba que cliente está invocando os métodos do serviço Web.

O método `gameOver` (linhas 195–233) exibe todas as cartas do carteador, mostra a mensagem apropriada em `statusJLabel` e exibe o número total de pontos finais do carteador e do jogador. O método `gameOver` recebe como um argumento um membro da enumeração `GameStatus` (definida nas linhas 25–31). A enumeração representa se o jogador empatou, perdeu ou ganhou o jogo; seus quatro membros são `PUSH`, `LOSE`, `WIN` e `BLACKJACK`.

Quando o jogador clica em `JButton` **Deal**, o método `dealJButtonActionPerformed` (linhas 543–602) limpa todos os `JLabels` que exibem cartas ou informações do status do jogo. Em seguida, as cartas são embaralhadas (linha 559), e o jogador e o carteador recebem duas cartas cada um (linhas 562–573). As linhas 580–581 somam então cada mão. Se o jogador e o carteador ambos obtiverem escores de 21, o programa chama o método `gameOver`, passando `GameStatus.PUSH` (linha 586). Se apenas o carteador tiver 21, o programa passa `GameStatus.LOSE` para o método `gameOver` (linha 590). Se apenas o jogador tiver 21 depois de as duas primeiras cartas serem distribuídas, o programa passa `GameStatus.BLACKJACK` para o método `gameOver` (linha 594).

Se `dealJButtonActionPerformed` não chamar `gameOver`, o jogador poderá pedir mais cartas clicando no `JButton` **Hit**, que chama `hitJButtonActionPerformed` nas linhas 605–628. Toda vez que um jogador clica em **Hit**, o programa dá ao jogador mais uma carta (linha 609) e a exibe na GUI (linha 613). Se o jogador exceder 21, o jogo termina e o jogador perde (linha 621). Se o jogador tiver exatamente 21, o jogador não poderá pedir mais cartas (linha 625), e o método `dealerPlay` é chamado (linha 626).

O método `dealerPlay` (linhas 86–139) exibe as cartas do carteador, distribui cartas ao carteador até que a mão do carteador tenha um valor de 17 ou mais (linhas 100–108). Se o carteador exceder 21, o jogador ganha (linha 116); do contrário, os valores das mãos são comparados, e `gameOver` é chamado com o argumento apropriado (linhas 122–133).

Clicar em `JButton` **Stand** indica que um jogador não quer receber outra carta. O método `standJButtonActionPerformed` (linhas 631–638) desativa os botões **Hit** e **Stand**, ativa o botão **Deal** e então chama o método `dealerPlay`.

O método `displayCard` (linhas 142–192) atualiza a GUI para exibir uma carta recém-distribuída. O método recebe como argumentos um índice de inteiros para o `JLabel` em `ArrayList` que precisa ter sua imagem configurada e uma `String` representando a carta. Uma `String` vazia indica que queremos exibir a carta virada para baixo. Se o método `displayCard` receber uma `String` que não está vazia, o programa extrai a face e o naipe de `String` e utiliza essas informações para exibir a imagem correta. A instrução `switch` (linhas 167–181) converte o número que representa o naipe em um número inteiro e atribui o caractere apropriado à variável `suitLetter` (`h` para *hearts* [copas], `d` para *diamonds* [ouros], `c` para *clubs* [paus] e `s` para *spades* [espadas]). O caractere em `suitLetter` é utilizado para completar o nome de arquivo da imagem (linhas 184–186). *Você deve adicionar a pasta `blackjack_images` ao seu projeto para que as linhas 152–154 e 184–186 possam acessar as imagens apropriadamente.* Para fazer isso, copie a pasta `blackjack_images` da pasta de exemplos deste capítulo e cole-a na pasta `src\com\deitel\java\blackjackclient` do projeto.

Você acabou de aprender a configurar um serviço Web para suportar tratamento de sessão a fim de poder monitorar do estado de sessão de cada cliente. Você também aprendeu a ativar um aplicativo cliente desktop para fazer parte do rastreamento de sessão. Aprenderá agora a acessar um banco de dados a partir de um serviço Web e a consumir um serviço Web a partir de um aplicativo Web cliente.

31.10 Consumindo um serviço Web baseado em SOAP orientado a banco de dados

Nossos exemplos anteriores acessavam os serviços Web a partir de aplicativos desktop criados no NetBeans. Contudo, podemos utilizá-los de uma maneira igualmente fácil em aplicativos Web criados com o NetBeans. Na verdade, como negócios baseados na Web estão se tornando cada vez mais disseminados, é comum que aplicativos Web consumam serviços Web. Nesta seção, apresentamos um serviço Web de reserva de passagens aéreas que recebe informações sobre o tipo de poltrona que um cliente deseja reservar e faz uma reserva se essa poltrona estiver disponível. Mais adiante na seção, apresentamos um aplicativo Web que permite a um cliente especificar uma solicitação de reserva e então utiliza o serviço Web de reservas de passagens aéreas para tentar executar a solicitação.

31.10.1 Criando o banco de dados `Reservation`

Nesse exemplo, nosso serviço Web utiliza um banco de dados `Reservation` que contém uma única tabela nomeada `Seats` para localizar uma poltrona correspondente à solicitação de um cliente. Para construir o banco de dados `Reservation`, revise os passos apresentados na Seção 30.2.1 para construir o banco de dados `AddressBook`. O diretório de exemplos deste capítulo contém o script SQL `Seats.sql` para construir a tabela `seats` e preenchê-la com dados de exemplo. Os dados de exemplos são mostrados na Figura 31.18.

number	location	class	taken
1	Aisle	Economy	0
2	Aisle	Economy	0
3	Aisle	First	0
4	Middle	Economy	0
5	Middle	Economy	0
6	Middle	First	0
7	Window	Economy	0
8	Window	Economy	0
9	Window	First	0
10	Window	First	0

Figura 31.18 | Dados da tabela `seats`.

Criando o serviço Web de reservas

Você agora pode criar um serviço Web que utiliza o banco de dados `Reservation` (Figura 31.19). O serviço Web de reservas de passagens áreas tem um único método Web — `reserve` (linhas 26–78) — que pesquisa a tabela `Seats` para localizar uma poltrona correspondente à solicitação de um usuário. O método recebe dois argumentos — uma `String` representando o tipo de poltrona desejado (isto é, `"Window"`, `"Middle"` ou `"Aisle"`) e uma `String` representando o tipo de classe desejado (isto é, `"Economy"` ou `"First"`). Se ele localizar uma poltrona apropriada, o método `reserve` atualiza o banco de dados para fazer a reserva e retorna `true`; do contrário, nenhuma reserva é feita, e o método retorna `false`. Observe que as instruções nas linhas 34–39 e linhas 45–48 que consultam e atualizam o banco de dados utilizam objetos dos tipos JDBC `ResultSet` e `PreparedStatement`.

Observação de engenharia de software 31.1

É altamente recomendável utilizar `PreparedStatements` para criar instruções SQL a fim de proteger-se contra os chamados ataques de injeção de SQL nos quais o código executável é inserido no código SQL. O site `www.owasp.org/index.php/Preventing_SQL_Injection_in_Java` fornece um resumo dos ataques de injeção de SQL e modos de se proteger deles.

```
1  // Figura 31.19: Reservation.java
2  // Serviço Web de reservas de passagens aéreas.
3  package com.deitel.java.reservation;
4
5  import java.sql.Connection;
6  import java.sql.DriverManager;
7  import java.sql.PreparedStatement;
8  import java.sql.ResultSet;
9  import java.sql.SQLException;
10 import javax.jws.WebMethod;
11 import javax.jws.WebParam;
12 import javax.jws.WebService;
13
14 @WebService( name = "Reservation", serviceName = "ReservationService" )
15 public class Reservation
16 {
17    private static final String DATABASE_URL =
18       "jdbc:derby://localhost:1527/Reservation";
19    private static final String USERNAME = "test";
20    private static final String PASSWORD = "test";
21    private Connection connection;
22    private PreparedStatement lookupSeat;
23    private PreparedStatement reserveSeat;
24
25    // um WebMethod que pode reservar uma poltrona
26    @WebMethod( operationName = "reserve" )
27    public boolean reserve( @WebParam( name = "seatType" ) String seatType,
28       @WebParam( name = "classType" ) String classType )
29    {
```

```
30          try
31          {
32             connection = DriverManager.getConnection(
33                DATABASE_URL, USERNAME, PASSWORD );
34             lookupSeat = connection.prepareStatement(
35                "SELECT \"number\" FROM \"seats\" WHERE (\"taken\" = 0) " +
36                "AND (\"location\" = ?) AND (\"class\" = ?)" );
37             lookupSeat.setString( 1, seatType );
38             lookupSeat.setString( 2, classType );
39             ResultSet resultSet = lookupSeat.executeQuery();
40
41             // se a poltrona solicitada estiver disponível, faz a reserva
42             if ( resultSet.next() )
43             {
44                int seat = resultSet.getInt( 1 );
45                reserveSeat = connection.prepareStatement(
46                   "UPDATE \"seats\" SET \"taken\"=1 WHERE \"number\"=?" );
47                reserveSeat.setInt( 1, seat );
48                reserveSeat.executeUpdate();
49                return true;
50             } // fim do if
51
52             return false;
53          } // fim do try
54          catch ( SQLException e )
55          {
56             e.printStackTrace();
57             return false;
58          } // fim do catch
59          catch ( Exception e )
60          {
61             e.printStackTrace();
62             return false;
63          } // fim do catch
64          finally
65          {
66             try
67             {
68                lookupSeat.close();
69                reserveSeat.close();
70                connection.close();
71             } // fim do try
72             catch ( Exception e )
73             {
74                e.printStackTrace();
75                return false;
76             } // fim do catch
77          } // fim de finally
78       } // fim da reserva WebMethod
79    } // fim da classe Reservation
```

Figura 31.19 | Serviço Web de reservas de passagens aéreas.

O nosso banco de dados contém quatro colunas — o número de poltronas (isto é, `1`-`10`), o tipo de poltrona (isto é, `Window`, `Middle` ou `Aisle`), o tipo de classe (isto é, `Economy` ou `First`) e uma coluna que contém `1` (verdadeiro) ou `0` (falso) para indicar se a poltrona já foi reservada. As linhas 34–39 recuperam os números de poltrona de quaisquer poltronas disponíveis correspondentes à poltrona e o tipo de classe solicitados. Essa instrução preenche o `resultSet` com os resultados da consulta.

```
SELECT "number"
FROM "seats"
WHERE ("taken" = 0) AND ("type" = tipo) AND ("class" = classe)
```

Os parâmetros *type* e *class* na consulta são substituídos pelos valores dos parâmetros `seatType` e `classType` do método `reserve`. Ao utilizar as ferramentas NetBeans para criar uma tabela de banco de dados e suas colunas, as ferramentas NetBeans inserem automaticamente os nomes de tabela e coluna entre aspas duplas. Por essa razão, você precisa colocar os nomes de tabela e coluna entre aspas duplas nas instruções SQL que interagem com o banco de dados `Reservation`.

Se `resultSet` não estiver vazio (isto é, pelo menos uma poltrona está disponível que corresponde aos critérios selecionados), a condição na linha 42 será `true` e o serviço Web fará a reserva do primeiro número de poltrona correspondente. Lembre-se de que o método `ResultSet next` retorna `true` se existir uma linha não vazia, e posiciona o cursor nessa linha. Obtemos o número de poltrona (linha 44) acessando a primeira coluna de `resultSet` (isto é, `resultSet.getInt(1)` — a primeira coluna na linha). As linhas 45–48 configuram uma `PreparedStatement` e executam a SQL:

```
UPDATE "seats"
SET "taken" = 1
WHERE ("number" = número)
```

que marca a poltrona como reservada no banco de dados. O parâmetro *number* é substituído pelo valor de `seat`. O método `reserve` retorna `true` (linha 49) para indicar que a reserva foi bem-sucedida. Se não houver uma poltrona correspondente, ou se uma exceção ocorreu, o método `reserve` retornará `false` (linhas 52, 57, 62 e 75) para indicar que nenhuma poltrona correspondeu com a solicitação do usuário.

31.10.2 Criando um aplicativo Web para interagir com o serviço `Reservation`

Esta seção apresenta um aplicativo Web `ReservationClient` que consome o serviço Web `Reservation`. O aplicativo permite aos usuários selecionar poltronas com base na classe (`"Economy"` ou `"First"`) e localização (`"Aisle"`, `"Middle"` ou `"Window"`) e então submeter suas solicitações ao serviço Web de reservas de passagens aéreas. Se a solicitação de banco de dados não for bem-sucedida, o aplicativo instrui o usuário a modificar a solicitação e tentar novamente. O aplicativo apresentado aqui foi construído utilizando as técnicas apresentadas nos capítulos 6–30. Supomos que você já tenha lido esses capítulos e, portanto, sabe como construir a GUI de um aplicativo Web, criar handlers de evento e adicionar propriedades ao bean de sessão de um aplicativo Web (Seção 29.7.2).

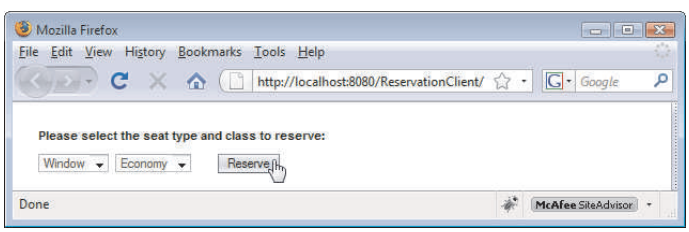
`Reserve.jsp`

`Reserve.jsp` (Figura 31.20) define dois `DropDownLists` e um `Button`. O `seatTypeDropDown` (linhas 23–28) exibe todos os tipos de poltrona que os usuários podem selecionar. O `classTypeDropDownList` (linhas 29–35) fornece escolhas ao tipo de classe. Quando o usuário faz uma seleção a partir de cada um destes, os handlers de evento correspondentes armazenam os valores selecionados no bean de sessão. Adicionamos as propriedades `seatType` e `classType` ao bean de sessão para essa finalidade. Os usuários clicam em `reserveButton` (linhas 36–40) para submeter solicitações depois de fazer seleções a partir de `DropDownLists`. A página também define três `Labels` — `instructionLabel` (linhas 18–22) para exibir instruções, `successLabel` (linhas 41–44) para indicar uma reserva bem-sucedida e `errorLabel` (linhas 45–49) para exibir uma mensagem se não houver nenhuma poltrona que corresponda à seleção do usuário. O arquivo de bean de página (Figura 31.21) contém os handlers de evento para `seatTypeDropDown`, `classTypeDropDown` e `reserveButton`.

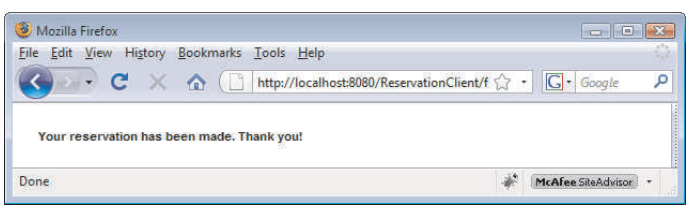
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Figura 31.20 Reserve.jsp -->
3  <!-- JSP que permite a um usuário selecionar uma poltrona -->
4  <jsp:root version="2.1" xmlns:f="http://java.sun.com/jsf/core"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:jsp="http://java.sun.com/JSP/Page"
7     xmlns:webuijsf="http://www.sun.com/webui/webuijsf">
8     <jsp:directive.page contentType="text/html;charset=UTF-8"
9        pageEncoding="UTF-8"/>
10    <f:view>
11       <webuijsf:page id="page1">
12          <webuijsf:html id="html1">
13             <webuijsf:head id="head1">
14                <webuijsf:link id="link1" url="/resources/stylesheet.css"/>
15             </webuijsf:head>
16             <webuijsf:body id="body1" style="-rave-layout: grid">
17                <webuijsf:form id="form1">
18                   <webuijsf:label binding="#{Reserve.instructionLabel}"
19                      id="instructionLabel"
20                      style="left: 24px; top: 24px; position: absolute"
21                      text="Please select the seat type and class to
22                         reserve:"/>
23                   <webuijsf:dropDown binding="#{Reserve.seatTypeDropDown}"
24                      id="seatTypeDropDown" items="#{Reserve.
25                         seatTypeDropDownDefaultOptions.options}"
26                      style="left: 24px; top: 48px; position: absolute"
27                      valueChangeListenerExpression=
28                         "#{Reserve.seatTypeDropDown_processValueChange}"/>
```

```
                    <webuijsf:dropDown
                       binding="#{Reserve.classTypeDropDown}"
                       id="classTypeDropDown" items="#{Reserve.
                          classTypeDropDownDefaultOptions.options}"
                       style="left: 96px; top: 48px; position: absolute"
                       valueChangeListenerExpression="#{Reserve.
                          classTypeDropDown_processValueChange}"/>
                    <webuijsf:button
                       actionExpression="#{Reserve.reserveButton_action}"
                       binding="#{Reserve.reserveButton}" id="reserveButton"
                       style="left: 191px; top: 48px; position: absolute"
                       text="Reserve"/>
                    <webuijsf:label binding=v
                       id="successLabel" rendered="false" style="left: 24px;
                          top: 24px; position: absolute"
                       text="Your reservation has been made. Thank you!"/>
                    <webuijsf:label binding="#{Reserve.errorLabel}"
                       id="errorLabel" rendered="false" style="color: red;
                          left: 24px; top: 96px; position: absolute"
                       text="This type of seat is not available. Please
                          modify your request and try again."/>
                 </webuijsf:form>
              </webuijsf:body>
           </webuijsf:html>
        </webuijsf:page>
     </f:view>
  </jsp:root>
```

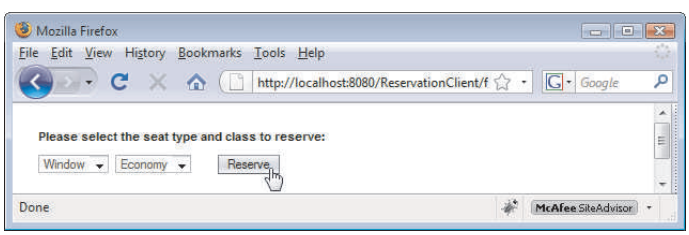
a) Selecionando um lugar



b) Lugar reservado com sucesso



c) Tentando reservar outro lugar na janela na classe econômica quando não há esse lugar disponível



d) Nenhum lugar corresponde com o tipo de lugar e de classe solicitados

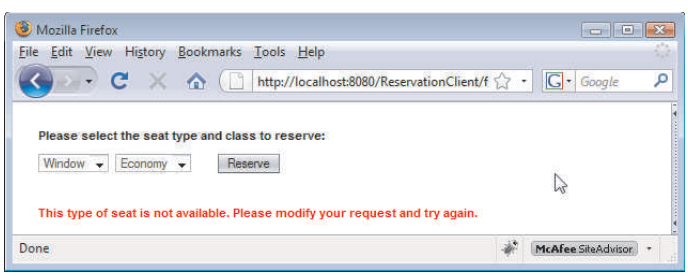


Figura 31.20 | O JSP que permite a um usuário selecionar uma poltrona.

Reserve.java

A Figura 31.21 contém o código de bean de página que fornece a lógica para `Reserve.jsp`. Como discutido na Seção 29.5.2, a classe que representa o bean da página estende `AbstractPageBean`. Quando o usuário seleciona um valor em um dos `DropDownLists`, o handler de evento correspondente — `seatTypeDropDown_process-ValueChange` (linhas 199–204) ou `classTypeDropDown_processValueChange` (linhas 207–212) — é chamado para configurar as propriedades de sessão `seatType` e `classType`, que adicionamos ao bean de sessão do aplicativo Web. Os valores dessas propriedades são utilizados como argumentos na chamada ao método `reserve` do serviço Web. Quando o usuário clica em **Reserve** na página JSP, o handler de evento `reserveButton_action` (linhas 215–248) executa. As linhas 219–221 utilizam o objeto SEI (criado nas linhas 40–41) para invocar o método `reserve` do serviço Web, passando o tipo de poltrona selecionado e o tipo de classe como argumentos. Se `reserve` retornar `true`, as linhas 225–230 ocultarão os componentes GUI no JSP e exibirão o `successLabel` (linha 229) para agradecer ao usuário por fazer uma reserva; do contrário, as linhas 234–239 asseguram que os componentes GUI permaneçam visíveis, exibem o `errorLabel` (linha 239) para notificar o usuário de que o tipo de poltrona solicitado não está disponível e instruir o usuário a tentar novamente.

```
1   // Figura 31.21: Reserve.java
2   // bean de página para reserva de poltrona pelo cliente.
3   package reservationclient;
4
5   import com.deitel.java.reservation.Reservation;
6   import com.deitel.java.reservation.ReservationService;
7   import com.sun.rave.web.ui.appbase.AbstractPageBean;
8   import com.sun.webui.jsf.component.Button;
9   import com.sun.webui.jsf.component.DropDown;
10  import com.sun.webui.jsf.component.Label;
11  import com.sun.webui.jsf.model.SingleSelectOptionsList;
12  import javax.faces.FacesException;
13  import javax.faces.event.ValueChangeEvent;
14
15  public class Reserve extends AbstractPageBean
16  {
17     // referencia o objeto SEI (isto é, o proxy)
18     private Reservation reservationServiceProxy; // referência a proxy
19
20     // Managed Component Definition"
21     private void _init() throws Exception
22     {
23        seatTypeDropDownDefaultOptions.setOptions(
24           new com.sun.webui.jsf.model.Option[]
25           {
26              new com.sun.webui.jsf.model.Option( "Aisle", "Aisle" ),
27              new com.sun.webui.jsf.model.Option( "Middle", "Middle" ),
28              new com.sun.webui.jsf.model.Option( "Window", "Window" )
29           }
30        );
31        classTypeDropDownDefaultOptions.setOptions(
32           new com.sun.webui.jsf.model.Option[]
33           {
34              new com.sun.webui.jsf.model.Option( "Economy", "Economy" ),
35              new com.sun.webui.jsf.model.Option( "First", "First" )
36           }
37        );
38
39        // obtém a SEI
40        ReservationService reservationService = new ReservationService();
41        reservationServiceProxy = reservationService.getReservationPort();
42     } // fim do método _init
43
44     // As linhas 44-197 do código autogerado foram removidas para economizar
45     // espaço. O código completo está disponível na pasta desse exemplo.
46
198    // armazena o tipo de poltrona selecionado no bean de sessão
199    public void seatTypeDropDown_processValueChange(
200       ValueChangeEvent event )
201    {
202       getSessionBean1().setSeatType(
203          ( String ) seatTypeDropDown.getSelected() );
204    } // fim do método seatTypeDropDown_processValueChange
```

```
205
206       // armazena a classe selecionada no bean de sessão
207       public void classTypeDropDown_processValueChange(
208          ValueChangeEvent event )
209       {
210          getSessionBean1().setClassType(
211             ( String ) classTypeDropDown.getSelected() );
212       } // fim do método classTypeDropDown_processValueChange
213
214       // invoca o serviço Web quando o usuário clica no botão Reserve
215       public String reserveButton_action()
216       {
217          try
218          {
219             boolean reserved = reservationServiceProxy.reserve(
220                getSessionBean1().getSeatType(),
221                getSessionBean1().getClassType() );
222
223             if ( reserved ) // exibe successLabel; oculta todos os outros
224             {
225                instructionLabel.setRendered( false );
226                seatTypeDropDown.setRendered( false );
227                classTypeDropDown.setRendered( false );
228                reserveButton.setRendered( false );
229                successLabel.setRendered( true );
230                errorLabel.setRendered( false );
231             } // fim do if
232             else // exibe todos, exceto successLabel
233             {
234                instructionLabel.setRendered( true );
235                seatTypeDropDown.setRendered( true );
236                classTypeDropDown.setRendered( true );
237                reserveButton.setRendered( true );
238                successLabel.setRendered( false );
239                errorLabel.setRendered( true );
240             } // fim de else
241          } // fim do try
242          catch ( Exception e )
243          {
244             e.printStackTrace();
245          } // fim do catch
246
247          return null;
248       } // fim do método reserveButton_action
249    } // fim da classe Reserve
```

Figura 31.21 | Bean de página para reserva de poltrona pelo cliente.

31.11 Gerador de equação: retornando tipos definidos pelo usuário

A maioria dos serviços Web que demonstramos recebia e retornava instâncias de tipo primitivo. Também é possível processar instâncias dos tipos de classe em um serviço Web. Esses tipos podem ser passados a ou retornados de métodos de serviço Web.

Esta seção apresenta um serviço Web `EquationGenerator` RESTful que gera equações aritméticas aleatórias do tipo `Equation`. O cliente é um aplicativo de ensino de matemática que aceita informações sobre a pergunta que o usuário deseja tentar (adição, subtração ou multiplicação) e o nível de habilidade do usuário (1 especifica equações que utilizam números de 1 a 9, 2 especifica equações que envolvem números de 10 a 99, e 3 especifica equações que contêm números de 100 a 999). O serviço Web gera então uma equação que consiste em números aleatórios no intervalo apropriado. O aplicativo cliente recebe a `Equation` e exibe a pergunta de exemplo ao usuário.

Definindo a classe `Equation`

Definimos a classe `Equation` na Figura 31.22. Todos os programas nesta seção têm uma cópia dessa classe no pacote correspondente. Exceto pelo nome do pacote, a classe é idêntica em cada projeto, portanto a mostramos somente uma vez. Observe que, como a classe `TextMessage` utilizada antes, as cópias do lado do servidor e do lado do cliente da classe `Equation` não estão relacionados entre si. O único requisito para que a serialização e desserialização funcionem com as classes JAXB e Gson é que a classe `Equation` deve ter as mesmas propriedades `public` tanto no servidor como no cliente. Essas propriedades podem ser variáveis de instância públicas ou variáveis de instância privadas que têm os métodos *set* e *get* correspondentes.

```
1  // Figura 31.22: Equation.java
2  // Classe Equation que contém informações sobre uma equação.
3  package com.deitel.equationgeneratorxml;
4
5  public class Equation
6  {
7     private int leftOperand;
8     private int rightOperand;
9     private int result;
10    private String operationType;
11
12    // necessário construtor sem argumento
13    public Equation()
14    {
15       this( 0, 0, "add" );
16    } // fim do construtor sem argumentos
17
18    // o construtor que recebe o tipo de operação e os operandos
19    public Equation( int leftValue, int rightValue, String type )
20    {
21       leftOperand = leftValue;
22       rightOperand = rightValue;
23
24       // determina o resultado
25       if ( type.equals( "add" ) ) // adição
26       {
27          result = leftOperand + rightOperand;
28          operationType = "+";
29       } // fim do if
30       else if ( type.equals( "subtract" ) ) // subtração
31       {
32          result = leftOperand - rightOperand;
33          operationType = "-";
34       } // fim do if
35       else // multiplicação
36       {
37          result = leftOperand * rightOperand;
38          operationType = "*";
39       } // fim de else
40    } // fim do construtor com três argumentos
41
42    // obtém o leftOperand
43    public int getLeftOperand()
44    {
45       return leftOperand;
46    } // fim do método getLeftOperand
47
48    // setter necessário
49    public void setLeftOperand( int value )
50    {
51       leftOperand = value;
52    } // fim do método setLeftOperand
53
54    // obtém o rightOperand
55    public int getRightOperand()
56    {
57       return rightOperand;
58    } // fim do método getRightOperand
59
60    // setter necessário
61    public void setRightOperand( int value )
62    {
63       rightOperand = value;
64    } // fim do método setRightOperand
65
66    // obtém o resultValue
67    public int getResult()
68    {
```

```
69          return result;
70       } // fim do método getResult
71
72       // setter necessário
73       public void setResult( int value )
74       {
75          result = value;
76       } // fim do método setResult
77
78       // obtém o operationType
79       public String getOperationType()
80       {
81          return operationType;
82       } // fim do método getOperationType
83
84       // setter necessário
85       public void setOperationType( String value )
86       {
87          operationType = value;
88       } // fim do método setOperationType
89
90       // retorna o lado esquerdo da equação como uma String
91       public String getLeftHandSide()
92       {
93          return leftOperand + " " + operationType + " " + rightOperand;
94       } // fim do método getLeftHandSide
95
96       // retorna o lado direito da equação como uma String
97       public String getRightHandSide()
98       {
99          return "" + result;
100      } // fim do método getRightHandSide
101
102      // retorna uma representação String de uma Equation
103      public String toString()
104      {
105         return getLeftHandSide() + " = " + getRightHandSide();
106      } // fim do método toString
107   } // fim da classe Equation
```

Figura 31.22 | A classe `Equation` que contém informações sobre uma equação.

As linhas 19–40 definem um construtor que recebe dois `int`s representando os operandos esquerdos e direitos, e uma `String` representando a operação aritmética. O construtor armazena essas informações e então calcula o resultado. O construtor sem parâmetros (linhas 13–16) chama o construtor de três argumentos (linhas 19–40) e passa valores padrão.

A classe `Equation` define os métodos *get* e *set* para variáveis de instância `leftOperand` (linhas 43–52), `rightOperand` (linhas 55–64), `result` (linhas 67–76) e `operationType` (linhas 79–88). Ela também fornece métodos *get* para os lados esquerdo e direito da equação e um método `toString` que retorna a equação inteira como uma `String`. Uma variável de instância só pode ser serializada se ela tiver um método *get* e um *set*. Como os diferentes lados da equação e o resultado de `toString` podem ser gerados a partir de outras variáveis de instância, não há necessidade de enviá-los pela rede. O cliente nesse estudo de caso não utiliza o método `getRightHandSide`, mas nós o incluímos aqui no caso de futuros clientes optarem por utilizá-lo.

31.11.1 Criando o serviço Web XML baseado em REST `EquationGenerator`

A Figura 31.23 apresenta a classe `EquationGeneratorXML` do serviço Web para criar `Equações` geradas aleatoriamente. O método `getXml` (linhas 19–38) recebe dois parâmetros — uma `String` representando a operação matemática (`"add"`, `"subtract"` ou `"multiply"`) e um `int` representando o nível de dificuldade. O JAX-RS converte automaticamente os argumentos no tipo correto e retornará um erro "not found" ao cliente se o argumento não puder ser convertido de `String` para o tipo de destino. Os tipos suportados para conversão incluem tipos inteiros, tipos de ponto flutuante, `boolean` e as classes empacotadoras de tipo correspondentes.

```
1  // Figura 31.23: EquationGeneratorXMLResource.java
2  // O gerador de equação RESTful que retorna a XML.
3  package com.deitel.equationgeneratorxml;
4
5  import java.io.StringWriter;
```

```
6  import java.util.Random;
7  import javax.ws.rs.GET;
8  import javax.ws.rs.Path;
9  import javax.ws.rs.PathParam;
10 import javax.ws.rs.Produces;
11 import javax.xml.bind.JAXB; // classe utilitária para operações JAXB comuns
12
13 @Path( "equation" )
14 public class EquationGeneratorXMLResource
15 {
16    static Random randomObject = new Random(); // gerador de número aleatório
17
18    // recupera uma equação formatada como XML
19    @GET
20    @Path( "{operation}/{level}" )
21    @Produces( "application/xml" )
22    public String getXml( @PathParam( "operation" ) String operation,
23       @PathParam( "level" ) int level )
24    {
25       // calcula os valores mínimos e máximos dos números
26       int minimum = ( int ) Math.pow( 10, level - 1 );
27       int maximum = ( int ) Math.pow( 10, level );
28
29       // cria os números no lado esquerdo da equação
30       int first = randomObject.nextInt( maximum - minimum ) + minimum;
31       int second = randomObject.nextInt( maximum - minimum ) + minimum;
32
33       // cria o objeto Equation e o empacota na XML
34       Equation equation = new Equation( first, second, operation );
35       StringWriter writer = new StringWriter(); // saída da XML aqui
36       JAXB.marshal( equation, writer ); // grava Equation em StringWriter
37       return writer.toString(); // retorna a string XML
38    } // fim do método getXml
39 } // fim da classe EquationGeneratorXMLResource
```

Figura 31.23 | Um gerador de equação REST que retorna XML.

O método `getXml` primeiro determina os valores mínimos (inclusivamente) e máximos (exclusivamente) para os números na equação que ele retornará (linhas 26–27). Ele então utiliza um membro estático da classe `Random` (linha 16) para gerar dois números aleatórios nesse intervalo (linhas 30–31). A linha 34 cria um objeto `Equation`, passando esses dois números e a operação solicitada ao construtor. O método `getXml` utiliza então JAXB para converter o objeto `Equation` em XML (linha 36), que é gerado para o `StringWriter` criado na linha 35. Por fim, ele recupera os dados que foram gravados em `StringWriter` e retorna-os ao cliente. [*Nota*: Reimplementaremos esse serviço Web com o JSON na Seção 31.11.3.]

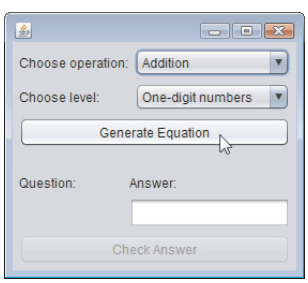
31.11.2 Consumindo o serviço Web XML baseado em REST EquationGenerator

O aplicativo `EquationGeneratorXMLClient` (Figura 31.24) recupera um objeto `Equation` formatado como XML a partir do serviço Web `EquationGeneratorXML`. O aplicativo cliente exibe então o lado esquerdo de `Equation` e espera que o usuário avalie a expressão e insira o resultado.

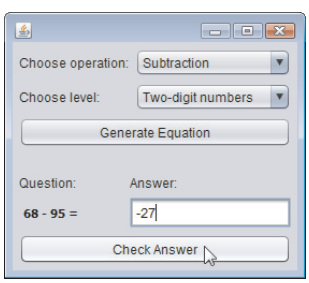
```
1  // Figura 31.24: EquationGeneratorXMLClientJFrame.java
2  // Programa de ensino de matemática que utiliza REST e XML para gerar equações.
3  package com.deitel.equationgeneratorxmlclient;
4
5  import javax.swing.JOptionPane;
6  import javax.xml.bind.JAXB; // classe utilitária para operações JAXB comuns
7
8  public class EquationGeneratorXMLClientJFrame extends javax.swing.JFrame
9  {
10    private String operation = "add"; // operação feita pelo usuário é testada em
11    private int difficulty = 1; // 1, 2 ou 3 dígitos em cada número
12    private int answer; // resposta correta à pergunta
13
14    // construtor sem argumento
15    public EquationGeneratorXMLClientJFrame()
16    {
```

```
17          initComponents();
18       } // fim do construtor sem argumentos
19
20       // O método initComponents é autogerado pelo NetBeans e é chamado
21       // a partir do construtor para inicializar a GUI. Esse método não é mostrado
22       // aqui para economizar espaço. Abra EquationGeneratorXMLClientJFrame.java na
23       // pasta desse exemplo para examinar o código gerado completo.
24
143      // determina se o usuário respondeu corretamente
144      private void checkAnswerJButtonActionPerformed(
145         java.awt.event.ActionEvent evt)
146      {
147         if ( answerJTextField.getText().equals( "" ) )
148         {
149            JOptionPane.showMessageDialog(
150               this, "Please enter your answer." );
151         } // fim do if
152
153         int userAnswer = Integer.parseInt( answerJTextField.getText() );
154
155         if ( userAnswer == answer )
156         {
157            equationJLabel.setText( "" ); // limpa o rótulo
158            answerJTextField.setText( "" ); // limpa o campo de texto
159            checkAnswerJButton.setEnabled( false );
160            JOptionPane.showMessageDialog( this, "Correct! Good Job!",
161               "Correct", JOptionPane.PLAIN_MESSAGE );
162         } // fim do if
163         else
164         {
165            JOptionPane.showMessageDialog( this, "Incorrect. Try again.",
166               "Incorrect", JOptionPane.PLAIN_MESSAGE );
167         } // fim de else
168      } // fim do método checkAnswerJButtonActionPerformed
169
170      // recupera a equação do serviço Web e exibe o lado esquerdo ao usuário
171      private void generateJButtonActionPerformed(
172         java.awt.event.ActionEvent evt)
173      {
174         try
175         {
176            String url = String.format( "http://localhost:8080/" +
177               "EquationGeneratorXML/resources/equation/%s/%d",
178               operation, difficulty );
179
180            // converte a XML de volta em um objeto Equation
181            Equation equation = JAXB.unmarshal( url, Equation.class );
182
183            answer = equation.getResult();
184            equationJLabel.setText( equation.getLeftHandSide() + " =" );
185            checkAnswerJButton.setEnabled( true );
186         } // fim do try
187         catch ( Exception exception )
188         {
189            exception.printStackTrace();
190         } // fim do catch
191      } // fim do método generateJButtonActionPerformed
192
193      // obtém a operação matemática selecionada pelo usuário
194      private void operationJComboBoxItemStateChanged(
195         java.awt.event.ItemEvent evt)
196      {
197         String item = ( String ) operationJComboBox.getSelectedItem();
198
199         if ( item.equals( "Addition" ) )
200            operation = "add"; // o usuário selecionou adição
201         else if ( item.equals( "Subtraction" ) )
202            operation = "subtract"; // o usuário selecionou subtração
203         else
```

```
204             operation = "multiply"; // o usuário selecionou multiplicação
205          } // fim do método operationJComboBoxItemStateChanged
206
207          // obtém o nível de dificuldade selecionado pelo usuário
208          private void levelJComboBoxItemStateChanged(
209             java.awt.event.ItemEvent evt)
210          {
211             // índice inicia em 0, assim adiciona 1 para obter o nível de dificuldade
212             difficulty = levelJComboBox.getSelectedIndex() + 1;
213          } // fim do método levelJComboBoxItemStateChanged
214
215          // método main inicia a execução
216          public static void main(String args[])
217          {
218             java.awt.EventQueue.invokeLater(
219                new Runnable()
220                {
221                   public void run()
222                   {
223                      new EquationGeneratorXMLClientJFrame().setVisible( true );
224                   } // fim do método run
225                } // fim da classe interna anônima
226             ); // fim da chamada a java.awt. EventQueue.invokeLater
227          } // fim de main
228
229          // Declaração de variáveis - não modifique
230          private javax.swing.JLabel answerJLabel;
231          private javax.swing.JTextField answerJTextField;
232          private javax.swing.JButton checkAnswerJButton;
233          private javax.swing.JLabel equationJLabel;
234          private javax.swing.JButton generateJButton;
235          private javax.swing.JComboBox levelJComboBox;
236          private javax.swing.JLabel levelJLabel;
237          private javax.swing.JComboBox operationJComboBox;
238          private javax.swing.JLabel operationJLabel;
239          private javax.swing.JLabel questionJLabel;
240          // Fim da declaração de variáveis
241       } // fim da classe EquationGeneratorXMLClientJFrame
```



a) Gerando uma equação simples.



b) Respondendo uma equação de subtração mais complicada.

Figura 31.24 | Programa de ensino de matemática utilizando REST e XML para gerar equações.

A configuração padrão do nível de dificuldade é 1, mas o usuário pode alterar isso escolhendo um nível no `JComboBox` **Choose level**. Alterar o valor selecionado invoca o handler de evento `levelJComboBoxItemStateChanged` (linhas 208–213), que configura a variável `difficulty` de instância com o nível selecionado pelo usuário. Embora a configuração padrão para o tipo de pergunta seja **Addition**, o usuário também pode alterar isso escolhendo no `JComboBox` **Choose operation**. Isso invoca o handler de evento `operationJComboBoxItemStateChanged` nas linhas 194–205, que atribui à variável de instância `operation` a `String` correspondente à seleção do usuário.

O handler de evento para `generateJButton` (linhas 171–191) constrói o URL para invocar o serviço Web, em seguida passa esse URL para o método `unmarshal`, com uma instância de `Class<Equation>`, para que o JAXB possa converter a XML em um objeto `Equation` (linha 181). Depois que a XML foi convertida de volta em uma `Equation`, as linhas 183–184 recuperam a resposta correta e exibem o lado esquerdo da equação. O botão **Check Answer** é então ativado (linha 185), e o usuário deve resolver o problema e inserir a resposta.

Quando o usuário insere um valor e clica em **Check Answer**, o handler de evento `checkAnswer JButtonActionPerformed` (linhas 144–168) recupera a resposta do usuário que foi armazenada anteriormente (linha 155). Se elas corresponderem, as linhas 157–161 reinicializam os elementos GUI para que o usuário possa gerar outra equação e dizer ao usuário que a resposta estava correta. Se não corresponderem, uma caixa de mensagem solicita ao usuário a tentar novamente é exibida (linhas 165–166).

31.11.3 Criando o serviço Web JSON baseado em REST EquationGenerator

Como vimos na Seção 31.8, serviços Web RESTful também podem retornar dados formatados como JSON. A Figura 31.25 é uma reimplementação do serviço `EquationGeneratorXML` que retorna uma `Equation` no formato JSON.

```
1  // Figura 31.25: EquationGeneratorJSONResource.java
2  // O gerador de equação RESTful que retorna JSON.
3  package com.deitel.equationgeneratorjson;
4
5  import com.google.gson.Gson; // converte POJO em JSON e volta novamente
6  import java.util.Random;
7  import javax.ws.rs.GET;
8  import javax.ws.rs.Path;
9  import javax.ws.rs.PathParam;
10 import javax.ws.rs.Produces;
11
12 @Path( "equation" )
13 public class EquationGeneratorJSONResource
14 {
15    static Random randomObject = new Random(); // gerador de número aleatório
16
17    // recupera uma equação formatada como JSON
18    @GET
19    @Path( "{operation}/{level}" )
20    @Produces( "application/json" )
21    public String getJson( @PathParam( "operation" ) String operation,
22       @PathParam( "level" ) int level )
23    {
24       // calcula os valores mínimos e máximos dos números
25       int minimum = ( int ) Math.pow( 10, level - 1 );
26       int maximum = ( int ) Math.pow( 10, level );
27
28       // cria os números no lado esquerdo da equação
29       int first = randomObject.nextInt( maximum - minimum ) + minimum;
30       int second = randomObject.nextInt( maximum - minimum ) + minimum;
31
32       // cria o objeto Equation e o resultado de retorno
33       Equation equation = new Equation( first, second, operation );
34       return new Gson().toJson( equation ); // converte em JSON e retorna
35    } // fim do método getJson
36 } // fim da classe EquationGeneratorJSONResource
```

Figura 31.25 | O gerador de equação RESTful que retorna JSON.

A lógica implementada aqui é a mesma que a da versão XML, exceto a última linha (linha 34), que utiliza o Gson para converter o objeto `Equation` no JSON em vez de utilizar o JAXB para convertê-lo em XML. Observe que a anotação `@Produces` (linha 20) também mudou para refletir o formato de dados JSON.

31.11.4 Consumindo o serviço Web JSON baseado em REST EquationGenerator

O programa na Figura 31.26 consome o serviço `EquationGeneratorJSON` e executa a mesma função que `EquationGeneratorXMLClient` — a única diferença é a maneira como o objeto `Equation` é recuperado do serviço Web. As linhas 181–183 constroem o URL que é utilizado para invocar o serviço `EquationGeneratorJSON`. Como no exemplo `WelcomeRESTJSONClient`, utilizamos a classe `URL` e um `InputStreamReader` para invocar o serviço Web e ler a resposta (linhas 186–187). O JSON recuperado é desserializado utilizando o Gson (linha 191) e convertido de volta em um objeto `Equation`. Como anteriormente, utilizamos o método `getResult` (linha 194) do objeto desserializado para obter a resposta e o método `getLeftHandSide` (linha 195) para exibir o lado esquerdo da equação.

```
1  // Figura 31.26: EquationGeneratorJSONClientJFrame.java
2  // Programa de ensino de matemática que utiliza REST e JSON para gerar equações.
3  package com.deitel.equationgeneratorjsonclient;
4
5  import com.google.gson.Gson; // converte POJO em JSON e volta novamente
6  import java.io.InputStreamReader;
7  import java.net.URL;
8  import javax.swing.JOptionPane;
```

```
 9
10  public class EquationGeneratorJSONClientJFrame extends javax.swing.JFrame
11  {
12     private String operation = "add"; // operação feita pelo usuário é testada em
13     private int difficulty = 1; // 1, 2, ou 3 dígitos em cada número
14     private int answer; // resposta correta à pergunta
15
16     // construtor sem argumento
17     public EquationGeneratorJSONClientJFrame()
18     {
19        initComponents();
20     } // fim do construtor sem argumentos
21
22     // O método initComponents é autogerado pelo NetBeans e é chamado
23     // a partir do construtor para inicializar a GUI. Esse método não é mostrado
24     // aqui para economizar espaço. Abre EquationGeneratorJSONClientJFrame.java no
25     // pasta desse exemplo para examinar o código gerado completo.
26
147    // determina se o usuário respondeu corretamente
148    private void checkAnswerJButtonActionPerformed(
149       java.awt.event.ActionEvent evt)
150    {
151       if ( answerJTextField.getText().equals( "" ) )
152       {
153          JOptionPane.showMessageDialog(
154             this, "Please enter your answer." );
155       } // fim do if
156
157       int userAnswer = Integer.parseInt( answerJTextField.getText() );
158
159       if ( userAnswer == answer )
160       {
161          equationJLabel.setText( "" ); // limpa o rótulo
162          answerJTextField.setText( "" ); // limpa o campo de texto
163          checkAnswerJButton.setEnabled( false );
164          JOptionPane.showMessageDialog( this, "Correct! Good Job!",
165             "Correct", JOptionPane.PLAIN_MESSAGE );
166       } // fim do if
167       else
168       {
169          JOptionPane.showMessageDialog( this, "Incorrect. Try again.",
170             "Incorrect", JOptionPane.PLAIN_MESSAGE );
171       } // fim de else
172    } // fim do método checkAnswerJButtonActionPerformed
173
174    // recupera a equação do serviço Web e exibe o lado esquerdo ao usuário
175    private void generateJButtonActionPerformed(
176       java.awt.event.ActionEvent evt)
177    {
178       try
179       {
180          // URL do serviço EquationGeneratorJSON, com parâmetros
181          String url = String.format( "http://localhost:8080/" +
182             "EquationGeneratorJSON/resources/equation/%s/%d",
183             operation, difficulty );
184
185          // abre URL e cria um Reader para ler os dados
186          InputStreamReader reader =
187             new InputStreamReader( new URL( url ).openStream() );
188
189          // converte o JSON de volta em um objeto Equation
190          Equation equation =
191             new Gson().fromJson( reader, Equation.class );
192
193          // atualiza o estado interno e a GUI para refletir a equação
194          answer = equation.getResult();
195          equationJLabel.setText( equation.getLeftHandSide() + " =" );
196          checkAnswerJButton.setEnabled( true );
197       } // fim do try
```

```
198          catch ( Exception exception )
199          {
200             exception.printStackTrace();
201          } // fim do catch
202       } // fim do método generateJButtonActionPerformed
203
204       // obtém a operação matemática selecionada pelo usuário
205       private void operationJComboBoxItemStateChanged(
206          java.awt.event.ItemEvent evt)
207       {
208          String item = ( String ) operationJComboBox.getSelectedItem();
209
210          if ( item.equals( "Addition" ) )
211             operation = "add"; // o usuário selecionou adição
212          else if ( item.equals( "Subtraction" ) )
213             operation = "subtract"; // o usuário selecionou subtração
214          else
215             operation = "multiply"; // o usuário selecionou multiplicação
216       } // fim do método operationJComboBoxItemStateChanged
217
218       // obtém o nível de dificuldade selecionado pelo usuário
219       private void levelJComboBoxItemStateChanged(
220          java.awt.event.ItemEvent evt)
221       {
222          // índice inicia em 0, assim adiciona 1 para obter o nível de dificuldade
223          difficulty = levelJComboBox.getSelectedIndex() + 1;
224       } // fim do método levelJComboBoxItemStateChanged
225
226       // método main inicia a execução
227       public static void main( String args[] )
228       {
229          java.awt.EventQueue.invokeLater(
230             new Runnable()
231             {
232                public void run()
233                {
234                   new EquationGeneratorJSONClientJFrame().setVisible( true );
235                } // fim do método run
236             } // fim da classe interna anônima
237          ); // fim da chamada a java.awt. EventQueue.invokeLater
238       } // fim de main
239
240       // Declaração de variáveis - não modifique
241       private javax.swing.JLabel answerJLabel;
242       private javax.swing.JTextField answerJTextField;
243       private javax.swing.JButton checkAnswerJButton;
244       private javax.swing.JLabel equationJLabel;
245       private javax.swing.JButton generateJButton;
246       private javax.swing.JComboBox levelJComboBox;
247       private javax.swing.JLabel levelJLabel;
248       private javax.swing.JComboBox operationJComboBox;
249       private javax.swing.JLabel operationJLabel;
250       private javax.swing.JLabel questionJLabel;
251       // Fim da declaração de variáveis
252    } // fim da classe EquationGeneratorJSONClientJFrame
```

Figura 31.26 | Programa de ensino de matemática que utiliza REST e JSON para gerar equações.

31.12 Conclusão

Este capítulo introduziu serviços da Web — um conjunto de tecnologias para construir sistemas distribuídos nos quais os componentes de sistema se comunicam entre si por redes. Em particular, apresentamos serviços Web baseados em JAX-WS SOAP e serviços Web baseados em JAX-RS REST. Você aprendeu que um serviço Web é uma classe que permite ao software cliente chamar os métodos do serviço Web remotamente via protocolos e formatos de dados comuns como XML, JSON, HTTP, SOAP e REST. Também discutimos vários benefícios da computação distribuída com serviços Web.

Explicamos como o NetBeans e o JAX-WS e JAX-RS APIs facilitam publicar e consumir serviços Web. Você aprendeu a definir métodos e serviços Web utilizando o protocolo SOAP e a arquitetura REST e a retornar dados nos formatos XML e JSON. Você consumiu serviços Web baseados em SOAP utilizando classes proxy para chamar os métodos do serviço Web. Você também consumiu serviços Web baseados em REST utilizando a classe `URL` para invocar os serviços e abrir InputStreams a partir dos quais os clientes podem ler as respostas dos serviços. Você aprendeu a definir serviços Web e métodos Web e também a consumir ambos a partir de aplicativos desktop Java e aplicativos Web. Depois de explicar os mecânicos dos serviços Web por meio dos nossos exemplos `Welcome`, demonstramos serviços Web mais sofisticados que utilizam rastreamento de sessão, acesso a banco de dados e tipos definidos pelo usuário. Também explicamos a serialização XML e JSON e mostramos como recuperar objetos dos tipos definidos pelo usuário a partir de serviços Web.

Resumo

Seção 31.1 Introdução

- Um serviço Web é um componente de software armazenado em um computador que pode ser acessado por um aplicativo (ou outro componente de software) em outro computador por uma rede.
- Serviços Web se comunicam utilizando tecnologias como XML, JSON e HTTP.
- O JAX-WS é baseado no Simple Object Access Protocol (SOAP) — um protocolo baseado em XML que permite a serviços Web e clientes se comunicarem.
- O JAX-RS utiliza o Representational State Transfer (REST) — uma arquitetura de rede que utiliza os mecanismos de solicitação/resposta tradicionais da Web como solicitações `GET` e `POST`.
- Serviços da Web permitem aos negócios conduzir transações via serviços Web padronizados amplamente disponíveis em vez de contar com aplicativos proprietários.
- Serviços Web são independentes de plataforma e linguagem, portanto, as empresas podem colaborar via serviços Web sem problemas de compatibilidade de hardware, software e comunicação.
- O NetBeans é uma entre várias ferramentas que permitem "publicar" e/ou "consumir" serviços Web.

Seção 31.2 Fundamentos do serviço Web

- A máquina na qual um serviço Web reside é chamada host de serviço Web.
- Um aplicativo cliente que acessa o serviço Web envia uma chamada de método por uma rede ao host de serviço Web, que processa a chamada e retorna uma resposta pela rede ao aplicativo.
- No Java, um serviço Web é implementado como uma classe. A classe que representa o serviço Web reside em um servidor — ela não é parte do aplicativo cliente.
- Disponibilizar um serviço Web para receber solicitações de cliente é conhecido como publicar um serviço Web; utilizar um serviço Web a partir de um aplicativo cliente é conhecida como consumir um serviço Web.

Seção 31.3 Simple Object Access Protocol (SOAP)

- O SOAP é um protocolo independente de plataforma que utiliza XML para fazer chamadas de procedimento remoto, geralmente via HTTP. Cada solicitação e resposta são empacotadas em uma mensagem SOAP — uma mensagem XML que contém as informações que um serviço Web exige para processar a mensagem.
- Mensagens SOAP são escritas em XML para que sejam legíveis por computadores e humanos e para que sejam independentes de plataforma.
- O SOAP suporta um extenso conjunto de tipos — tipos primitivos, bem como `DateTime`, `XmlNode` e outros. O SOAP também pode transmitir arrays desses tipos.
- Quando um programa invoca um método de um serviço Web SOAP, a solicitação e todas as informações relevantes são empacotadas em uma mensagem SOAP, colocadas em um envelope SOAP e enviadas ao servidor no qual o serviço Web reside.
- Quando um serviço Web recebe uma mensagem SOAP, ele analisa a XML que representa a mensagem e então processa o conteúdo da mensagem. A mensagem especifica o método que o cliente deseja executar e os argumentos que o cliente passou para esse método.
- Depois que um serviço Web analisa uma mensagem SOAP, ele chama o método apropriado com os argumentos especificados (se houver algum) e reenvia a resposta ao cliente em outra mensagem SOAP. O cliente analisa a resposta para recuperar o resultado do método.

Seção 31.4 REpresentational State Transfer (REST)

- O REpresentational State Transfer (REST) refere-se a um estilo arquitetônico de implementar serviços Web. Esses serviços Web costumam ser chamados de serviços Web RESTful. Embora o próprio REST não seja um padrão, serviços Web RESTful são implementados utilizando padrões Web.
- Cada operação em um serviço Web RESTful é identificada por um URL exclusivo.
- O REST pode retornar dados em muitos formatos, incluindo XML e JSON.

Seção 31.5 JavaScript Object Notation (JSON)

- A JavaScript Object Notation (JSON) é uma alternativa à XML para representar dados.
- O JSON é um formato baseado em texto de troca de dados utilizado para representar objetos em JavaScript como coleções de pares de nome/valor representados como `Strings`.
- O JSON é um formato simples que torna objetos fáceis de ler, criar e analisar e permite a programas transmitir dados eficientemente pela Internet porque ele é bem menos prolixo do que a XML.
- Cada valor em um array JSON pode ser uma string, um número, um objeto JSON, `true`, `false` ou `null`.

Seção 31.6.1 Criando de um projeto de aplicativo Web e adicionando uma classe de serviço Web no NetBeans

- Ao criar um serviço Web no NetBeans, você focaliza a lógica do serviço Web e deixa o IDE tratar a infraestrutura do serviço Web.
- Para criar um serviço Web no NetBeans, você primeiro cria um projeto Web Application.

Seção 31.6.2 Definindo o serviço Web `WelcomeSOAP` no NetBeans

- Por padrão, cada nova classe de serviço Web criada com as APIs do JAX-WS é um POJO (Plain Old Java Object) — você não precisa estender uma classe ou implementar uma interface para criar um serviço Web.
- Ao implantar um aplicativo Web que contém um serviço Web JAX-WS, o servidor cria os artefatos no lado do servidor que suportam o serviço Web.
- A anotação `@WebService` indica que uma classe representa um serviço Web. O elemento `name` opcional especifica o nome da classe de interface de ponto de extremidade de serviço (*Service Endpoint Interface* — SEI). O elemento `serviceName` opcional especifica o nome da classe que o cliente utiliza para obter um objeto SEI.
- O NetBeans insere a anotação `@WebService` no início de cada nova classe de serviço Web que você cria. Você pode adicionar os elementos `name` `serviceName` opcionais nos parênteses da anotação.
- Os métodos que são marcados com a anotação `@WebMethod` podem ser chamados remotamente.
- Os métodos que não são marcados com `@WebMethod` não são acessíveis a clientes que consomem o serviço Web. Esses métodos costumam ser métodos utilitários dentro da classe de serviço Web.
- O elemento `operationName` opcional da anotação `@WebMethod` especifica o nome do método que é exposto aos clientes do serviço Web.
- Parâmetros de métodos Web são anotados com a anotação `@WebParam`. O elemento opcional `name` indica o nome de parâmetro que é exposto aos clientes do serviço Web.

Seção 31.6.3 Publicando o serviço Web `WelcomeSOAP` a partir do NetBeans

- O NetBeans trata todos os detalhes da criação e implantação de um serviço Web para você. Isso inclui criar o framework necessário para suportar o serviço Web.
- Para determinar se há algum erro de compilação no seu projeto, clique com o botão direito do mouse no nome do projeto na guia NetBeans Projects e selecione a opção Build.
- Quando o projeto compila com sucesso, você pode selecionar Deploy para implantá-lo no servidor que você selecionou durante a configuração do aplicativo.
- Se o código no projeto mudou desde a última construção, selecionar Deploy também constrói o projeto.
- Selecionar Run executa o aplicativo Web.
- As opções Deploy e Run também constroem o projeto se ele mudou e iniciam o servidor de aplicativos se ele ainda não estiver em execução.
- Para assegurar que todos os arquivos de código-fonte em um projeto são recompilados durante a próxima operação de construção, utilize as opções Clean ou Clean and Build.

Seção 31.6.4 Testando o serviço Web `WelcomeSOAP` com a página Web `Tester` do servidor de aplicativos GlassFish

- GlassFish pode criar dinamicamente uma página Web para testar métodos de um serviço Web em um navegador Web. Para abrir a página de teste, expanda o nó Web Services do projeto na guia NetBeans Projects, então clique com o botão direito do mouse no nome da classe do serviço Web e escolha Test Web Service.
- Um cliente só pode acessar um serviço Web quando o servidor de aplicativos está em execução. Se o NetBeans inicializar o servidor de aplicativos para você, o servidor será desativado quando você fecha o NetBeans. Para manter o servidor de aplicativos pronto e funcionando, você pode inicializá-lo independentemente do NetBeans.

Seção 31.6.5 Descrevendo um serviço Web com a Web Service Description Language (WSDL)

- Para consumir um serviço Web, um cliente deve saber onde localizá-lo e deve ter a descrição do serviço Web.
- JAX-WS utiliza a Web Service Description Language Web (WSDL) — um vocabulário XML padrão para descrever serviços a Web de uma maneira independente de plataforma.

- O servidor gera a WSDL de um serviço Web dinamicamente para você, e as ferramentas cliente podem analisar a WSDL para ajudar a criar a classe proxy no lado do cliente que um cliente utiliza para acessar o serviço Web.
- Para visualizar a WSDL para um serviço Web, digite o URL no campo de endereço do navegador seguido por `?WSDL` ou clique no link **WSDL File** no página Web `Tester` do serviço.

Seção 31.6.6 Criando um cliente para consumir o serviço Web `WelcomeSOAP`

- Um cliente de serviço Web pode ser qualquer tipo do aplicativo ou até mesmo outro serviço Web.
- Você ativa um aplicativo cliente baseado em Java para consumir um serviço Web adicionando uma referência de serviço Web, que define a classe SEI para o cliente poder acessar o serviço Web.
- Um aplicativo que consome um serviço Web baseado em SOAP invoca métodos em um objeto SEI que interagem com o serviço Web no nome do cliente.
- O objeto SEI trata os detalhes da passagem dos argumentos de método e recebimento dos valores de retorno do serviço Web. Essa comunicação pode ocorrer por uma rede local, pela Internet ou até mesmo com um serviço Web no mesmo computador.
- O NetBeans cria essas classes SEI para você.
- Ao adicionar a referência de serviço Web, o IDE cria e compila os artefatos no lado do cliente — o framework do código Java que suporta a classe SEI do lado do cliente. A classe SEI utiliza o resto dos artefatos para interagir com o serviço Web.
- Uma referência de serviço Web é adicionada fornecendo ao NetBeans o URL do arquivo WSDL do serviço Web.

Seção 31.6.7 Consumindo o serviço Web `WelcomeSOAP`

- Ao criar uma GUI no NetBeans, ele utiliza nomes de classe totalmente qualificados; portanto, declarações `import` são desnecessárias.
- Para consumir um serviço Web JAX-WS, você deve obter um objeto SEI chamando o método `getWelcomeSOAPPort` na classe que foi especificada com o elemento `serviceName` da anotação `@WebService`. Esse método retorna um objeto SEI da classe que foi especificada com o elemento `name` da anotação `@WebService`. Você então invoca os métodos de serviços por meio do objeto SEI.

Seção 31.7.1 Criando um serviço Web XML baseado em REST

- O plug-in **RESTful Web Services** para o NetBeans fornece várias templates para criar serviços Web RESTful, incluindo aquelas que podem interagir com bancos de dados no nome do cliente.
- A anotação `@Path` em uma classe de serviço Web JAX-RS indica o URI para acessar o serviço Web. Ela é anexada ao URL do projeto de aplicativo Web para invocar o serviço. Os métodos da classe também podem utilizar a anotação `@Path`.
- Partes do caminho especificado entre colchetes indicam parâmetros — eles são marcadores de lugar para argumentos que são passados para o serviço Web como parte do caminho. O caminho de base do serviço é o diretório `resources` do projeto.
- Os argumentos em um URL podem ser utilizados como argumentos para um método de serviço Web. Para fazer isso, vincule os parâmetros descritos na especificação `@Path` para os parâmetros de um método de serviço Web com a anotação `@PathParam`. Quando a solicitação é recebida, o servidor passa o(s) argumento(s) no URL para o(s) parâmetro(s) apropriado(s) no método de serviço Web.
- A anotação `@GET` denota que um método é acessado via solicitação `GET` HTTP. Anotações semelhantes existem para as solicitações `PUT`, `POST`, `DELETE` e `HEAD` HTTP.
- A anotação `@Produces` denota o tipo de conteúdo retornado ao cliente. É possível ter múltiplos métodos com o mesmo caminho e método HTTP, mas diferentes anotações `@Produces`, e o JAX-RS chamará o método correspondente ao tipo de conteúdo solicitado pelo cliente.
- A anotação `@Consumes` restringe o tipo de conteúdo que um serviço Web aceita de uma solicitação `PUT`.
- JAXB (Java Architecture for XML Binding) é um conjunto de classes para converter POJOs em e da XML. A classe `JAXB` (pacote `javax.xml.bind`) contém métodos `static` para operações comuns.
- O método `static marshal` da classe `JAXB` converte um objeto Java no formato XML.
- O GlassFish não fornece páginas de teste para serviços RESTful, mas o NetBeans gera uma página de teste que pode ser acessada clicando com o botão direito do mouse no nó do projeto na guia **Projects** e selecionando **Test RESTful Web Services**.
- Na página de teste, selecione um elemento de método na coluna esquerda. O lado direito da página exibe um formulário que permite escolher o tipo MIME dos dados e deixa inserir os argumentos do método. Clique no botão **Test** para invocar o serviço Web e exibir os dados retornados.
- WADL (Web Application Description Language) tem objetivos de design semelhantes à WSDL, mas descreve serviços RESTful em vez de serviços SOAP.

Seção 31.7.2 Consumindo um serviço Web XML baseado em REST

- Clientes de serviços Web RESTful não requerem referências de serviço Web.
- A classe `JAXB` tem um método `static unmarshal` que recebe como argumentos um nome de arquivo ou URL como uma `String`, e um objeto `Class<T>` que indica a classe Java na qual a XML será convertida.

Seção 31.8 Publicando e consumindo serviços Web JSON baseados em REST

- Componentes JSON — objetos, arrays, strings, números — podem ser facilmente mapeados para construções no Java e em outras linguagens de programação.
- Há muitas bibliotecas JSON código-fonte aberto para o Java e outras linguagens. A biblioteca Gson em `code.google.com/p/google-gson/` fornece um modo simples de converter POJOs em e do JSON.

Seção 31.8.1 Criando um serviço Web JSON baseado em REST

- Para adicionar um arquivo JAR como uma biblioteca no NetBeans, clique com o botão direito do mouse na pasta **Libraries** do seu projeto, escolha **Add JAR/Folder...**, localize o arquivo JAR e clique em **Open**.
- Para um método de serviço Web que retorna texto JSON, o argumento para o atributo `@Produces` deve ser `"application/json"`.
- No JSON, todos os dados devem ser encapsulados em um tipo de dados composto.
- Crie um objeto `Gson` (do pacote `com.google.gson`) e chame o método `toJson` para converter um objeto na sua representação `String` JSON.

Seção 31.8.2 Consumindo um serviço Web JSON baseado em REST

- Para ler dados JSON de um URL, crie um objeto `URL` e chame o método `openStream`. Isso invoca o serviço Web e retorna um `InputStream` do qual o cliente pode ler a resposta. Empacote o `InputStream` em um `InputStreamReader` para que ele possa ser passado como o primeiro argumento para o método `fromJson` da classe `Gson`.

Seção 31.9 Rastreamento de sessão em um serviço Web baseado em SOAP

- Pode ser vantajoso que um serviço Web mantenha informações de estado de cliente, eliminando assim a necessidade de passar informações cliente entre o cliente e o serviço Web múltiplas vezes. O armazenamento de informações de sessão também permite que um serviço Web diferencie seus clientes.

Seção 31.9.1 Criando um serviço Web `Blackjack`

- Para utilizar o rastreamento de sessão em um serviço Web, você deve incluir o código dos recursos que mantêm as informações de estado de sessão. JAX-WS trata isso para você via anotação `@Resource`. Essa anotação "injeta" o código de suporte complexo na sua classe, permitindo assim que você se concentre na lógica do negócio em vez do código de suporte.
- Utilizar anotações para adicionar código que suporta suas classes é conhecido como injeção de dependência. Anotações como `@WebService`, `@WebMethod` e `@WebParam` também executam a injeção de dependência.
- Um objeto `WebServiceContext` ativa um serviço Web para acessar e manter informações para uma solicitação específica. O código que cria um objeto `WebServiceContext` é injetado na classe por uma anotação `@Resource`.
- O objeto `WebServiceContext` é utilizado para obter um objeto `MessageContext`. Um serviço Web utiliza um `MessageContext` para obter um objeto `HttpSession` para o cliente atual.
- O método `get` do objeto `MessageContext` obtém o objeto `HttpSession` para o cliente atual. O método `get` recebe uma constante indicando o que obter do `MessageContext`. A constante `MessageContext.SERVLET_REQUEST` indica que queremos obter o objeto `HttpServletRequest` para o cliente atual. Chamamos então o método `getSession` para obter o objeto `HttpSession` a partir do objeto `HttpServletRequest`.
- O método `HttpSession` `getAttribute` recebe uma `String` que identifica o `Object` a obter do estado de sessão.

Seção 31.9.2 Consumindo o serviço Web `Blackjack`

- No framework JAX-WS, o cliente deve indicar se ele quer permitir que o serviço Web mantenha informações de sessão. Para fazer isso, primeiro faça uma coerção do objeto proxy no tipo de interface `BindingProvider`. Um `BindingProvider` permite que o cliente manipule as informações de solicitação que serão enviadas ao servidor. Essas informações são armazenadas em um objeto que implementa a interface `RequestContext`. `BindingProvider` e `RequestContext` são parte do framework que é criado pelo IDE quando você adiciona um cliente de serviço Web ao aplicativo.
- Em seguida, invoque o método `getRequestContext` de `BindingProvider` para obter o objeto `RequestContext`. Chame então o método `put` de `RequestContext` para configurar a propriedade `BindingProvider.SESSION_MAINTAIN_PROPERTY` como `true`, o que ativa o rastreamento de sessão no lado de cliente para que o serviço Web saiba que o cliente está invocando os métodos Web do serviço.

Seção 31.11 Gerador de equação: retornando tipos definidos pelo usuário

- Também é possível processar instâncias dos tipos de classe em um serviço Web. Esses tipos podem ser passados a ou retornados de métodos de serviço Web.
- Uma variável de instância só pode ser serializada se ela for `public` ou tiver um método *get* e um *set*.
- As propriedades que podem ser geradas dos valores de outras propriedades não devem ser serializadas para evitar redundância.
- O JAX-RS converte automaticamente argumentos de uma anotação `@Path` no tipo correto, e retornará um erro "not found" ao cliente se o argumento não puder ser convertido da `String` passada como parte do URL no tipo de destino. Os tipos suportados para conversão incluem tipos inteiros, tipos de ponto flutuante, `boolean` e as classes empacotadoras de tipo correspondentes.

Terminologia

adicionando uma referência de serviço Web a um aplicativo, 1024
artefatos do lado do cliente, 1025
artefatos do lado do servidor, 1021
`BindingProvider`, interface, 1045
business-to-business (B2B), transações, 1019
classe proxy de um serviço Web, 1022
`@Consumes`, anotação, 1029
consumir um serviço Web, 1019
descrição de um serviço Web, 1024
firewall, 1019
`fromJson`, método da classe Gson, 1034
`@GET`, anotação, 1029
`getAttribute`, método da interface `HttpSession`, 1037
`getRequestContext`, método da interface `BindingProvider`, 1045
`getSession`, método da interface `HttpSession`, 1037
host de serviço Web, 1019
injeção de dependência, 1037
interface de ponto de extremidade de serviço (SEI), 1022
`JAXB`, classe, 1029
JAXB (Java Architecture for XML Binding), 1029
JAX-RS, 1018
JAX-WS, 1018
`Json`, método da classe `Gson`, 1034
JSON (JavaScript Object Notation), 1020
`marshal`, método da classe `JAXB`, 1030
`MessageContext`, interface, 1037
`name`, atributo da anotação `@WebParam`, 1022
`name`, atributo da anotação `@WebService`, 1022
`openStream`, método da classe `URL`, 1034
`operationName`, atributo da anotação `@WebMethod`, 1022
`@Path`, anotação, 1029
`@PathParam`, anotação, 1029
POJO (Plain Old Java Object), 1021
`@Produces`, anotação, 1029
publicar um serviço Web, 1019, 1060
`put`, método da interface `RequestContext`, 1045
`RequestContext`, interface, 1045
`@Resource`, anotação, 1037
RESTful, serviços Web, 1020
REST (Representational State Transfer), 1018
SEI (*Service Endpoint Interface*), 1022
`serviceName`, atributo da anotação `@WebService`, 1022
serviço Web, 1018
Serviço Web de Neutralidade Sexual, 1065
`setAttribute`, método da interface `HttpSession`, 1037
SOAP, formato wire, 1019
SOAP, mensagem, 1019
SOAP (Simple Object Access Protocol), 1018
`StringWriter`, classe, 1030
`toJson`, método da classe `Gson`, 1033
`unmarshal`, método da classe `JAXB`, 1031
WADL (Web Application Description Language), 1030
`@WebMethod`, anotação, 1022
Web Application, projeto, 1020
`@WebParam`, anotação, 1022
`@WebService`, anotação, 1021
`WebServiceContext`, interface, 1037
WSDL (Web Service Description Language), 1024

Exercícios de autorrevisão

31.1 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Todos os métodos de uma classe de serviço Web podem ser invocados pelos clientes desse serviço Web.

b) Ao consumir um serviço Web em um aplicativo cliente criado no NetBeans, você deve criar a classe proxy que permite ao cliente se comunicar com o serviço Web.

c) Uma classe proxy que se comunica com um serviço Web normalmente utiliza o SOAP para enviar e receber mensagens.

d) O rastreamento de sessão é automaticamente ativado em um cliente de um serviço Web.

e) Métodos Web não podem ser declarados `static`.

f) Um tipo definido pelo usuário utilizado em um serviço Web deve definir os métodos *get* e *set* para qualquer propriedade que será serializada.

g) Operações em um serviço Web REST são definidas pelos seus próprios URL únicos.

h) Um serviço Web baseado em SOAP pode retornar dados no formato JSON.

31.2 Preencha as lacunas de cada uma das seguintes afirmações:

a) Uma diferença chave entre o SOAP e o REST é que os dados das mensagens SOAP são empacotados em um(a) _____.

b) Um serviço Web no Java é um(a) _____ — ele não precisa implementar uma interface ou estender classes.

c) Solicitações de serviço Web são geralmente transportadas pela Internet via protocolo _____.

d) Para definir o nome exposto de um método Web, utilize o elemento _____ da anotação `@WebMethod`.

e) _____ transforma um objeto em um formato que pode ser enviado entre um serviço Web e um cliente.

f) Para retornar dados no formato JSON a partir de um método de um serviço Web baseado em REST, a anotação `@Produces` é configurada como _____.

g) Para retornar dados no formato XML a partir de um método de um serviço Web baseado em REST a anotação `@Produces` é configurada como _____.

Respostas dos exercícios de autorrevisão

31.1 a) Falso. Somente os métodos declarados com a anotação `@WebMethod` podem ser invocados pelos clientes de um serviço Web. b) Falso. A classe proxy é criada pelo NetBeans quando você adiciona um cliente de serviço Web ao aplicativo. c) Verdadeiro. d) Falso. No framework JAX-WS, o cliente deve indicar se ele quer permitir que o serviço Web mantenha informações de sessão. Primeiro, você deve fazer coerção do objeto proxy no tipo de interface `BindingProvider` e então utilizar o método `getRequestContext` de `BindingProvider` para obter o objeto `RequestContext`. Por fim, você deve utilizar o método `put` de `RequestContext` para configurar a propriedade `BindingProvider.SESSION_MAINTAIN_PROPERTY` como `true`. e) Verdadeiro. f) Verdadeiro. g) Verdadeiro. h) Falso. Um serviço Web SOAP retorna implicitamente os dados no formato XML.

31.2 a) Mensagem SOAP ou envelope SOAP. b) POJO (Plain Old Java Object) c) HTTP. d) `operationName`. serialização e). f) `"application/json"`. g) `"application/xml"`.

Exercícios

31.3 (***Serviço Web de lista telefônica***) Crie um serviço Web RESTful que armazena entradas de lista telefônica no banco de dados `PhoneBookDB` e um aplicativo cliente Web que consome esse serviço. O serviço Web deve gerar XML. Utilize os passos na Seção 30.2.1 para criar o banco de dados `PhoneBook` utilizando o script `PhoneBookDB.sql` fornecido na pasta de exemplos. O banco de dados contém uma tabela — `PhoneBook` — com três colunas — `LastName`, `FirstName` e `PhoneNumber`. As colunas `LastName` armazenam até 30 caracteres. A coluna `PhoneNumber` suporta números de telefone da forma `(800) 555-1212` que contêm 14 caracteres.

Dê ao usuário cliente a capacidade de inserir um novo contato (método Web `addEntry`) e localizar contatos pelo sobrenome (método Web `getEntries`). Passe somente `String`s como argumentos para o serviço Web. O método Web `getEntries` deve retornar um array `String`s que contém as entradas de lista telefônica correspondentes. Cada `String` no array deve consistir no sobrenome, nome e número de telefone para uma entrada de lista telefônica. Esses valores devem ser separados por vírgulas.

A consulta `SELECT` que localizará uma entrada `PhoneBook` pelo sobrenome deve ser:

```
SELECT LastName, FirstName, PhoneNumber
FROM PhoneBook
WHERE (LastName = Sobrenome)
```

A instrução `INSERT` que insere uma nova entrada no banco de dados `PhoneBook` deve ser:

```
INSERT INTO PhoneBook (LastName, FirstName, PhoneNumber)
VALUES (Sobrenome, Nome, NúmeroDoTelefone)
```

31.4 (***Modificação do serviço Web de lista telefônica***) Modifique o Exercício 31.3 para que ele utilize uma classe nomeada `PhoneBookEntry` a fim de representar uma linha no banco de dados. O serviço Web deve retornar os objetos do tipo `PhoneBookEntry` no formato XML para o método `getEntries`, e o aplicativo cliente deve utilizar o método `JAXB unmarshal` para recuperar objetos `PhoneBookEntry`.

31.5 (***Serviço Web de lista telefônica com JSON***) Modifique o Exercício 31.4 para que a classe `PhoneBookEntry` seja passada para e do serviço Web como um objeto JSON. Utilize a serialização para converter o objeto JSON em um objeto do tipo `PhoneBookEntry`.

31.6 (***Modificação do serviço Web Blackjack***) Modifique o exemplo do serviço Web `Blackjack` na Seção 31.9 para incluir a classe `Card`. Modifique o método Web `dealCard` para que ele retorne um objeto do tipo `Card` e modifique o método Web `getHandValue` para que ele receba um array de objetos `Card` do cliente. Também modifique o aplicativo cliente para monitorar as cartas distribuídas utilizando `ArrayList`s dos objetos `Card`. A classe proxy criada pelo NetBeans tratará o parâmetro de array de um método Web como uma `List`, assim você pode passar essas `ArrayList`s dos objetos `Card` diretamente para o método `getHandValue`. Sua classe `Card` deve incluir os métodos *get* e *set* para a face e o naipe da carta.

31.7 (***Modificação do serviço Web de reservas de passagens aéreas***) Modifique o serviço Web de reservas de passagens aéreas na Seção 31.10 para que ele contenha dois métodos separados — um que permite aos usuários visualizar todas as poltronas disponíveis, e o outro que permite aos usuários fazer a reserva de uma determinada poltrona atualmente disponível. Utilize um objeto do tipo `Ticket` para passar informações para e a partir do serviço Web. O serviço Web deve ser capaz de tratar casos em que dois usuários visualizam as poltronas disponíveis, um faz a reserva de uma poltrona e o segundo usuário tenta fazer a reserva da mesma poltrona, sem saber que ela já foi reservada. Os nomes dos métodos que executam devem ser `reserve` e `getAllAvailableSeats`.

31.8 (***Serviço Web de código Morse***) No Exercício 16.22, você aprendeu o código Morse e escreveu aplicativos que podem traduzir frases em inglês para o código Morse e vice-versa. Crie um serviço Web baseado em SOAP que forneça dois métodos — um que traduz uma frase inglesa ao Código Morse e outro que traduz o código Morse para o inglês. Em seguida, construa um aplicativo GUI tradutor do código Morse que invoca o serviço Web para executar essas traduções.

Fazendo a diferença

31.9 (***Serviço Web de scanner de spam***) No Exercício 16.27, você criou um aplicativo scanner de spam que varria um e-mail e atribuía uma pontuação com base na ocorrência de certas palavras e frases que comumente aparecem em e-mails de spam e quantas vezes as palavras e frases ocorreram no e-mail. Crie um serviço Web de scanner de spam baseado em SOAP. Em seguida, modifique o aplicativo GUI que você criou no Exercício 16.27 para utilizar o serviço Web a fim de analisar um e-mail. Exiba então a avaliação em pontos retornada pelo serviço Web.

31.10 (***Serviço Web de SMS***) No Exercício 16.28, você criou um aplicativo tradutor de mensagens SMS. Crie um serviço Web baseado em SOAP com três métodos:

a) um que recebe uma abreviação SMS e retorna a palavra inglesa ou frase correspondente,

b) um que recebe uma mensagem SMS inteira e retorna o texto inglês correspondente, e

c) um que traduz o texto inglês em uma mensagem SMS.

Utilize o serviço Web a partir de um aplicativo GUI que exibe as respostas do serviço Web.

31.11 (***Serviço Web de neutralidade sexual***) No Exercício 1.14, você pesquisou como eliminar o machismo em todas as formas de comunicação. Você então descreveu o algoritmo que utilizaria para ler todo um parágrafo de texto e substituir palavras específicas por equivalentes neutras ao sexo. Crie um serviço Web baseado em SOAP que recebe um parágrafo de texto e então substitui palavras específicas ao sexo da pessoa por palavras neutras. Utilize o serviço Web a partir de um aplicativo GUI que exibe o texto neutro resultante.

A Tabela de precedência de operadores

Os operadores são mostrados em ordem de precedência decrescente de cima para baixo (Figura A.1).

Operador	Descrição	Associatividade
`++` `--`	incremento pós-fixo unário decremento pós-fixo unário	da direita para a esquerda

`++`
`--`
`+`
`-`
`!`
`~`
`( tipo )`
	pré-incremento unário
	pré-decremento unário
	mais unário
	menos unário
	negação unária lógica
	complemento unário sobre bits
	coerção unária
		da direita para a esquerda

`*`
`/`
`%`
	multiplicação
	divisão
	módulo
		da esquerda para a direita

`+` `-`	adição ou concatenação de strings subtração	da esquerda para a direita

`<<`
`>>`
`>>>`
	deslocamento de bits para a esquerda
	deslocamento para a direita com sinal
	deslocamento para a direita sem sinal
		da esquerda para a direita

`<`
`<=`
`>`
`>=`
`instanceof`
	menor que
	menor que ou igual a
	o maior que
	maior que ou igual a
	comparação de tipo
		da esquerda para a direita

`==` `!=`	é igual a não é igual a	da esquerda para a direita
`&`	E sobre bits E lógico booleano	da esquerda para a direita
`^`	OU exclusivo sobre bits OU lógico booleano exclusivo	da esquerda para a direita

Operador	Descrição	Associatividade
|	OU inclusivo sobre bits OU inclusivo lógico booleano	da esquerda para a direita
&&	E condicional	da esquerda para a direita
||	OU condicional	da esquerda para a direita
?:	condicional	da direita para a esquerda
= += -= *= /= %= &= ^= |= <<= >>= >>>=	atribuição atribuição de adição atribuição de subtração atribuição de multiplicação atribuição de divisão atribuição de resto atribuição E sobre bits atribuição de OU exclusivo sobre bits atribuição de OU inclusivo sobre bits atribuição de deslocamento para a esquerda sobre bits atribuição de bits com deslocamento para a direita com sinal	da direita para a esquerda

Figura A.1 | Tabela de precedência de operadores.

B Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Figura B.1 | Conjunto de caracteres ASCII.

Os dígitos à esquerda da tabela são os dígitos à esquerda dos equivalentes decimais (0–127) dos códigos de caractere e os dígitos na parte superior da tabela são os dígitos à direita dos códigos de caractere. Por exemplo, o código de caractere para “F” é 70 e o código de caractere para “&” é 38. Portanto, cruzando linhas com colunas, você obtém os códigos de caractere para cada caractere na tabela.

A maioria dos usuários deste livro está interessada no conjunto de caracteres ASCII utilizado para representar caracteres da língua inglesa em muitos computadores. O ASCII é um subconjunto do conjunto de caracteres Unicode utilizado pelo Java para representar caracteres da maioria dos idiomas do mundo. Para mais informações sobre o conjunto de caracteres Unicode, consulte o Apêndice L.

Palavras-chave e palavras reservadas

C

Palavras-chave do Java				
`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`continue`
`default`	`do`	`double`	`else`	`enum`
`extends`	`final`	`finally`	`float`	`for`
`if`	`implements`	`import`	`instanceof`	`int`
`interface`	`long`	`native`	`new`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`strictfp`	`super`	`switch`	`synchronized`
`this`	`throw`	`throws`	`transient`	`try`
`void`	`volatile`	`while`		
Palavras-chave que não são atualmente utilizadas:				
`const`	`goto`			

Figura C.1 | Palavras-chave do Java.

O Java também contém as palavras reservadas `true` e `false`, que são literais `boolean` e `null`, que é o literal que representa uma referência a nada. Assim como as palavras-chave, essas palavras reservadas não podem ser utilizadas como identificadores.

D Tipos primitivos

Tipo	Tamanho em bits	Valores	Padrão
`boolean`		`true` ou `false`	
[*Nota*: a representação de um boolean é específica à Java Virtual Machine em cada plataforma.]			
`char`	16	`'\u0000'` a `'\uFFFF'` (0 a 65535)	(conjunto de caracteres Unicode ISO)
`byte`	8	–128 a +127 (-2^7 a $2^7 - 1$)	
`short`	16	–32.768 a +32.767 (–215 a 215 – 1)	
`int`	32	–2.147.483.648 a +2.147.483.647 (-2^{31} a $2^{31} - 1$)	
`long`	64	–9.223.372.036.854.775.808 a +9.223.372.036.854.775.807 (-2^{63} a $2^{63} - 1$)	
`float`	32	*Intervalo negativo:* –3,4028234663852886E+38 a –1,40129846432481707e–45 *Intervalo positivo:* 1,40129846432481707e–45 a 3,4028234663852886E+38	(ponto flutuante IEEE 754)
`double`	64	*Intervalo negativo:* –1,7976931348623157E+308 a –4,94065645841246544e–324 *Intervalo positivo:* 4,94065645841246544e–324 a 1,7976931348623157E+308	(ponto flutuante IEEE 754)

Figura D.1 | Tipos primitivos do Java.

Para obter mais informações sobre o IEEE 754, visite `grouper.ieee.org/groups/754/`. Para informações adicionais sobre o Unicode, consulte o Apêndice L, "Unicode®".

Usando a documentação da Java API

E

E.1 Introdução

A biblioteca de classes Java contém milhares de classes e interfaces predefinidas que os programadores podem utilizar para escrever seus próprios aplicativos. Essas classes são agrupadas em pacotes com base nas suas funcionalidades. Por exemplo, as classes e interfaces utilizadas para processamento de arquivos estão agrupadas no pacote `java.io` e as classes e interfaces para aplicativos de rede estão agrupadas no pacote `java.net`. A **documentação da Java API** lista os membros `public` e `protected` de cada classe e os membros `public` de cada interface na biblioteca de classes Java. A documentação apresenta uma visão geral de todas as classes e interfaces, resume seus membros (isto é, os campos, construtores e métodos das classes e os campos e métodos das interfaces) e fornece descrições detalhadas sobre cada membro. A maioria dos programadores Java conta com essa documentação ao escrever programas. Normalmente, os programadores pesquisariam a API para encontrar o seguinte:

1. o pacote que contém uma classe ou interface particular;
2. os relacionamentos entre uma classe ou interface particular e outras classes e interfaces;
3. classe ou constantes de interface — normalmente declaradas como campos `public static final`;
4. construtores para determinar como um objeto da classe pode ser inicializado;
5. os métodos de uma classe para determinar se eles são `static` ou não `static`, os números e tipos dos argumentos que você precisa passar, os tipos de retorno e quaisquer exceções que poderiam ser lançadas a partir do método.

Além disso, os programadores costumam recorrer à documentação para descobrir classes e interfaces que eles ainda não usaram. Por isso, demonstramos a documentação com classes que você conhece e com classes que talvez ainda não tenha estudado. Mostramos como usar a documentação para localizar as informações necessárias para usar uma classe ou interface eficientemente.

E.2 Navegando pela Java API

Você pode fazer download da documentação da Java API para seu disco rígido local ou visualizá-la on-line. Para fazer o download da documentação da Java API, vá para `java.sun.com/javase/downloads/` role até a seção **Additional Resources** e clique no botão **Download** à direita de **Java SE 6 Documentation**. Você será solicitado a aceitar um contrato de licença. Para fazer isso, clique em **Accept** e então clique em **Continue**. Marque a caixa de seleção ao lado de **Java SE Development Kit Documentation 6** e então clique no link `jdk-6u10-docs.zip` logo abaixo da caixa de seleção. Depois de baixar o arquivo compactado, você pode usar um programa como o WinZip (`www.winzip.com`) para extrair os arquivos. Se estiver utilizando Windows, extraia o conteúdo do diretório de instalação do seu JDK. Para visualizar a documentação da API no seu disco rígido local no Microsoft Windows, abra a página `C:\Program Files\Java\`*`SuaVersãoDoJDK`*`\docs\api\index.html` no seu navegador. Para examinar a documentação de API on-line, vá para `java.sun.com/javase/6/docs/api/index.html` (Figura E.1).

Frames na página `index.html` da documentação da API

A documentação da API está dividida em três frames (veja a Figura E.1). O frame superior esquerdo lista todos os pacotes da Java API em ordem alfabética. O frame inferior esquerdo lista as classes e interfaces da Java API em ordem alfabética. Nomes de interfaces são exibidos

em itálico. Ao clicar em um pacote específico no frame superior esquerdo, o frame esquerdo inferior lista as classes e interfaces do pacote selecionado. O frame direito fornece inicialmente uma breve descrição de cada pacote da especificação da Java API — leia essa visão geral para conhecer as capacidades gerais das Java APIs. Se selecionar uma classe ou interface no frame esquerdo inferior, o frame direito exibe informações sobre essa classe ou interface.

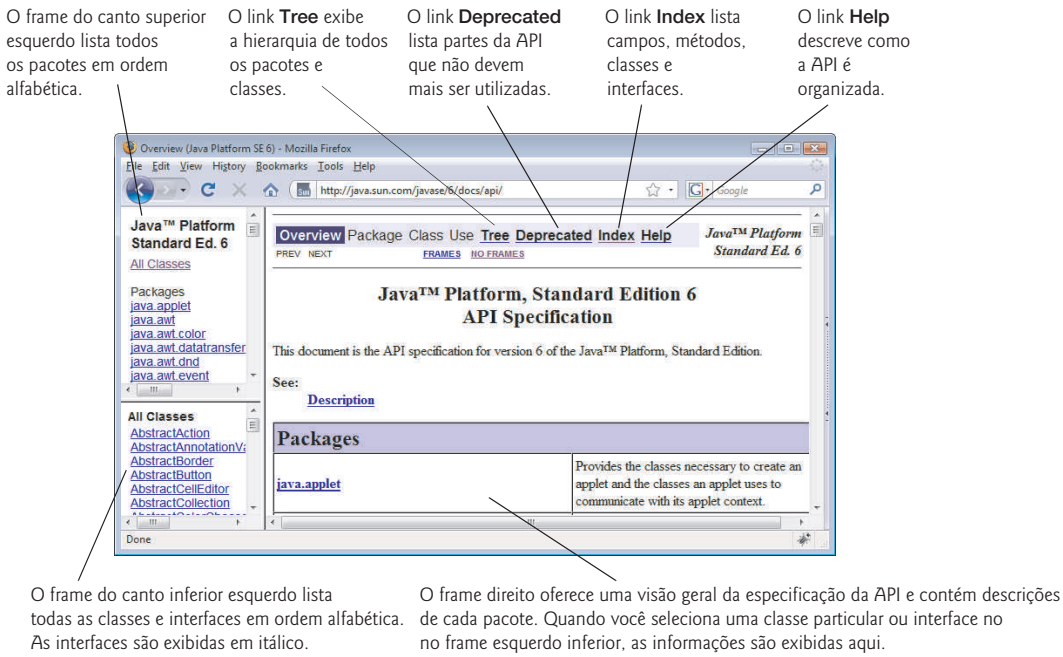


Figura E.1 | Visão geral da Java API. (Cortesia da Sun Microsystems, Inc.)

Links importantes na página `index.html`

Na parte superior do frame direito (Figura E.1), há quatro links — **Tree**, **Deprecated**, **Index** e **Help**. O link **Tree** exibe a hierarquia de todos os pacotes, classes e interfaces em uma estrutura de árvore. O link **Deprecated** exibe interfaces, classes, exceções, campos, construtores e métodos que não devem ser mais utilizados. O link **Index** exibe classes, interfaces, campos, construtores e métodos em ordem alfabética. O link **Help** descreve como a documentação da API é organizada. Você provavelmente deve iniciar lendo a página **Help**.

Visualizando a página **Index**

Se não souber o nome da classe que você está procurando, mas souber o nome de um método ou campo, poderá utilizar o índice da documentação para localizar a classe. O link **Index** está localizado próximo ao canto superior direito do frame direito. A página de índice (Figura E.2) exibe campos, construtores, métodos, interfaces e classes em ordem alfabética. Por exemplo, se estiver procurando o método

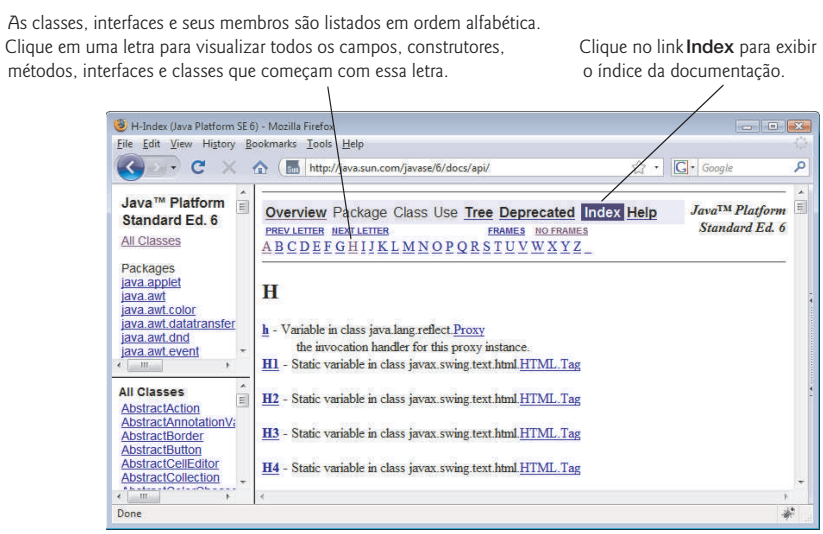


Figura E.2 | Visualizando a página **Index**. (Cortesia da Sun Microsystems, Inc.)

`hasNextInt` de `Scanner` e não conhecer o nome da classe, poderá clicar no link `H` a fim de ir para a listagem alfabética de todos os itens na Java API que começam com `"h"`. Role até o método `hasNextInt` (Figura E.3). Depois que estiver aí, cada método chamado `hasNextInt` é listado com o nome do pacote e classe ao qual o método pertence. A partir daí, você pode clicar no nome da classe para visualizar os detalhes completos da classe ou pode clicar no nome do método para visualizar os detalhes desse método.

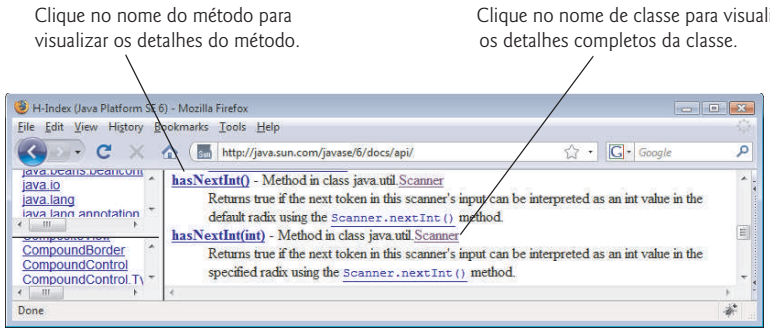


Figura E.3 | Role até o método `hasNextInt`. (Cortesia da Sun Microsystems, Inc.)

Visualizando um pacote específico

Quando você clicar no nome de pacote no frame superior esquerdo, todas as classes e interfaces desse pacote são exibidas no frame inferior esquerdo e são divididas em cinco subseções — **Interfaces**, **Classes**, **Enums**, **Exceptions** e **Errors** — cada um listado alfabeticamente. Por exemplo, os conteúdos do pacote `javax.swing` são exibidos no frame inferior esquerdo (Figura E.4) quando você clica em `javax.swing` no frame superior esquerdo. Você pode clicar no nome de pacote no frame esquerdo inferior para obter uma visão geral do pacote. Se achar que um pacote contém várias classes que poderiam ser úteis no seu aplicativo, a visão geral do pacote poderá ser especialmente útil.

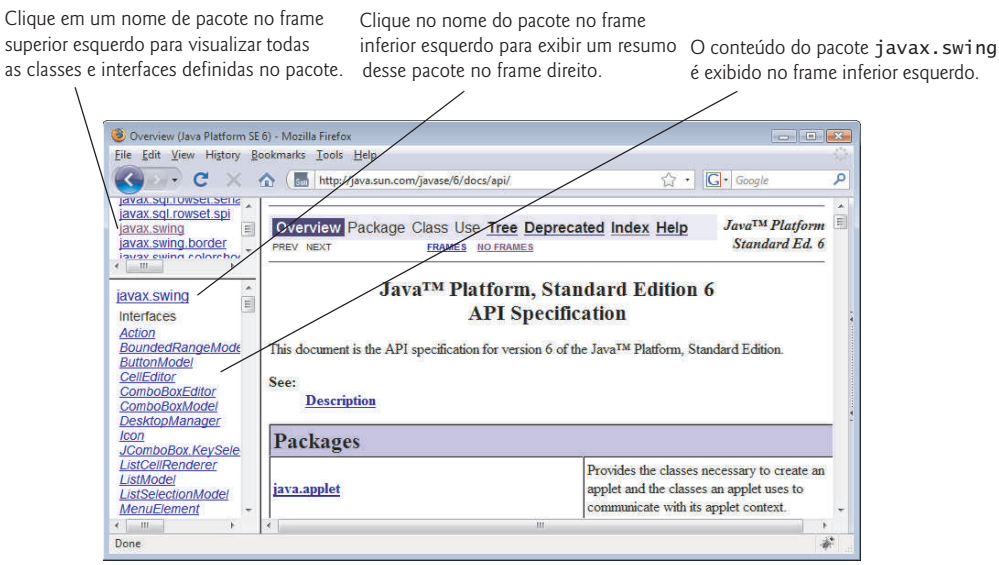


Figura E.4 | Clicar em um nome de pacote no frame superior esquerdo para visualizar todas as classes e interfaces declaradas nesse pacote. (Cortesia da Sun Microsystems, Inc.)

Visualizando os detalhes de uma classe

Ao clicar em um nome de classe ou em um nome de interface no frame esquerdo inferior, o frame direito exibe os detalhes dessa classe ou interface. Primeiro, você verá o nome do pacote da classe seguido por uma hierarquia que mostra o relacionamento da classe com outras classes. Você também verá uma lista das interfaces implementadas pela classe e as subclasses conhecidas dessa classe. A Figura E.5 mostra o começo da página de documentação para a classe `JButton` do pacote `javax.swing`. A página primeiro mostra o nome do pacote em que a classe aparece. Isso é seguido pela hierarquia de classes que leva à classe `JButton`, as interfaces que a classe `JButton` implementa e as subclasses da classe `JButton`. A parte inferior do frame direito mostra o começo da descrição da classe `JButton`. Observe que quando você vê a documentação de uma interface, o frame direito não exibe uma hierarquia dessa interface. Em vez disso, o frame direito lista as superinterfaces da interface, subinterfaces conhecidas e classes de implementação conhecidas.

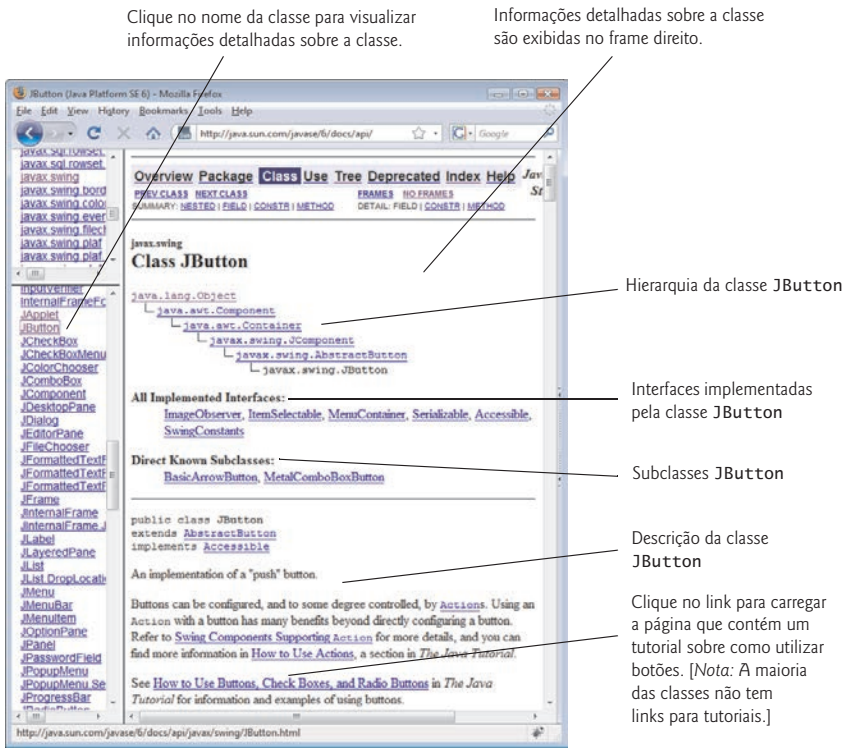


Figura E.5 | Clicando no nome de uma classe para visualizar informações detalhadas sobre a classe. (Cortesia da Sun Microsystems, Inc.)

Seções de resumo na página de documentação de uma classe

Outras partes de cada página da API estão listadas a seguir. Cada parte só é apresentada se a classe contiver ou herdar os itens especificados. Os membros de classe mostrados nas seções de resumo são `public` a menos que eles sejam explicitamente marcados como `protected`. Os membros `private` de uma classe não são mostrados na documentação, uma vez que eles não podem ser utilizados diretamente nos seus programas.

1. A seção **Nested Class Summary** resume as classes aninhadas `public` e `protected` da classe — isto é, classes definidas dentro da classe. A menos que explicitamente especificado, essas classes são `public` e não `static`.
2. A seção **Field Summary** resume os campos `public` e `protected` da classe. A menos que explicitamente especificado, esses campos são `public` e não `static`. A Figura E.6 mostra a seção **Field Summary** da classe `Color`.

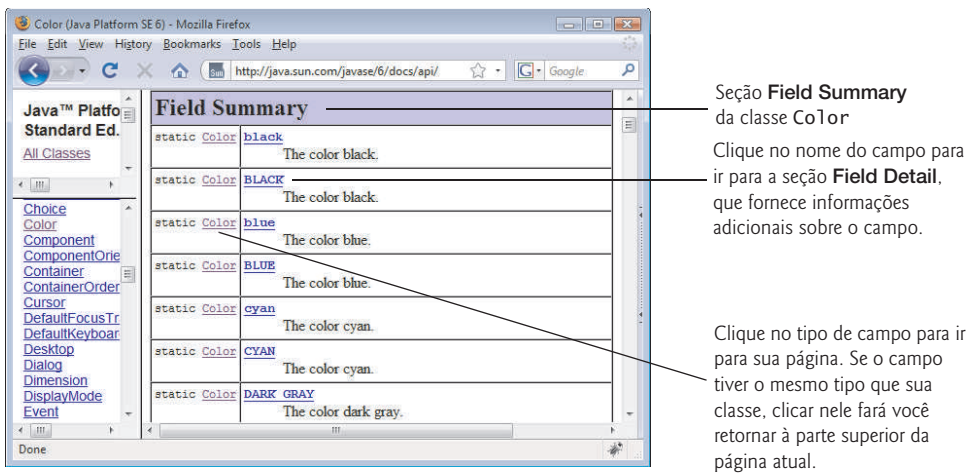


Figura E.6 | Seção **Field Summary** da classe `Color`. (Cortesia da Sun Microsystems, Inc.)

3. A seção **Constructor Summary** resume os construtores da classe. Estes não são herdados, portanto essa seção só aparece na documentação para uma classe se a classe declarar um ou mais construtores. A Figura E.7 mostra a seção **Constructor Summary** da classe `JButton`.

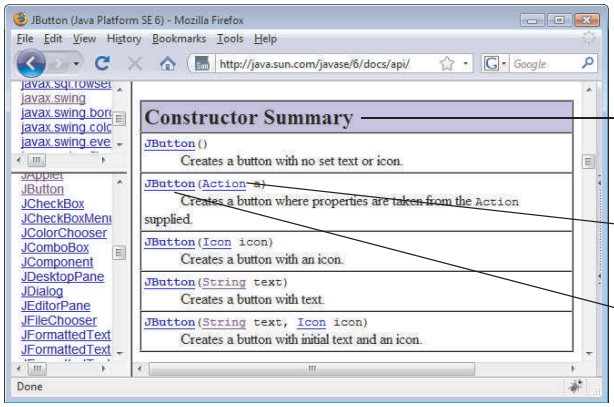


Figura E.7 | Seção **Constructor Summary** da classe `JButton`. (Cortesia da Sun Microsystems, Inc.)

4. A seção **Method Summary** resume os métodos `public` e `protected` da classe. A menos que explicitamente especificado, esses métodos são `public` e não `static`. A Figura E.8 mostra a seção **Method Summary** da classe `BufferedInputStream`.

Observe que, em geral, as seções de resumo fornecem apenas uma descrição de uma frase do membro de uma classe. Detalhes adicionais são apresentados nas seções de detalhes discutidas a seguir.

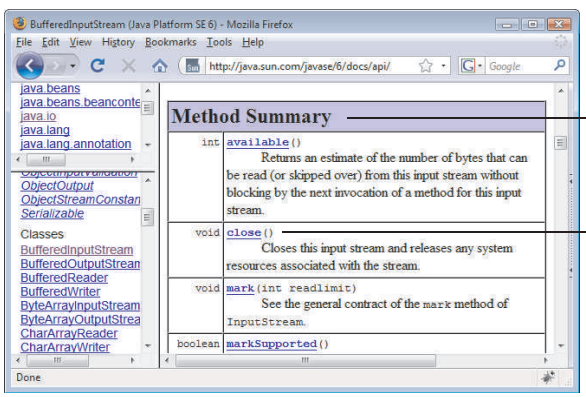


Figura E.8 | Seção **Method Summary** da classe `BufferedInputStream`. (Cortesia da Sun Microsystems, Inc.)

Seções de detalhes na página de Documentação de uma classe

Depois das seções de resumo, há seções de detalhes que normalmente fornecem mais discussão sobre membros particulares de classe. Não há uma seção de detalhes para classes aninhadas. Quando você clica no link no **Nested Class Summary** para uma classe aninhada em particular, é exibida uma página de documentação descrevendo essa classe aninhada. As seções de detalhes são descritas a seguir.

1. A seção **Field Detail** fornece a declaração de cada campo. Ela também discute cada campo, incluindo os modificadores e significado do campo. A Figura E.9 mostra a seção **Field Detail** da classe `Color`.

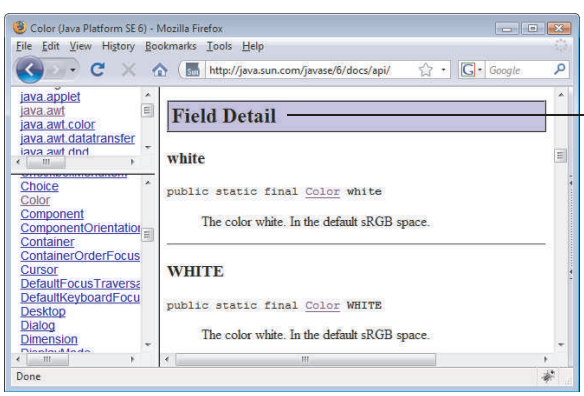


Figura E.9 | Seção **Field Detail** da classe `Color`. (Cortesia da Sun Microsystems, Inc.)

2. A seção **Constructor Detail** fornece a primeira linha da declaração de cada construtor e discute os construtores. A discussão inclui os modificadores de cada construtor, uma descrição de cada construtor, os parâmetros de cada construtor e quaisquer exceções lançadas por cada construtor. A Figura E.10 mostra a seção Constructor Detail da classe `JButton`.

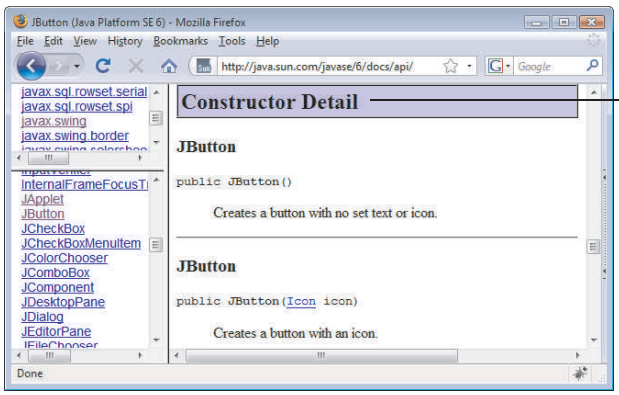


Figura E.10 | Seção **Constructor Detail** da classe `JButton`. (Cortesia da Sun Microsystems, Inc.)

3. A seção **Method Detail** fornece a primeira linha de cada método. A discussão de cada método inclui seus modificadores, uma descrição mais completa do método, os parâmetros do método, o tipo de retorno do método e quaisquer exceções lançadas pelo método. A Figura E.11 mostra a seção Method Detail da classe `BufferedInputStream`.

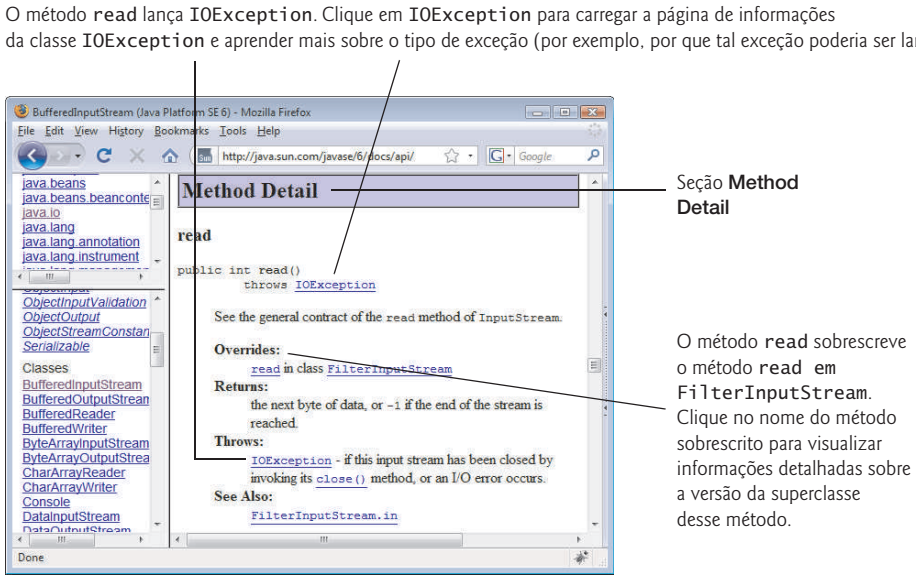


Figura E.11 | Seção **Method Detail** da classe `BufferedInputStream`. (Cortesia da Sun Microsystems, Inc.)

Os detalhes de método mostram outros métodos que talvez sejam de interesse (rotulados como **See Also**). Se o método sobrescrever um método da superclasse, o nome do método da superclasse e o nome da superclasse serão fornecidos para que você possa vincular-se ao método ou superclasse para informações adicionais.

À medida que examina a documentação, você notará que há links para outros campos, métodos, classes aninhadas e classes de primeiro nível. Esses links permitem pular da classe que você está examinando para outra parte relevante da documentação.

Fomos criados para cometer equívocos, programados para erro.
— Lewis Thomas

O que antecipamos raramente ocorre; o que menos esperamos geralmente acontece.
— Benjamin Disraeli

Ele pode correr, mas não pode se esconder.
— Joe Louis

Uma coisa é mostrar a um homem que ele está errado e, uma outra, é colocá-lo de posse da verdade.
— John Locke

F

Utilizando o depurador

Objetivos

Neste apêndice, você aprenderá:

- A configurar pontos de interrupção para depurar aplicativos.
- A utilizar o comando `run` para executar um aplicativo por meio do depurador.
- A utilizar o comando `stop` para configurar um ponto de interrupção.
- A utilizar o comando `cont` para continuar a execução.
- A utilizar o comando `print` para avaliar expressões.
- A utilizar o comando `set` para alterar valores de variáveis durante a execução do programa.
- A utilizar os comandos `step`, `step up` e `next` para controlar a execução.
- A utilizar o comando `watch` para ver como um campo é modificado durante a execução do programa.
- A utilizar o comando `clear` para listar pontos de interrupção ou remover um ponto de interrupção.

Sumário

F.1 Introdução

F.2 Pontos de interrupção e os comandos `run`, `stop`, `cont` e `print`

F.3 Os comandos `print` e `set`

F.4 Controlando a execução utilizando os comandos `step`, `step up` e `next`

F.5 O comando `watch`

F.6 O comando `clear`

F.7 Resumo

Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão

F.1 Introdução

No Capítulo 2, você aprendeu que há dois tipos de erros — erros de sintaxe e erros de lógica — e aprendeu a eliminar erros de sintaxe do seu código. Erros de lógica não impedem que o aplicativo compile com sucesso, mas fazem com que ele produza resultados errôneos quando executado. O JDK inclui um software chamado **depurador** que permite monitorar a execução dos seus aplicativos para que você possa localizar e remover erros de lógica. O depurador será uma das suas ferramentas mais importantes de desenvolvimento de aplicativos. Muitos IDEs fornecem depuradores próprios semelhantes àquele incluído no JDK ou fornecem uma interface gráfica com o usuário para o depurador do JDK.

Este apêndice demonstra os recursos-chave do depurador do JDK que utilizam aplicativos de linha de comando e não recebem nenhuma entrada do usuário. Os mesmos recursos de depurador discutidos aqui podem ser utilizados para depurar aplicativos que recebem a entrada do usuário, mas depurar esses aplicativos requer uma configuração um pouco mais complexa. Para focalizar os recursos do depurador, optamos por demonstrá-lo com aplicativos de linha de comando simples que não envolvem nenhuma entrada do usuário. Você também pode localizar mais informações sobre o depurador de Java em `java.sun.com/javase/6/docs/technotes/tools/windows/jdb.html`.

F.2 Pontos de interrupção e os comandos `run`, `stop`, `cont` e `print`

Iniciamos nosso estudo do depurador investigando os **pontos de interrupção**, que são marcadores que podem ser configurados em qualquer linha executável do código. Quando a execução do aplicativo alcança um ponto de interrupção, a execução pausa, permitindo que você examine os valores das variáveis para ajudar a determinar se há erros de lógica. Por exemplo, você pode examinar o valor de uma variável que armazena o resultado de um cálculo a fim de determinar se o cálculo foi realizado corretamente. Observe que configurar um ponto de interrupção em uma linha de código que não é executável (como um comentário) faz com que o depurador exiba uma mensagem de erro.

Para ilustrar os recursos do depurador, utilizaremos o aplicativo `AccountTest` (Figura F.1), que cria e manipula um objeto da classe `Account` (Figura 3.13). A execução de `AccountTest` inicia em `main` (linhas 7–24). A linha 9 cria um objeto `Account` com um saldo inicial de US$ 50.00. Lembre-se de que o construtor de `Account` aceita um argumento, que especifica o `balance` (saldo) inicial de `Account`. As linhas 12–13 geram a saída do saldo inicial na conta utilizando o método `getBalance` de `Account`. A linha 15 declara e inicializa uma variável local `depositAmount`. As linhas 17–19 então imprimem `depositAmount` e o adicionam ao `balance` de `Account` utilizando seu método `credit`. Por fim, as linhas 22–23 exibem o novo `balance`. [*Nota*: O diretório de exemplos do Apêndice F contém uma cópia de `Account.java` idêntica àquela na Figura 3.13.]

```
1  // Figura F.1: AccountTest.Java
2  // Cria e manipula um objeto Account.
3
4  public class AccountTest
5  {
6     // método main inicia a execução
7     public static void main( String[] args )
8     {
9        Account account = new Account( 50.00 ); // cria o objeto Account
10
11       // exibe o saldo inicial do objeto Account
12       System.out.printf( "initial account balance: $%.2f\n",
13          account.getBalance() );
```

```
14
15          double depositAmount = 25.0; // deposita uma quantia
16
17          System.out.printf( "\nadding %.2f to account balance\n\n",
18             depositAmount );
19          account.credit( depositAmount ); // adiciona ao saldo da conta
20
21          // exibe um novo saldo
22          System.out.printf( "new account balance: $%.2f\n",
23             account.getBalance() );
24       } // fim de main
25
26    } // fim da classe AccountTest
```

```
initial account balance: $50.00

adding 25.00 to account balance

new account balance: $75.00
```

Figura F.1 | A classe `AccountTest` cria e manipula um objeto `Account`.

Nos passos a seguir, utilize pontos de interrupção e vários comandos de depurador para examinar o valor da variável `depositAmount` declarada em `AccountTest` (Figura F.1)

1. ***Abrindo a janela Prompt do MS-DOS e mudando de diretório.*** Abra a janela Command Prompt selecionando Start> Programs> Accessories> Command Prompt. Mude para o diretório contendo os exemplos do Apêndice F digitando `cd C:\examples\debugger` [*Nota:* Se seus exemplos estiverem em um diretório diferente, utilize esse diretório aqui].
2. ***Compilando o aplicativo para depuração***. O depurador Java só funciona com os arquivos `.class` compilados com a opção de compilador **`-g`**, que gera informações utilizadas pelo depurador para ajudar a depurar seus aplicativos. Compile o aplicativo com a opção de linha de comando `-g` digitando `javac -g AccountTest.java Account.java`. A partir do Capítulo 3, lembre-se de que esse comando compila tanto `AccountTest.java` como `Account.java`. O comando `java -g *.java` compila todos os arquivos `.java` do diretório de trabalho para depuração.
3. ***Iniciando o depurador.*** No Prompt de Comando, digite **`jdb`** (Figura F.2). Esse comando iniciará o depurador Java e permitirá que você utilize os seus recursos. [*Nota*: modificamos as cores da nossa janela Command Prompt por questão de legibilidade.]

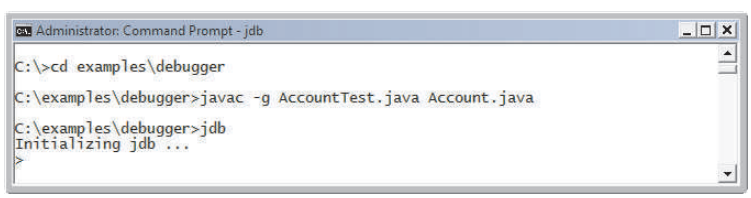


Figura F.2 | Iniciando o depurador Java.

4. ***Executando um aplicativo no depurador.*** Execute o aplicativo `AccountTest` por meio do depurador digitando **`run`** `AccountTest` (Figura F.3). Se você não configurar pontos de interrupção antes de executar seu aplicativo no depurador, o aplicativo executará como faria utilizando o comando `java`.

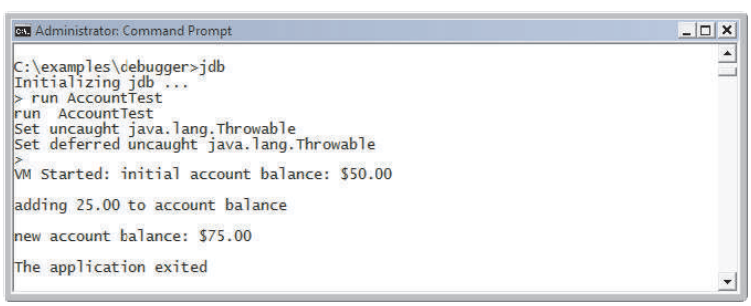


Figura F.3 | Executando o aplicativo `AccountTest` por meio do depurador.

5. ***Reiniciando o depurador.*** Para utilização adequada do depurador, você deve configurar pelo menos um ponto de interrupção antes de executar o aplicativo. Reinicie o depurador digitando `jdb`.
6. ***Inserindo pontos de interrupção em Java.*** Você configura um ponto de interrupção em uma linha específica do código no seu aplicativo. Os números das linhas utilizados nestes passos são provenientes do código-fonte na Figura F.1. Configure um ponto de interrupção na linha 12 do código-fonte digitando `stop at AccountTest:12` (Figura F.4). O **comando `stop`** insere um ponto de interrupção no número da linha especificada depois do comando. Você pode configurar quantos pontos de interrupção forem necessários. Configure outro ponto de interrupção na linha 19 digitando `stop at AccountTest:19` (Figura F.4). Quando o aplicativo executa, ele interrompe a execução em qualquer linha que contém um ponto de interrupção. Diz-se que o aplicativo está no **modo de interrupção** (**break mode**) quando o depurador pausa a execução do aplicativo. Pontos de interrupção podem ser até mesmo configurados depois de o processo de depuração ter iniciado. Observe que o comando de depurador `stop in`, seguido por um nome de classe, um ponto e um nome de método (por exemplo, `stop in Account.credit`) instrui o depurador a configurar um ponto de interrupção na primeira instrução executável do método especificado. O depurador pausa a execução quando o controle do programa entra no método.

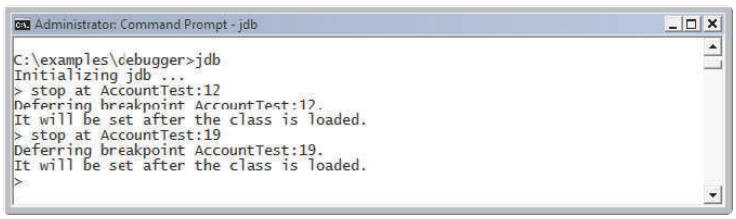


Figura F.4 | Configurando pontos de interrupção nas linhas 12 e 19.

7. ***Executando o aplicativo e começando o processo de depuração.*** Digite `run AccountTest` para executar o aplicativo e começar o processo de depuração (Figura F.5) O depurador imprime texto indicando que os pontos de interrupção foram configurados nas linhas 12 e 19. Ele identifica cada ponto de interrupção como um "deferred breakpoint" ("ponto de interrupção adiado"), pois cada um foi configurado antes de o aplicativo iniciar a execução no depurador. O aplicativo pausa quando a execução alcança o ponto de interrupção na linha 12. Nesse ponto, o depurador o notifica de que um ponto de interrupção foi alcançado e exibe o código-fonte nessa linha (12). Essa linha do código é a próxima instrução que será executada.

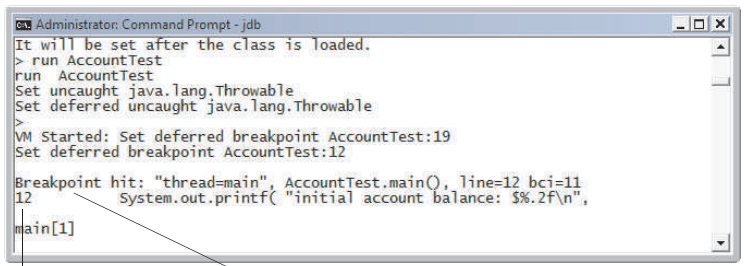


Figura F.5 | Reiniciando o aplicativo `AccountTest`.

8. ***Utilizando o comando `cont` para resumir a execução.*** Digite `cont`. O **comando `cont`** faz com que o aplicativo continue a execução até o próximo ponto de interrupção ser alcançado (linha 19), quando o depurador o notifica (Figura F.6). Observe que a saída normal de `AccountTest` aparece entre as mensagens no depurador.

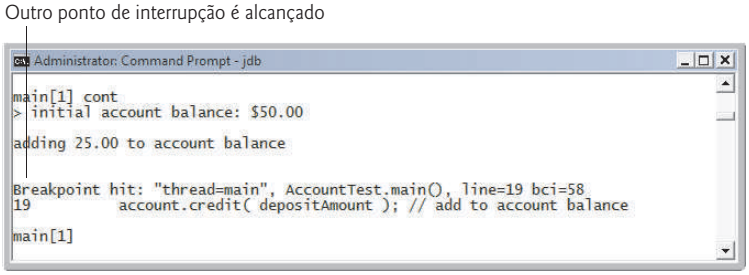


Figura F.6 | A execução alcança o segundo ponto de interrupção.

9. ***Examinando o valor de uma variável.*** Digite `print depositAmount` para exibir o valor atual armazenado na variável `depositAmount` (Figura F.7). O **comando `print`** permite examinar dentro do computador o valor de uma das suas variáveis. Esse comando lhe ajudará a encontrar e eliminar erros de lógica no seu código. Observe que o valor exibido é `25.0` — o valor atribuído a `deposit-Amount` na linha 15 da Figura F.1.

Figura F.7 | Examinando o valor da variável `depositAmount`.

10. ***Continuando a execução do aplicativo.*** Digite `cont` para continuar a execução do aplicativo. Não há nenhum ponto de interrupção, portanto o aplicativo não está mais no modo de interrupção. O aplicativo continua a execução e consequentemente termina (Figura F.8). O depurador irá parar quando o aplicativo terminar.

Figura F.8 | Continuando a execução do aplicativo e encerrando o depurador.

Nesta seção, você aprendeu a ativar o depurador e a configurar pontos de interrupção para examinar variáveis com o comando `print` enquanto um aplicativo está em execução. Você também aprendeu a utilizar o comando `cont` para continuar a execução depois que um ponto de interrupção é alcançado.

F.3 Os comandos `print` e `set`

Na seção anterior, você aprendeu a utilizar o comando `print` do depurador para examinar o valor de uma variável durante a execução do programa. Nesta seção, você aprenderá a utilizar o comando `print` para examinar o valor de expressões mais complexas. Você também aprenderá como utilizar o **comando `set`**, que permite ao programador atribuir novos valores a variáveis.

Para esta seção, supomos que você seguiu os *Passos 1* e *2*, da Seção F.2, para abrir a janela Command Prompt, mudar para o diretório que contém os exemplos deste apêndice (por exemplo, `C:\examples\debugger`) e compilar o aplicativo `AccountTest` (e a classe `Account`) para depuração.

1. ***Iniciando a depuração.*** No Command Prompt, digite `jdb` para iniciar o depurador de Java.
2. ***Inserindo um ponto de interrupção.*** Configure um ponto de interrupção na linha 19 no código-fonte digitando `stop at AccountTest:19`.
3. ***Executando o aplicativo e alcançando um ponto de interrupção***. Digite `run AccountTest` para começar o processo de depuração (Figura F.9). Isso fará com que o método `main` de `AccountTest` seja executado até o ponto de interrupção na linha 19 ser alcançado. Isso suspende a execução do aplicativo e alterna o aplicativo para o modo de interrupção. Nesse ponto, as instruções nas linhas 9–13 criaram um objeto `Account` e imprimiram o saldo inicial da `Account` obtido chamando o método `getBalance`. A instrução na linha 15 (Figura F.1) declarou e inicializou a variável local `depositAmount` como `25.0`. A instrução na linha 19 é a próxima instrução que será executada.

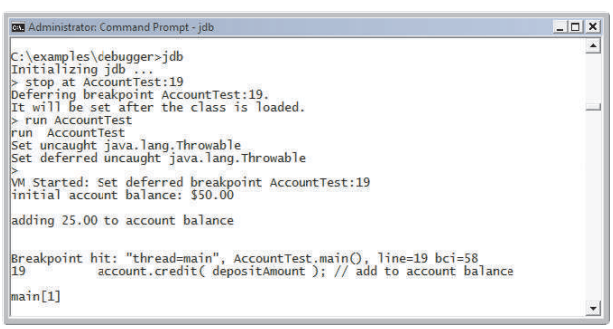


Figura F.9 | A execução do aplicativo é suspensa quando o depurador alcança o ponto de interrupção na linha 19.

4. ***Avaliando expressões aritméticas e booleanas.*** A partir da Seção F.2, lembre-se de que depois que o aplicativo entra no modo de interrupção, você pode explorar os valores das variáveis do aplicativo usando o comando `print` do depurador. Você também pode usar o comando `print` para avaliar expressões aritméticas e `booleanas`. Na janela **Command Prompt**, digite `print depositAmount - 2.0`. Observe que o comando `print` retorna o valor `23.0` (Figura F.10). Mas esse comando na verdade não muda o valor de `depositAmount`. Na janela **Command Prompt**, digite `print depositAmount == 23.0`. As expressões que contêm o símbolo == são tratadas como expressões `boolean`. O valor retornado é `false` (Figura. O f 10) porque `depositAmount` não contém atualmente o valor `23.0` — `depositAmount` ainda é `25.0`.

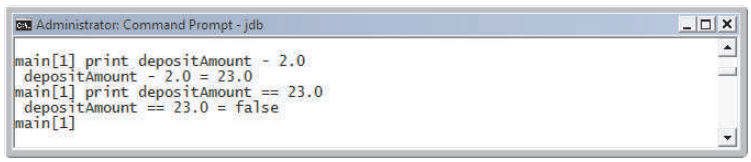


Figura F.10 | Examinando os valores de uma expressão aritmética e `boolean`.

5. ***Modificando valores.*** O depurador permite alterar os valores das variáveis durante a execução do aplicativo. Isso pode ser valioso para experimentar diferentes valores e localizar erros de lógica nos aplicativos. Você pode utilizar o comando `set` do depurador para alterar o valor de uma variável. Digite `set depositAmount = 75.0`. O depurador altera o valor de `depositAmount` e exibe seu novo valor (Figura F.11)

Figura F.11 | Modificando valores.

6. ***Visualizando o resultado do aplicativo.*** Digite `cont` para continuar a execução do aplicativo. A linha 19 de `AccountTest` (Figura F.1) executa, passando `depositAmount` para o método `credit` de `Account`. O método `main` então exibe o novo saldo. Observe que o resultado é `$125.00` (Figura F.12) Isso mostra que o passo anterior alterou o valor de `depositAmount` a partir do seu valor inicial (`25.0`) para `75.0`.

Novo saldo da conta com base no valor alterado da variável `depositAmount`

Figura F.12 | Saída exibida depois do processo de depuração.

Nesta seção, você aprendeu a utilizar o comando `print` do depurador para avaliar expressões aritméticas e `boolean`. Você também aprendeu a utilizar o comando `set` para modificar o valor de uma variável durante a execução do seu aplicativo.

F.4 Controlando a execução utilizando os comandos `step`, `step up` e `next`

Às vezes, será necessário executar um aplicativo linha por linha para encontrar e corrigir erros. Investigar uma parte do seu aplicativo desta maneira pode ajudá-lo a verificar se o código de um método executa corretamente. Nesta seção, você aprenderá a utilizar o depurador para essa tarefa. Os comandos que você aprenderá nesta seção permitem executar um método linha por linha, executar todas as instruções de um método de uma vez ou executar apenas as instruções remanescentes de um método (se você já tiver executado algumas instruções dentro do método).

Mais uma vez, supomos que você esteja trabalhando no diretório que contém os exemplos deste apêndice e que tenha compilado para depuração com a opção de compilador `-g`.

1. ***Iniciando o depurador.*** Inicie o depurador digitando `jdb`.
2. ***Configurando um ponto de interrupção.*** Digite `stop at AccountTest:19` para configurar um ponto de interrupção na linha 19.
3. ***Executando o aplicativo.*** Execute o aplicativo digitando `run AccountTest`. Depois de o aplicativo exibir as duas mensagens de saída, o depurador indica que o ponto de interrupção foi alcançado e exibe o código na linha 19 (Figura F.13). O depurador e o aplicativo então pausam e esperam o próximo comando a ser inserido.

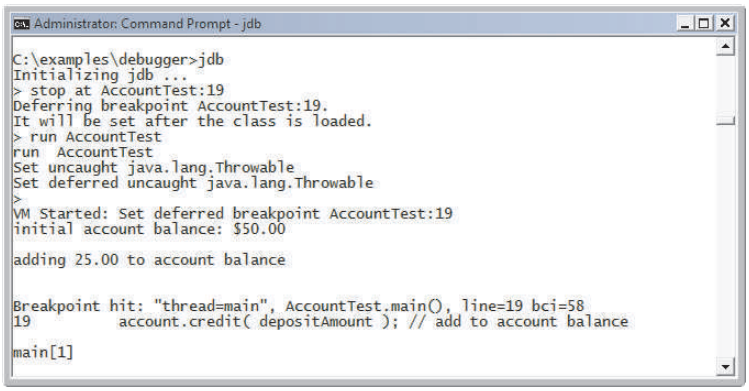


Figura F.13 | Alcançando o ponto de interrupção no aplicativo `AccountTest`.

4. ***Usando o comando step.*** O **comando `step`** executa a próxima instrução no aplicativo. Se a próxima instrução a executar for uma chamada de método, o controle será transferido para o método chamado. O comando `step` permite que você entre em um método e estude as instruções individuais desse método. Por exemplo, você pode utilizar os comandos `print` e `set` para visualizar e modificar as variáveis dentro do método. Você utilizará agora o comando `step` para entrar no método `credit` da classe `Account` (Figura 3.13) digitando `step` (Figura F.14). O depurador indica que o passo foi completado e exibe a próxima instrução executável — nesse caso, a linha 21 da classe `Account` (Figura 3.13).

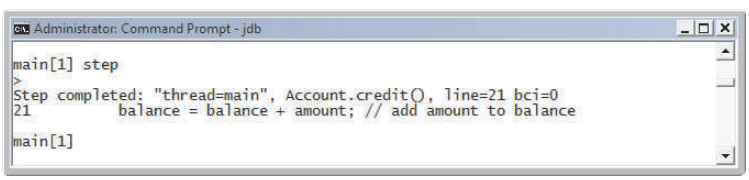


Figura F.14 | Entrando na inspeção (stepping into) do método `credit`.

5. ***Utilizando o comando step up***. Depois de entrar no método `credit` com Step Into, digite **`step up`**. Esse comando executa as instruções restantes no método e retorna o controle ao local em que o método foi chamado. O método `credit` contém apenas uma instrução para adicionar o parâmetro `amount` do método à variável de instância `balance`. O comando `step up` executa essa instrução e então pausa antes da linha 22 em `AccountTest`. Portanto, a próxima ação a ocorrer será imprimir o novo saldo da conta (Figura F.15) Em métodos longos, é recomendável examinar algumas linhas-chave do código e, então, continuar a depurar o código do chamador. O comando `step up` é útil para situações em que você não quer continuar a investigar todo o método linha por linha.
6. ***Utilizando o comando cont para continuar a execução.*** Insira o comando `cont` (Figura F.16) para continuar a execução. A instrução nas linhas 22–23 é executada, exibindo o novo saldo e então o aplicativo e o depurador terminam.

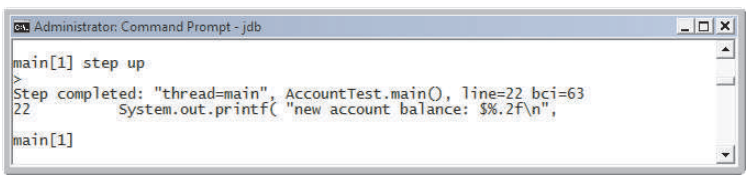


Figura F.15 | Saindo da inspeção (stepping out) de um método.

Figura F.16 | Continuando a execução do aplicativo `AccountTest`.

7. ***Reiniciando o depurador.*** Reinicie o depurador digitando `jdb`.
8. ***Configurando um ponto de interrupção.*** Os pontos de interrupção só persistem até o fim da sessão de depuração em que eles são configurados — depois de o depurador encerrar, todos os pontos de interrupção são removidos. (Na Seção F.6, você aprenderá a remover um ponto de interrupção manualmente antes do fim da sessão de depuração). Portanto, o ponto de interrupção configurado para a linha 19 no *Passo 2* não mais existirá depois da reinicialização do depurador no *Passo 7*. Para redefinir o ponto de interrupção na linha 19, digite mais uma vez `stop at AccountTest:19`.
9. ***Executando o aplicativo.*** Digite `run AccountTest` para executar o aplicativo. Como no *Passo 3*, `AccountTest` é executado até o ponto de interrupção na linha 19 ser alcançado, o depurador então pausa e espera o próximo comando (Figura F.17)

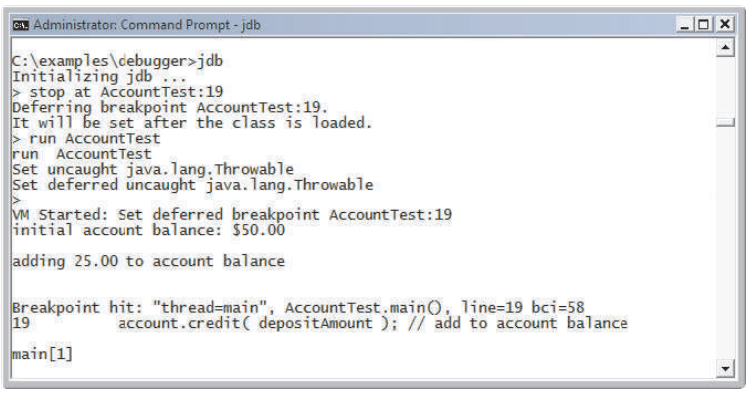


Figura F.17 | Alcançando o ponto de interrupção no aplicativo `AccountTest`.

10. ***Utilizando o comando next.*** Digite **`next`**. Esse comando se comporta como o comando `step`, exceto quando a próxima instrução a executar contém uma chamada de método. Nesse caso, o método chamado é executado na sua totalidade e o aplicativo avança para a próxima linha executável depois da chamada de método (Figura F.18) A partir do que foi discutido no *Passo 4*, lembre-se de que o comando `step` entraria no método chamado. Neste exemplo, o comando `next` faz com que método `Account credit` execute e então o depurador pausa na linha 22 em `AccountTest`.

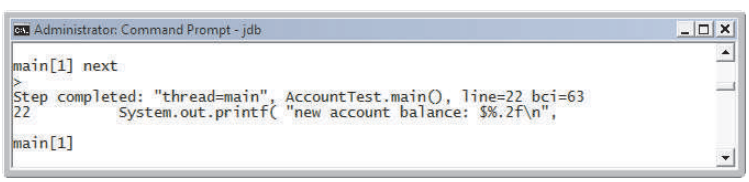


Figura F.18 | Inspeção (stepping over) de uma chamada de método.

11. ***Utilizando o comando exit.*** Utilize o **comando `exit`** para encerrar a sessão de depuração (Figura F.19). Esse comando faz com que o aplicativo `AccountTest` termine imediatamente em vez de executar as instruções restantes em `main`. Observe que ao depurar alguns tipos de aplicativos (por exemplo, aplicativos GUI), o aplicativo continuará a executar mesmo depois de a sessão de depuração terminar.

Figura F.19 | Encerrando o depurador.

Nesta seção, você aprendeu a utilizar os comandos `step` e `step up` do depurador para depurar métodos chamados durante a execução do seu aplicativo. Viu como o comando `next` pode ser utilizado para inspecionar uma chamada de método. Também aprendeu que o comando `exit` encerra uma sessão de depuração.

F.5 O comando watch

Nesta seção, apresentamos o **comando `watch`**, que instrui o depurador a monitorar um campo. Quando esse campo está em vias de ser alterado, o depurador o notificará. Aprenderá também a utilizar o comando `watch` para ver como o campo `balance` do objeto `Account` é modificado durante a execução do aplicativo `AccountTest`.

Como ocorreu nas duas seções anteriores, supomos que você seguiu o *Passo 1* e o *Passo 2* na Seção F.2 para abrir a Command Prompt, mudar para o diretório de exemplos correto e compilar as classes `AccountTest` e `Account` para depuração (isto é, com a opção de compilador `-g`).

1. ***Iniciando o depurador.*** Inicie o depurador digitando `jdb`.
2. ***Monitorando o campo de uma classe***. Configure um ponto de observação no campo `balance` digitando `watch Account.balance` (Figura F.20). Você pode configurar um watch em qualquer campo durante a execução do depurador. Sempre que o valor em um campo está em vias de mudar, o depurador entra no modo de interrupção e o notifica de que o valor irá mudar. Os pontos de monitoração só podem ser colocados em campos, não em variáveis locais.

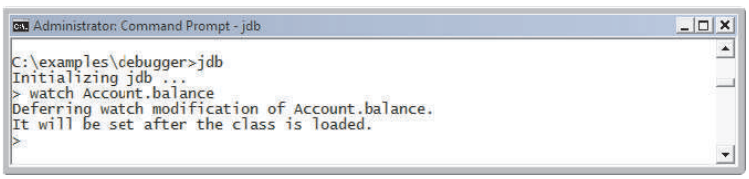


Figura F.20 | Configurando um watch no campo `balance` de `Account`.

3. ***Executando o aplicativo.*** Execute o aplicativo com o comando `run AccountTest`. O depurador agora o notificará que o valor do campo `balance` irá mudar (Figura F.21) Quando o aplicativo é inicializado, uma instância de `Account` é criada com um saldo inicial de US$ 50.00 e uma referência ao objeto `Account` é atribuído à variável local `account` (linha 9, Figura F.1). Com base na Figura 3.13, lembre-se de que quando o construtor desse objeto é executado, se o parâmetro `initialBalance` for maior que `0.0`, a variável de instância `balance` será atribuída ao valor do parâmetro `initialBalance`. O depurador o notifica de que o valor de `balance` será configurado como `50.0`.

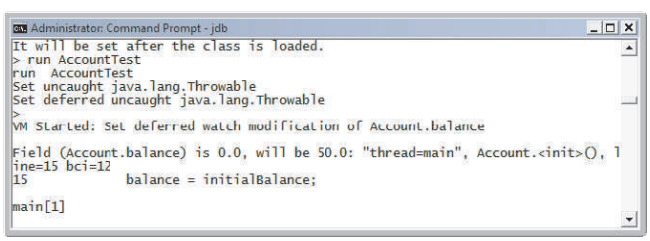


Figura F.21 | O aplicativo `AccountTest` para quando `account` é criado e seu campo `balance` será modificado.

4. ***Adicionando dinheiro à conta.*** Digite `cont` para continuar a executar o aplicativo. O aplicativo executa normalmente até alcançar o código na linha 19 da Figura F.1, que chama o método `credit` de `Account` para aumentar o `balance` do objeto `Account` por um `amount` especificado. O depurador o notifica de que a variável de instância `balance` irá mudar (Figura F.22) Observe que, embora a linha 19 da classe `AccountTest` chame o método `credit`, é a linha 21 do método `credit` de `Account` que na verdade altera o valor de `balance`.

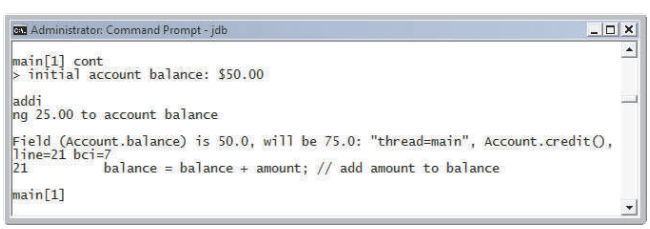


Figura F.22 | Altere o valor de `balance` chamando o método `Account` `credit`.

5. ***Continuando a execução.*** Digite `cont` — o aplicativo terminará a execução porque o aplicativo não tenta nenhuma alteração adicional em `balance` (Figura F.23).

Figura F.23 | Continuando a execução de `AccountTest`.

6. ***Reiniciando o depurador e redefinindo o ponto de monitoração sobre a variável.*** Digite `jdb` para reiniciar o depurador. Uma vez mais, configure um comando watch na variável de instância `Account.balance` digitando `watch Account.balance` e, então, digite `run AccountTest` para executar o aplicativo (Figura F.24).

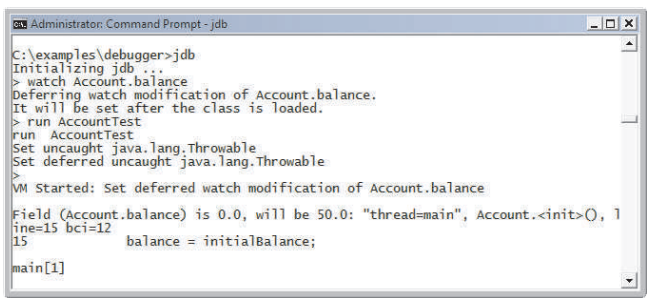


Figura F.24 | Reiniciando o depurador e redefinindo o ponto de monitoração na variável `balance`.

7. ***Removendo o ponto de monitoração no campo.*** Suponha que em um campo você queira monitorar somente uma parte da execução de um programa. É possível remover o ponto de monitoração do depurador na variável `balance` digitando **`unwatch`** `Account.balance` (Figura F.25). Digite `cont` — o aplicativo terminará a execução sem reentrar no modo de interrupção.

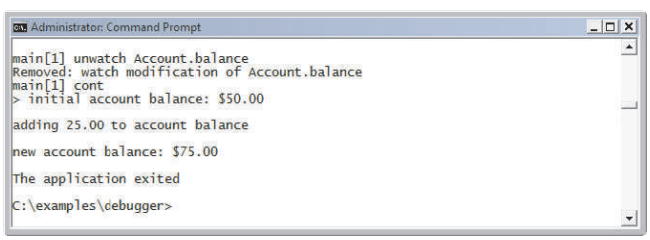


Figura F.25 | Removendo o ponto de monitoração na variável `balance`.

8. ***Fechando a janela Prompt do MS-DOS.*** Feche a janela Command Prompt clicando no botão de fechar.

Nesta seção, você aprendeu a utilizar o comando `watch` para ativar o depurador a fim de notificá-lo sobre alterações no valor de um campo por todo o ciclo de vida de um aplicativo. Também aprendeu a utilizar o comando `unwatch` para remover um watch em um campo antes do fim do aplicativo.

F.6 O comando `clear`

Na seção anterior, você aprendeu a utilizar o comando `unwatch` para remover um watch em um campo. O depurador também fornece o comando `clear` para remover um ponto de interrupção de um aplicativo. Frequentemente, precisará depurar aplicativos que contêm ações repetitivas, como um loop. Talvez você queira examinar os valores das variáveis durante várias, mas possivelmente não todas, iterações do loop. Se configurar um ponto de interrupção no corpo de um loop, o depurador pausará antes de cada execução da linha que contém um ponto de interrupção. Depois de determinar que o loop está funcionando adequadamente, talvez queira remover o ponto de interrupção e permitir que as iterações restantes prossigam normalmente. Nesta seção, utilizaremos o aplicativo de juros compostos na Figura 5.6 para demonstrar como o depurador comporta-se quando você configura um ponto de interrupção no corpo de uma instrução `for` e como remover um ponto de interrupção no meio de uma sessão de depuração.

1. ***Abrindo a janela Prompt do MS-DOS, mudando de diretório e compilando o aplicativo para depuração.*** Abra a janela Command Prompt e, então, mude para o diretório que contém os exemplos deste apêndice. Para sua conveniência, fornecemos uma cópia do arquivo `Interest.java` neste diretório. Compile o aplicativo para depuração digitando `javac -g Interest.java`.
2. ***Iniciando o depurador e configurando pontos de interrupção.*** Inicie o depurador digitando `jdb`. Configure os pontos de interrupção nas linhas 13 e 22 da classe `Interest` digitando `stop at Interest:13` e `stop at Interest:22` (Figura F.26).

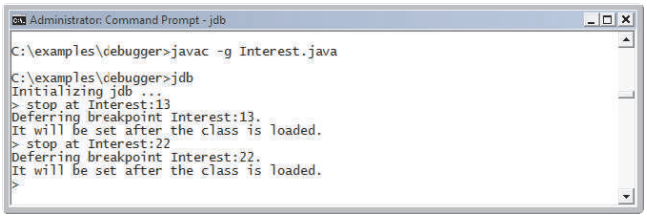


Figura F.26 | Configurando pontos de interrupção no aplicativo `Interest`.

3. ***Executando o aplicativo.*** Execute o aplicativo digitando `run Interest`. O aplicativo executa até alcançar o ponto de interrupção na linha 13 (Figura F.27).
4. ***Continuando a execução.*** Digite `cont` para continuar — o aplicativo executa a linha 13, imprimindo os títulos da coluna `"Year"` e `"Amount on deposit"`. Observe que a linha 13 aparece antes da instrução `for` nas linhas 16–23 em `Interest` (Figura 5.6) e assim é executada somente uma vez. A execução continua da linha 13 até o ponto de interrupção na linha 22 ser alcançado durante a primeira iteração da instrução `for` (Figura F.28).

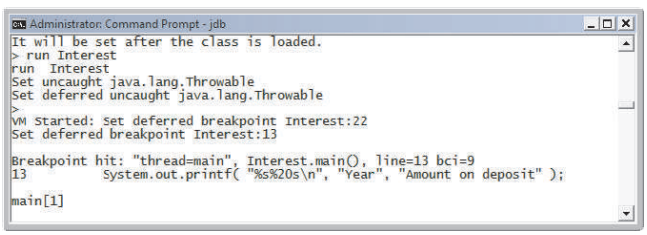


Figura F.27 | Alcançando o ponto de interrupção na linha 13 no aplicativo `Interest`.

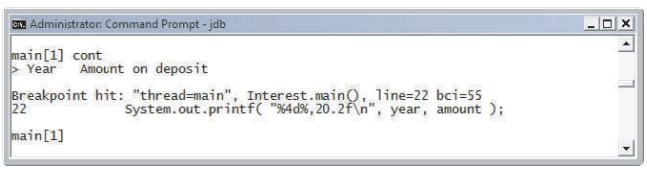


Figura F.28 | Alcançando o ponto de interrupção na linha 22 no aplicativo `Interest`.

5. ***Examinando valores de variáveis.*** Digite `print year` para examinar o valor atual da variável `year` (isto é, a variável de controle do `for`). Imprima também o valor da variável `amount` (Figura F.29).

Figura F.29 | Imprimindo `year` e `amount` durante a primeira iteração do `for` de `Interest`.

6. ***Continuando a execução.*** Digite `cont` para continuar a execução. A linha 22 executa e imprime os valores atuais de `year` e `amount`. Depois de o `for` entrar na sua segunda iteração, o depurador o notifica de que o ponto de interrupção na linha 22 foi alcançado uma segunda vez. Observe que o depurador pausa toda vez que uma linha em que um ponto de interrupção foi configurado está em

vias de executar — quando o ponto de interrupção aparece em um loop, o depurador pausa durante cada iteração. Imprima os valores das variáveis `year` e `amount` mais uma vez para ver como os valores mudaram desde a primeira iteração do `for` (Figura F.30).

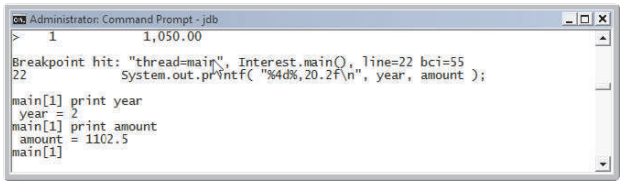


Figura F.30 | Imprimindo `year` e `amount` durante a segunda iteração do `for` de `Interest`.

7. ***Removendo um ponto de interrupção.*** Você pode exibir uma lista de todos os pontos de interrupção no aplicativo digitando **`clear`** (Figura F.31). Suponha que você esteja satisfeito com o funcionamento da instrução `for` do aplicativo `Interest`, assim quer remover o ponto de interrupção na linha 22 e permitir que as demais iterações do loop prossigam normalmente. Você pode remover o ponto de interrupção na linha 22 digitando `clear Interest:22`. Agora digite `clear` para listar os pontos de interrupção restantes no aplicativo. O depurador deve indicar que só o ponto de interrupção na linha 13 permanece (Figura F.31). Observe que esse ponto de interrupção já foi alcançado e assim não mais afetará a execução.

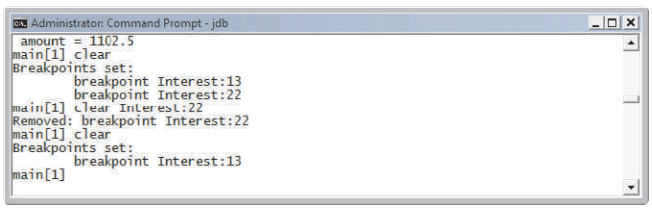


Figura F.31 | Removendo o ponto de interrupção na linha 22.

8. ***Continuando a execução depois de remover um ponto de interrupção.*** Digite `cont` para continuar a execução. Lembre-se de que a execução foi pausada pela última vez antes da instrução `printf` na linha 22. Se o ponto de interrupção na linha 22 foi removido com sucesso, continuar o aplicativo gerará a saída correta para as iterações atuais e remanescentes da instrução `for` sem que o aplicativo pare (Figura F.32).

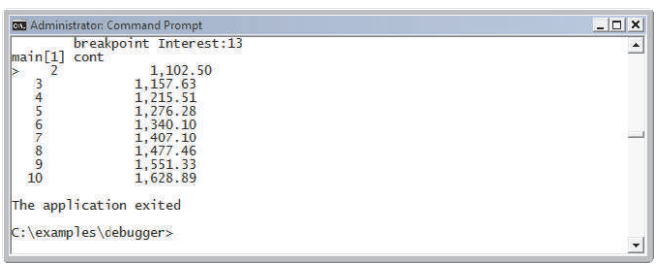


Figura F.32 | O aplicativo executa sem um ponto de interrupção configurado na linha 22.

Nesta seção, você aprendeu a utilizar o comando `clear` para listar todos os pontos de interrupção configurados para um aplicativo e a remover um ponto de interrupção.

F.7 Resumo

Neste apêndice, você aprendeu a inserir e remover pontos de interrupção no depurador. Os pontos de interrupção permitem pausar a execução do aplicativo para que possa examinar os valores das variáveis com o comando `print` do depurador. Essa capacidade o ajudará a localizar e corrigir erros de lógica nos seus aplicativos. Você viu como utilizar o comando `print` para examinar o valor de uma expressão e como utilizar o comando `set` para alterar o valor de uma variável. Também aprendeu os comandos de depurador (incluindo os comandos `step`, `step up` e `next`) que podem ser utilizados para determinar se um método está executando corretamente. Aprendeu ainda a utilizar o comando `watch` para monitorar um campo por toda a vida de um aplicativo. Por fim, você aprendeu a utilizar o comando `clear` para listar todos os pontos de interrupção configurados para um aplicativo ou a remover pontos de interrupção individuais para continuar a execução sem pontos de interrupção.

Exercícios de autorrevisão

F.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um ponto de interrupção não pode ser configurado em um(a) _____.

b) Você pode examinar o valor de uma expressão utilizando o comando _____ do depurador.

c) Você pode modificar o valor de uma variável utilizando o comando _____ do depurador.

d) Durante a depuração, o comando _____ executa as instruções restantes do método atual e retorna o controle do programa ao local em que o método foi chamado.

e) O comando _____ do depurador comporta-se como o comando `step` quando a próxima instrução a executar não contém uma chamada de método.

f) O comando `watch` do depurador permite visualizar todas as alterações em um(a) _____.

F.2 Determine se cada um dos seguintes itens é *verdadeiro* ou *falso*. Se *falso*, explique por quê.

a) Quando a execução do aplicativo é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução depois do ponto de interrupção.

b) Os comandos watch podem ser removidos com o comando `clear` do depurador.

c) A opção de compilador `-g` deve ser utilizada ao se compilar classes para depuração.

d) Quando um ponto de interrupção aparece em um loop, o depurador só faz uma pausa na primeira vez em que o ponto de interrupção é encontrado.

Respostas dos exercícios de autorrevisão

F.1 a) comentário. b) `print`. c) `set`. d) `step up`. e) `next`. f) campo.

F.2 a) Falso. Quando a execução do aplicativo é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução no ponto de interrupção. b) Falso. Os comandos watch podem ser removidos com o comando `unwatch` do depurador. c) Verdadeiro. d) Falso. Quando um ponto de interrupção aparece em um loop, o depurador faz uma pausa durante cada iteração.

Todas as notícias que merecem ser publicadas.
— **Adolph S. Ochs**

Que busca louca? Que luta para escapar?
— **John Keats**

Não remova o marco divisório dos campos.
— **Amenehope**

Saída formatada

Objetivos

Neste apêndice, você aprenderá:

- A entender os fluxos de entrada e saída.
- A utilizar a formatação `printf`.
- A imprimir com larguras e precisão de campo.
- A utilizar flags de formatação na string de formato `printf`.
- A imprimir com um índice de argumento.
- A gerar saída de literais e sequências de escape.
- A formatar a saída com a classe `Formatter`.

Sumário

G.1 Introdução
G.2 Fluxos
G.3 Formatando a saída com `printf`
G.4 Imprimindo inteiros
G.5 Imprimindo números de ponto flutuante
G.6 Imprimindo strings e caracteres
G.7 Imprimindo datas e horas
G.8 Outros caracteres de conversão
G.9 Imprimindo com larguras e precisões de campos
G.10 Utilizando flags na string de formato `printf`
G.11 Imprimindo com índices de argumento
G.12 Imprimindo literais e sequências de escape
G.13 Formatando saída com a classe `Formatter`
G.14 Conclusão

Resumo | Terminologia | Exercícios de autorrevisão | Respostas dos exercícios de autorrevisão | Exercícios

G.1 Introdução

Neste apêndice, discutimos os recursos de formatação do método `printf` e da classe `Formatter` (pacote `java.util`). A classe **`Formatter`** formata e gera a saída de dados para um destino especificado, como um fluxo de saída de string ou de arquivo. Muitos recursos do `printf` foram discutidos anteriormente neste livro. Este apêndice resume esses recursos e apresenta outros, como exibição de dados de data e hora em vários formatos, reordenamento de saída com base no índice do argumento e exibição de números e strings com vários flags.

G.2 Fluxos

Normalmente, a entrada e a saída são realizadas com fluxos, que são sequências de bytes. Nas operações de entrada, os bytes fluem de um dispositivo (por exemplo, teclado, unidade de disco, conexão de rede) para a memória principal. Nas operações de saída, os bytes fluem da memória principal para um dispositivo (por exemplo, monitor, impressora, unidade de disco, conexão de rede).

Quando a execução do programa inicia, três fluxos são criados. O fluxo de entrada padrão tipicamente lê bytes do teclado, e o stream de saída padrão tipicamente gera a saída de caracteres em uma janela de comando. Um terceiro fluxo, **fluxo de erro padrão** (`System.err`), costuma gerar a saída de caracteres em uma janela de comando e é utilizado para gerar saída a mensagens de erro para que elas possam ser visualizadas imediatamente. Em geral, os sistemas operacionais permitem que esses fluxos sejam redirecionados para outros dispositivos. Os fluxos são discutidos em detalhes no Capítulo 17, "Arquivos, fluxos e serialização de objetos", e no Capítulo 27, "Redes".

G.3 Formatando a saída com `printf`

Formatação de saída precisa é alcançada com `printf`. O Java SE 5 adotou (e aprimorou) esse recurso da linguagem de programação C. O método `printf` pode realizar as capacidades de formatação a seguir, cada uma das quais é discutida neste apêndice:

1. Arredondar valores de ponto flutuante para um número indicado de casas decimais.
2. Alinhar uma coluna de números com pontos de fração decimal aparecendo um em cima do outro.
3. Alinhamento à direita e alinhamento à esquerda das saídas.
4. Inserir caracteres literais em locais precisos em uma linha de saída.
5. Representar números de ponto flutuante em formato exponencial.
6. Representar inteiros em formato octal e hexadecimal.
7. Exibir todos os tipos de dados com campos de largura de tamanho fixo e precisão.
8. Exibir datas e horas em vários formatos.

Cada chamada a `printf` fornece como o primeiro argumento uma **string de formato** que descreve o formato de saída. A string de formato pode consistir em **texto fixo** e **especificadores de formato**. A saída de texto fixo é gerada por `printf` assim como seria gerada pelos métodos `System.out` `print` ou `println`. Cada especificador de formato é um marcador de lugar para um valor e especifica o tipo da saída de dados. Especificadores de formato também podem incluir informações opcionais de formatação.

Na forma mais simples, cada especificador de formato inicia com um sinal de porcentagem (`%`) e é seguido por um **caractere de conversão** que representa o tipo de dados do valor a ser impresso. Por exemplo, o especificador de formato `%s` é um marcador de lugar para uma `string` e o especificador de formato `%d` é um marcador de lugar para um valor `int`. As informações de formatação opcionais, como um índice de argumento, flags, precisão e largura de campo, são especificadas entre o sinal de porcentagem e o caractere de conversão. Demonstramos cada uma dessas capacidades.

G.4 Imprimindo inteiros

A Figura G.1 descreve os **caracteres de conversão integrais**. (Consulte no Apêndice H um resumo dos sistemas numéricos binários, octais, decimais e hexadecimais.) A Figura G.2 utiliza cada um para imprimir um número inteiro. Nas linhas 9–10, observe que o sinal de adição não é exibido por padrão, mas o sinal de subtração é. Mais adiante neste apêndice (Figura G.14), veremos como forçar a impressão de sinais de adição.

Caractere de conversão	Descrição
d	Exibe um inteiro decimal (base 10).
o	Exibe um inteiro octal (base 8).
x ou **X**	Exibe um número inteiro hexadecimal (base 16). `X` utiliza letras maiúsculas.

Figura G.1 | Caracteres de conversão de inteiros.

```
1  // Figura G.2: IntegerConversionTest.java
2  // Utilizando os caracteres de conversão de inteiros.
3
4  public class IntegerConversionTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%d\n", 26 );
9        System.out.printf("%d\n", +26 );
10       System.out.printf("%d\n", -26 );
11       System.out.printf("%o\n", 26 );
12       System.out.printf("%x\n", 26 );
13       System.out.printf("%X\n", 26 );
14    } // fim de main
15 } // fim da classe IntegerConversionTest
```

```
26
26
-26
32
1a
1A
```

Figura G.2 | Utilizando caracteres de conversão de inteiros.

O método `printf` tem a forma

```
printf( string-de-formato, lista-de-argumentos );
```

onde *string-de-formato* descreve o formato de saída e a *lista-de-argumentos* opcional contém os valores que correspondem a cada especificador de formato em *string-de-formato*. Há muitos especificadores de formato em uma string de formato.

Cada string de formato nas linhas 8–10 especifica que `printf` deve gerar saída de um inteiro decimal (`%d`) seguido por um caractere de nova linha. Na posição do especificador de formato, `printf` substitui o valor do primeiro argumento depois da string de formato. Se a string de formato contiver múltiplos especificadores de formato, em cada posição subsequente do especificador de formato, `printf` substituirá o valor do próximo argumento na lista de argumentos. O especificador de formato `%o` na linha 11 gera saída de inteiros no formato octal. O especificador de formato `%x` na linha 12 gera saída de inteiros no formato hexadecimal. O especificador de formato `%X` na linha 13 gera saída de inteiros no formato hexadecimal com letras maiúsculas.

G.5 Imprimindo números de ponto flutuante

A Figura G.3 descreve as conversões de pontos flutuantes. Os **caracteres de conversão** `e` e `E` exibem os valores de ponto flutuante em **notação científica computadorizada** (também chamada **notação exponencial**). A notação exponencial é o equivalente de computador à notação científica utilizada na matemática. Por exemplo, o valor `150,4582` é representado na notação científica matemática como

$$1{,}504582 \times 10^{2}$$

e é representado na notação exponencial como

```
1,504582e+02
```

no Java. Essa notação indica que `1,504582` é multiplicado por `10` elevado à segunda potência (`e+02`). O `e` significa "expoente".

A saída dos valores impressos com os caracteres de conversão `e`, `E` e `f` é gerada, por padrão, com seis dígitos de precisão à direita do ponto de fração decimal (por exemplo, `1,045921`) — outras precisões devem ser especificadas explicitamente. Para valores impressos com o caractere de conversão `g`, a precisão representa o número total de dígitos exibido sem o expoente. O padrão é seis dígitos (por exemplo, `12345678,9` é exibido como `1,23457e+07`). O **caractere de conversão `f`** sempre imprime pelo menos um dígito à esquerda do ponto de fração decimal. Os caracteres de conversão `e` e `E` imprimem o `e` em minúscula e o `E` em maiúscula antes do expoente e sempre imprime exatamente um dígito à esquerda do ponto de fração decimal. O arredondamento ocorre se o valor sendo formatado tiver mais dígitos significativos do que de precisão.

Caractere de conversão	Descrição
`e` ou `E`	Exibe um valor de ponto flutuante em notação exponencial. O caractere de conversão `E` exibe a saída em letras maiúsculas.
`f`	Exibe um valor de ponto flutuante no formato decimal.
`g` ou `G`	Exibe um valor de ponto flutuante no formato de ponto flutuante `f` ou no formato exponencial `e` com base na magnitude do valor. Se a magnitude for menor que 10^{-3}, maior ou igual a 10^{7}, o valor de ponto flutuante será impresso com `e` (ou `E`). Caso contrário, o valor é impresso no formato `f`. Quando o caractere de conversão `G` é utilizado, a saída é exibida em letras maiúsculas.
`a` ou `A`	Exibe um número de ponto flutuante no formato hexadecimal. O caractere de conversão `A` exibe a saída em letras maiúsculas.

Figura G.3 | Caracteres de conversão de ponto flutuante.

O **caractere de conversão `g`** (ou `G`) imprime o `e` (`E`) ou `f` em qualquer formato, dependendo do valor de ponto flutuante. Por exemplo, os valores `0,0000875`, `87500000,0`, `8,75`, `87,50` e `875,0` são impressos como `8,750000e-05`, `8,750000e+07`, `8,750000`, `87,500000` e `875,000000` com o caractere de conversão `g`. O valor `0,0000875` utiliza a notação `e` porque a magnitude é menor que 10^{-3}. O valor `87500000.0` usa a notação `e` porque a magnitude é maior do que 10^{7}. A Figura G.4 demonstra os caracteres de conversão de ponto flutuante.

```
1  // Figura G.4: FloatingNumberTest.java
2  // Utilizando caracteres de conversão de ponto flutuante.
3
4  public class FloatingNumberTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%e\n", 12345678.9 );
9        System.out.printf("%e\n", +12345678.9 );
10       System.out.printf("%e\n", -12345678.9 );
11       System.out.printf("%E\n", 12345678.9 );
12       System.out.printf("%f\n", 12345678.9 );
13       System.out.printf("%g\n", 12345678.9 );
14       System.out.printf("%G\n", 12345678.9 );
15    } // fim de main
16 } // fim da classe FloatingNumberTest
```

```
1.234568e+07
1.234568e+07
-1.234568e+07
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

Figura G.4 | Utilizando caracteres de conversão de ponto flutuante.

G.6 Imprimindo strings e caracteres

Os caracteres de conversão `c` e `s` imprimem caracteres e strings específicos, respectivamente. Os **caracteres de conversão `c` e `C`** requerem um argumento `char`. Os **caracteres de conversão `s` e `S`** podem receber uma `String` ou algum `Object` como um argumento. Quando os caracteres de conversão `C` e `S` são utilizados, a saída é exibida em letras maiúsculas. A Figura G.5 exibe caracteres, strings e objetos com os caracteres de conversão `c` e `s`. Observe que o autoboxing ("empacotamento") ocorre na linha 9 quando uma constante `int` é atribuída a um objeto `Integer`. A linha 15 gera saída de um argumento `Integer` com o caractere de conversão `s`, que invoca implicitamente o método `toString` para obter o valor de inteiro. Observe que você também pode gerar a saída de um objeto `Integer` utilizando o especificador de formato `%d`. Nesse caso, o valor `int` no objeto `Integer` será "desempacotado" (unboxing) e impresso.

Erro de programação comum G.1

Utilizar `%c` para imprimir uma `String` resulta em uma `IllegalFormatConversionException` — uma `String` não pode ser convertida em um caractere.

```
1  // Figura G.5: CharStringConversion.java
2  // Utilizando caractere e caracteres de conversão de string.
3  public class CharStringConversion
4  {
5     public static void main( String[] args )
6     {
7        char character = 'A';  // inicializa char
8        String string = "This is also a string";  // Objeto String
9        Integer integer = 1234;  // inicializa inteiro (autoboxing)
10
11       System.out.printf("%c\n", character );
12       System.out.printf("%s\n", "This is a string" );
13       System.out.printf("%s\n", string );
14       System.out.printf("%S\n", string );
15       System.out.printf("%s\n", integer ); // chamada implícita para toString
16    } // fim de main
17 } // fim da classe CharStringConversion
```

```
A
This is a string
This is also a string
THIS IS ALSO A STRING
1234
```

Figura G.5 | Utilizando caracteres de conversão de caractere e string.

G.7 Imprimindo datas e horas

O **caractere de conversão `t`** (ou `T`) é utilizado para imprimir datas e horas em vários formatos. Ele sempre é seguido por um **caractere de sufixo de conversão** que especifica que o formato de data e/ou hora. Quando o caractere de conversão `T` é utilizado, a saída é exibida em letras maiúsculas. A Figura G.6 lista os caracteres de sufixo de conversão comuns para formatar **composições de data e hora** que exibem tanto a data como a hora. A Figura G.7 lista os caracteres de sufixo de conversão comuns para formatar datas. A Figura G.8 lista os caracteres de sufixo de conversão comuns para formatar horas. Para a lista completa de caracteres de sufixo de conversão, visite `java.sun.com/javase/6/docs/api/java/util/Formatter.html`.

Caractere de sufixo de conversão	Descrição
c	Exibe a data e hora formatadas como `day month date hour:minute:second time-zone year` com três caracteres para `day` e `month`, dois dígitos para `date`, `hour`, `minute` e `second` e quatro dígitos para `year` — por exemplo, `Wed Mar 03 16:30:25 GMT-05:00 2004`. O relógio de 24 horas é utilizado. `GMT-05:00` é o fuso horário.

Caractere de sufixo de conversão	Descrição
F	Exibe a data formatada como `year-month-date` com quatro dígitos para o `year` e dois dígitos cada para `month` e a `date` (por exemplo, `2004-05-04`).
D	Exibe a data formatada como `month/day/year` com dois dígitos cada para o `month`, `day` e `year` (por exemplo, `03/03/04`).
r	Exibe a hora no formato de 12 horas como `hour:minute:second AM|PM` com dois dígitos cada para a `hour`, `minute` e `second` (por exemplo, `04:30:25 PM`).
R	Exibe a hora formatada como `hour:minute` com dois dígitos cada para a `hour` e `minute` (por exemplo, `16:30`). O relógio de 24 horas é utilizado.
T	Exibe a hora como `hour:minute:second` com dois dígitos para `hour`, `minute` e `second` (por exemplo, `16:30:25`). O relógio de 24 horas é utilizado.

Figura G.6 | Caracteres de sufixo de conversão de composição de data e hora.

Caractere de sufixo de conversão	Descrição
A	Exibe o nome completo do dia da semana (por exemplo, `Wednesday`).
a	Exibe o nome com três caracteres do dia da semana (por exemplo, `Wed`).
B	Exibe o nome completo do mês (por exemplo, `March`).
b	Exibe o nome abreviado com três caracteres do mês (por exemplo, `Mar`).
d	Exibe o dia do mês com dois dígitos, preenchendo com zeros à esquerda conforme necessário (por exemplo, `03`).
m	Exibe o mês com dois dígitos, preenchendo com zeros à esquerda conforme necessário (por exemplo, `07`).
e	Exibe o dia de mês sem zeros à esquerda (por exemplo, `3`).
Y	Exibe o ano com quatro dígitos (por exemplo, `2004`).
y	Exibe os dois últimos dígitos do ano com zeros à esquerda (por exemplo, `04`).
j	Exibe o dia do ano com três dígitos, preenchendo com zeros à esquerda conforme necessário (por exemplo, `016`).

Figura G.7 | Caracteres de sufixo de conversão de formatação de data.

Caractere de sufixo de conversão	Descrição
H	Exibe horas no relógio de 24 horas com um zero à esquerda conforme necessário (por exemplo, `16`).
I	Exibe horas no relógio de 12 horas com um zero à esquerda conforme necessário (por exemplo, `04`).
k	Exibe horas em relógio de 24 horas sem zeros à esquerda (por exemplo, `16`).
l	Exibe horas em relógio de 12 horas sem zeros à esquerda (por exemplo, `4`).
M	Exibe minutos com um zero à esquerda conforme necessário (por exemplo, `06`).
S	Exibe segundos com um zero à esquerda conforme necessário (por exemplo, `05`).
Z	Exibe a abreviação para o fuso horário (por exemplo, `EST`, significa Eastern Standard Time, que está 5 horas atrás do Greenwich Mean Time).
p	Exibe marcador de manhã ou de tarde em letras minúsculas (por exemplo, `pm`).
P	Exibe marcador de manhã ou de tarde em letras maiúsculas (por exemplo, `PM`).

Figura G.8 | Caracteres de sufixo de conversão de formatação de hora.

A Figura G.9 utiliza os caracteres de conversão `t` e `T` com os caracteres de sufixo de conversão para exibir datas e horas em vários formatos. O caractere de conversão `t` requer que o argumento correspondente seja uma data ou hora do tipo `long`, `Long`, **`Calendar`** (pacote `java.util`) ou **`Date`** (pacote `java.util`) — os objetos de cada uma dessas classes podem representar datas e horas. A classe `Calendar` é a preferida para esse propósito porque alguns construtores e métodos da classe `Date` são substituídos por aqueles da classe `Calendar`. A linha 10 invoca o método `static` **`getInstance`** da classe `Calendar` para obter um calendário com a data e hora atual. As linhas 13–17, 20–22 e 25–26 utilizam esse objeto de `Calendar` nas instruções `printf` como o valor a ser formatado com o caractere de conversão `t`. Observe que linhas 20–22 e 25–26 utilizam o **índice de argumentos** (`"1$"`) opcional para indicar que todos os especificadores de formato na string de formato utilizam o primeiro argumento depois da string de formato na lista de argumentos. Você aprenderá mais sobre índices de argumentos na Seção G.11. Utilizando o índice de argumentos elimina a necessidade de listar repetidamente o mesmo argumento.

```
1  // Figura G.9: DateTimeTest.java
2  // Formatando datas e horas com os caracteres de conversão t e T.
3  import java.util.Calendar;
4
5  public class DateTimeTest
6  {
7     public static void main( String[] args )
8     {
9        // obtém a data e hora atual
10       Calendar dateTime = Calendar.getInstance();
11
12       // imprimindo com caracteres de conversão para composições de data/hora
13       System.out.printf("%tc\n", dateTime );
14       System.out.printf("%tF\n", dateTime );
15       System.out.printf("%tD\n", dateTime );
16       System.out.printf("%tr\n", dateTime );
17       System.out.printf("%tT\n", dateTime );
18
19       // imprimindo com caracteres de conversão para data
20       System.out.printf("%1$tA, %1$tB %1$td, %1$tY\n", dateTime );
21       System.out.printf("%1$TA, %1$TB %1$Td, %1$TY\n", dateTime );
22       System.out.printf("%1$ta, %1$tb %1$te, %1$ty\n", dateTime );
23
24       // imprimindo com caracteres de conversão para hora
25       System.out.printf("%1$tH:%1$tM:%1$tS\n", dateTime );
26       System.out.printf("%1$tZ %1$tI:%1$tM:%1$tS %tP", dateTime );
27    } // fim de main
28 } // fim da classe DateTimeTest
```

```
Wed Feb 25 15:00:22 EST 2009
2009-02-25
02/25/09
03:00:22 PM
15:00:22
Wednesday, February 25, 2009
WEDNESDAY, FEBRUARY 25, 2009
Wed, Feb 25, 09
15:00:22
EST 03:00:22 PM
```

Figura G.9 | Formatando datas e horas com caracteres de conversão `t` e `T`.

G.8 Outros caracteres de conversão

Os caracteres de conversão restantes são **`b`**, **`B`**, **`h`**, **`H`**, **`%`** e **`n`**. Estes são discutidos na Figura G.10. As linhas 9–10 da Figura G.11 utilizam `%b` para imprimir o valor de valores `boolean` (ou `Boolean`) `false` e `true`. A linha 11 associa uma `String` a `%b`, que retorna `true` porque não é `null`. A linha 12 associa um objeto `null` a `%B`, que exibe `FALSE` porque `test` é `null`. As linhas 13–14 utilizam `%h` para imprimir as representações de string dos valores de código de hash para as strings `"hello"` e `"Hello"`. Esses valores poderiam ser utilizados para armazenar ou localizar as strings em uma `Hashtable` ou `HashMap` (ambas discutidas no Capítulo 20, "Coleções genéricas"). Observe que os valores de código de hash para essas duas strings diferem, pois uma string inicia com uma letra minúscula e a outra com uma letra

maiúscula. A linha 15 utiliza `%H` para imprimir `null` em letras maiúsculas. As duas últimas instruções `printf` (linhas 16–17) utilizam `%%` para imprimir o caractere `%` em uma string e `%n` para imprimir um separador de linha específico à plataforma.

Caractere de conversão	Descrição
`b` ou `B`	Imprime `"true"` ou `"false"` para o valor de um `boolean` ou `Boolean`. Esses caracteres de conversão também podem formatar o valor de qualquer referência. Se a referência for não `null`, `"true"` será impresso; do contrário, `"false"`. Quando o caractere de conversão `B` é utilizado, a saída é exibida em letras maiúsculas.
`h` ou `H`	Imprime a representação de string de um valor de código de hash do objeto em formato hexadecimal. Se o argumento correspondente for `null`, `"null"` será impresso. Quando o caractere de conversão `H` é utilizado, a saída é exibida em letras maiúsculas.
`%`	Imprime o caractere de porcentagem.
`n`	Imprime o separador de linha específico à plataforma (por exemplo, `\r\n` no Windows ou `\n` no UNIX/LINUX).

Figura G.10 | Outros caracteres de conversão.

```
1  // Figura G.11: OtherConversion.java
2  // Utilizando os caracteres de conversão b, B, h, % e n.
3
4  public class OtherConversion
5  {
6     public static void main( String[] args )
7     {
8        Object test = null;
9        System.out.printf( "%b \n", false );
10       System.out.printf( "%b \n", true );
11       System.out.printf( "%b \n", "Test" );
12       System.out.printf( "%B \n", test );
13       System.out.printf( "Hashcode of \"hello\" is %h\n", "hello" );
14       System.out.printf( "Hashcode of \"Hello\" is %h\n", "Hello" );
15       System.out.printf( "Hashcode of null is %H \n", test );
16       System.out.printf( "Printing a %% in a format string\n" );
17       System.out.printf( "Printing a new line %n next line starts here" );
18    } // fim de main
19 } // fim da classe OtherConversion
```

```
false
true
true
FALSE
Hashcode of "hello" is 5e918d2
Hashcode of "Hello" is 42628b2
Hashcode of null is NULL
Printing a % in a format string
Printing a new line
next line starts here
```

Figura G.11 | Utilizando os caracteres de conversão `b`, `B`, `h`, `H`, `%` e `n`.

Erro de programação comum G.2

Tentar imprimir um caractere de percentagem literal com % em vez de %% na string de formato resultaria em um erro de lógica difícil de detectar. Quando % aparece em uma string de formato, ele deve ser seguido por caractere de conversão na string. O caractere de percentagem individual poderia ser acidentalmente seguido por um caractere de conversão legítimo resultando em um erro de lógica.

G.9 Imprimindo com larguras e precisões de campos

O tamanho exato de um campo em que dados são impressos é especificado por uma **largura de campo**. Se a largura de campo for maior que os dados a serem impressos, os dados serão alinhados por padrão à direita no campo. Discutimos a justificação à esquerda na Seção G.10. Insira um inteiro que representa a largura de campo entre o `%` e o caractere de conversão (por exemplo, `%4d`) no especificador de formato. A Figura G.12 imprime dois grupos de cinco números cada, alinhando à direita os números que contém menos dígitos do que a largura de campo. Observe que a largura de campo é aumentada para imprimir valores com largura maior do que o campo e que o sinal de subtração para um valor negativo utiliza uma posição de caractere no campo. Além disso, se nenhuma largura de campo for especificada, os dados serão impressos exatamente de acordo com o número de posições necessárias. As larguras de campo podem ser usadas com todos os especificadores de formato, exceto o separador de linha (`%n`).

```
1  // Figura G.12: FieldWidthTest.java
2  // Alinhando inteiros à direita nos campos.
3
4  public class FieldWidthTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%4d\n", 1 );
9        System.out.printf("%4d\n", 12 );
10       System.out.printf("%4d\n", 123 );
11       System.out.printf("%4d\n", 1234 );
12       System.out.printf("%4d\n\n", 12345 ); // dados muito grandes
13
14       System.out.printf("%4d\n", -1 );
15       System.out.printf("%4d\n", -12 );
16       System.out.printf("%4d\n", -123 );
17       System.out.printf("%4d\n", -1234 ); // dados muito grandes
18       System.out.printf("%4d\n", -12345 ); // dados muito grandes
19    } // fim de main
20 } // fim da classe RightJustifyTest
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Figura G.12 | Alinhando inteiros à direita nos campos.

Erro de programação comum G.3

Não fornecer uma largura de campo suficientemente grande para tratar um valor a ser impresso pode deslocar outros dados que estão sendo impressos e produz saídas confusas. Conheça seus dados!

O método `printf` também fornece a capacidade de especificar a precisão com que os dados são impressos. A precisão tem diferentes significados para diferentes tipos. Quando usada com caracteres de conversão de ponto flutuante `e` e `f`, a precisão é o número de dígitos que aparece depois do ponto de fração decimal. Quando usada com os caracteres de conversão `g`, `a` ou `A`, a precisão é o número máximo de dígitos significativos a ser impresso. Quando usada com o caractere de conversão `s`, a precisão é o número máximo de caracteres a ser gravado da string. Para utilizar a precisão, coloque entre o sinal de porcentagem e o especificador de conversão um ponto de fração decimal (`.`) seguido por um inteiro que representa a precisão. A Figura G.13 demonstra o uso da precisão em strings de formato. Observe que quando um valor de ponto flutuante é impresso com uma precisão menor do que o número original de casas decimais no valor, o valor é arredondado. Também observe que o especificador de formato `%.3g` indica que o número total de dígitos usado para exibir o valor de ponto flutuante é 3. Como o valor tem três dígitos à esquerda do ponto de fração decimal, ele é arredondado para a casa das unidades.

A largura e a precisão de campo podem ser combinadas colocando a largura de campo, seguida por um ponto de fração decimal, seguido por uma precisão entre o sinal de porcentagem e o caractere de conversão, como na instrução

```
printf( "%9.3f", 123.456789 );
```

que exibe `123.457` com três dígitos à direita do ponto de fração decimal alinhado à direita em um campo de nove dígitos — esse número será precedido no campo por dois espaços em branco.

```
1  // Figura G.13: PrecisionTest.java
2  // Utilizando a precisão para números de ponto flutuante e strings.
3  public class PrecisionTest
4  {
5     public static void main( String[] args )
6     {
7        double f = 123.94536;
8        String s = "Happy Birthday";
9
10       System.out.printf( "Using precision for floating-point numbers\n" );
11       System.out.printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
12
13       System.out.printf( "Using precision for strings\n" );
14       System.out.printf( "\t%.11s\n", s );
15    } // fim de main
16 } // fim da classe PrecisionTest
```

```
Using precision for floating-point numbers
        123.945
        1.239e+02
        124

Using precision for strings
        Happy Birth
```

Figura G.13 | Utilizando a precisão para números de ponto flutuante e strings.

G.10 Utilizando flags na string de formato `printf`

Vários flags podem ser utilizados com o método `printf` para suplementar suas capacidades de formatação de saída. Sete flags estão disponíveis para utilização nas strings de formato (Figura G.14).

Flag	Descrição
`-` (sinal de subtração)	Alinha a saída à esquerda dentro do campo especificado.
`+` (sinal de adição)	Exibe um sinal de adição precedendo valores positivos e um sinal de subtração precedendo valores negativos.
espaço	Imprime um espaço antes de um valor positivo não impresso com o flag `+`.
`#`	O prefixo `0` para o valor de saída quando utilizado com o caractere de conversão octal `o`. Adiciona o prefixo `0x` ao valor de saída quando utilizado com o caractere de conversão hexadecimal `x`.
`0` (zero)	Preenche um campo com zeros à esquerda.
`,` (vírgula)	Utiliza o separador específico de localidade de milhares (isto é, `','` para localidade nos EUA) para exibir números decimais e de ponto flutuante.
`(`	Inclui números negativos dentro de parênteses.

Figura G.14 | Flags de string de formato.

Para utilizar um flag em uma string de formato, coloque-o imediatamente à direita do sinal de porcentagem. Vários flags podem ser utilizados no mesmo especificador de formato. A Figura G.15 demonstra o alinhamento à direita e o alinhamento à esquerda de uma string, um inteiro, um caractere e um número de ponto flutuante. Observe que a linha 9 serve como um mecanismo de contagem para a saída na tela.

```
1  // Figura G.15: MinusFlagTest java
2  // Valores para o alinhamento à direita e à esquerda.
3
4  public class MinusFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.println( "Columns:" );
9        System.out.println( "01234567890123456789012345678901234567890123456789\n" );
10       System.out.printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
11       System.out.printf(
12          "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
13    } // fim de main
14 } // fim da classe MinusFlagTest
```

```
Columns:
01234567890123456789012345678901234567890123456789

     hello         7         a  1.230000

hello     7         a         1.230000
```

Figura G.15 | Valores para o alinhamento à direita e à esquerda.

A Figura G.16 imprime um número positivo e um número negativo, cada um com e sem o **flag +**. Observe que o sinal de subtração é exibido nos dois casos, mas o sinal de adição somente quando o flag + é utilizado.

```
1  // Figura G.16: PlusFlagTest.java
2  // Imprimindo números com e sem o flag +.
3
4  public class PlusFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "%d\t%d\n", 786, -786 );
9        System.out.printf("%+d\t%+d\n", 786, -786 );
10    } // fim de main
11 } // fim da classe PlusFlagTest
```

```
786     -786
+786    -786
```

Figura G.16 | Imprimindo números com e sem o flag +.

A Figura G.17 prefixa um espaço para o número positivo com o **flag espaço**. Isso é útil para alinhar números positivos e negativos com o mesmo número de dígitos. Observe que o valor `-547` não é precedido por um espaço na saída por causa do seu sinal de subtração. A Figura G.18 utiliza o **flag #** para prefixar `0` para o valor octal e `0x` para o valor hexadecimal.

```
1  // Figura G.17: SpaceFlagTest.java
2  // Imprimindo um espaço antes de valores não negativos.
3
4  public class SpaceFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf( "% d\n% d\n", 547, -547 );
9     } // fim de main
10 } // fim da classe SpaceFlagTest
```

```
 547
-547
```

Figura G.17 | Imprimindo um espaço antes de valores não negativos.

```
1  // Figura G.18: PoundFlagTest.java
2  // Utilizando o flag # com os caracteres de conversão o e x.
3
4  public class PoundFlagTest
5  {
6     public static void main( String[] args )
7     {
8        int c = 31;      // inicializa c
9
10       System.out.printf( "%#o\n", c );
11       System.out.printf( "%#x\n", c );
12    } // fim de main
13 } // fim da classe PoundFlagTest
```

```
037
0x1f
```

Figura G.18 | Utilizando o flag # com caracteres de conversão o e x.

A Figura G.19 combina o flag +, o **flag 0** e o flag espaço para imprimir 452 em um campo de largura 9 com um sinal + e zeros à esquerda, em seguida imprime 452 em um campo de largura 9 utilizando somente o flag 0 e, então, imprime 452 em um campo de largura 9 utilizando somente o flag espaço.

```
1  // Figura G.19: ZeroFlagTest.java
2  // Imprimindo com flags 0 (zero) preenche com zeros à esquerda.
3
4  public class ZeroFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%+09d\n", 452 );
9        System.out.printf("%09d\n", 452 );
10       System.out.printf("% 9d\n", 452 );
11    } // fim de main
12 } // fim da classe ZeroFlagTest
```

```
+00000452
000000452
      452
```

Figura G.19 | Imprimindo com flags 0 (zero) preenche com zeros à esquerda.

A Figura G.20 utiliza o flag vírgula (,) para exibir um número decimal e um número de ponto flutuante com o separador de milhares. A Figura G.21 inclui números negativos entre parênteses utilizando o flag (. Observe que o valor 50 não é incluído entre parênteses na saída porque é um número positivo.

```
1  // Figura G.20: CommaFlagTest.java
2  // Utilizando o flag vírgula (,) para exibir números com separador de milhares.
3
4  public class CommaFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%,d\n", 58625 );
9        System.out.printf("%,.2f", 58625.21 );
10       System.out.printf("%,.2f", 12345678.9 );
11    } // fim de main
12 } // fim da classe CommaFlagTest
```

```
58,625
58,625.21
12,345,678.90
```

Figura G.20 | Utilizando o flag vírgula (,) para exibir números com o separador de milhares.

```
1  // Figura G.21: ParenthesesFlagTest.java
2  // Utilizando o flag ( para colocar parênteses em torno de números negativos.
3
4  public class ParenthesesFlagTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf("%(d\n", 50 );
9        System.out.printf("%(d\n", -50 );
10       System.out.printf("%(.1e\n", -50.0 );
11    } // fim de main
12 } // fim da classe ParenthesesFlagTest
```

```
50
(50)
(5.0e+01)
```

Figura G.21 | Utilizando o flag (para colocar parênteses em torno de números negativos.

G.11 Imprimindo com índices de argumento

Um **índice de argumentos** é um inteiro opcional seguido por um sinal $ que indica a posição do argumento na lista de argumentos. Por exemplo, as linhas 20–22 e 25–26 na Figura G.9 utilizam o índice de argumentos "1$" para indicar que todos os especificadores de formato utilizam o primeiro argumento da lista de argumentos. Os índices de argumentos permitem que os programadores reordenem a saída de modo que os argumentos na lista de argumentos não necessariamente estejam na ordem dos seus especificadores de formato correspondentes. Os índices de argumentos também ajudam a evitar a duplicação de argumentos. A Figura G.22 imprime os argumentos da lista de argumentos na ordem inversa utilizando o índice de argumentos.

```
1  // Figura G.22: ArgumentIndexTest
2  // Reordenando a saída com índices de argumentos.
3
4  public class ArgumentIndexTest
5  {
6     public static void main( String[] args )
7     {
8        System.out.printf(
9           "Parameter list without reordering: %s %s %s %s\n",
10          "first", "second", "third", "fourth" );
11       System.out.printf(
12          "Parameter list after reordering: %4$s %3$s %2$s %1$s\n",
13          "first", "second", "third", "fourth" );
14    } // fim de main
15 } // fim da classe ArgumentIndexTest
```

```
Parameter list without reordering: first second third fourth
Parameter list after reordering: fourth third second first
```

Figura G.22 | Reordenando a saída com índices de argumentos.

G.12 Imprimindo literais e sequências de escape

A maioria dos caracteres literais a ser impressa em uma instrução `printf` pode ser simplesmente incluída na string de formato. Entretanto, há vários caracteres "problemáticos", como aspas (") que delimitam a própria string de formato. Vários caracteres de controle, com a nova linha e tabulação, devem ser representados por sequências de escape. Uma sequência de escape é representada por uma barra invertida (\), seguida por um caractere de escape. A Figura G.23 lista as sequências de escape e as ações resultantes.

Erro de programação comum G.4

Tentar imprimir como dados literais em uma instrução `printf` um caractere de aspa dupla ou de barra invertida sem preceder esse caractere com uma barra invertida para formar uma sequência de escape adequada resultaria em um erro de sintaxe.

Sequência de escape	Descrição
`\'` (aspa simples)	Gera saída do caractere de aspa simples (').
`\"` (aspas duplas)	Gera a saída do caractere de aspas duplas (").
`\\` (barras invertidas)	Gera a saída do Caractere de barra invertida (\).
`\b` (backspace)	Move o cursor de volta uma posição na linha atual.
`\f` (nova página ou avanço de formulário)	Move o cursor para o início da próxima página lógica.
`\n` (nova linha)	Move o cursor para o começo da próxima linha.
`\r` (retorno de carro)	Move o cursor para o começo da linha atual.
`\t` (tabulação horizontal)	Move o cursor para a próxima posição da tabulação horizontal.

Figura G.23 | Sequências de escape.

G.13 Formatando saída com a classe `Formatter`

Até agora, discutimos a exibição de saída formatada para o fluxo de saída padrão. O que devemos fazer se quisermos enviar saídas formatadas a outros fluxos de saída ou dispositivos, como um `JTextArea` ou um arquivo? A solução conta com a classe `Formatter` (do pacote `Formatter`), que fornece as mesmas capacidades de formatação de `printf`. `Formatter` é uma classe utilitária que permite aos programadores gerar saída formatada de dados para um destino especificado, como um arquivo em disco. Por padrão, uma `Formatter` cria uma string na memória. A Figura G.24 mostra como utilizar uma classe `Formatter` para construir uma string formatada, que é então exibida em uma caixa de diálogo de mensagem.

```
1  // Figura G.24: FormatterTest.java
2  // Formatando a saída com a classe Formatter.
3  import java.util.Formatter;
4  import javax.swing.JOptionPane;
5
6  public class FormatterTest
7  {
8     public static void main( String[] args )
9     {
10       // cria Formatter e formata a saída
11       Formatter formatter = new Formatter();
12       formatter.format( "%d = %#o = %#X", 10, 10, 10 );
13
14       // exibe a saída em JOptionPane
15       JOptionPane.showMessageDialog( null, formatter.toString() );
16    } // fim de main
17 } // fim da classe FormatterTest
```

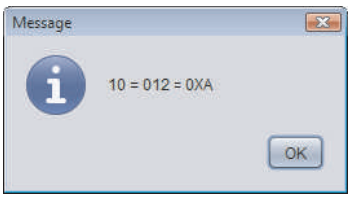


Figura G.24 | Formatando saída com a classe `Formatter`.

A linha 11 cria um objeto `Formatter` utilizando o construtor padrão, assim esse objeto construirá uma string na memória. Outros construtores são fornecidos para que você possa especificar o destino em que os dados formatados devem ser impressos. Para detalhes, consulte `java.sun.com/javase/6/docs/api/java/util/Formatter.html`.

A linha 12 invoca o método **`format`** para formatar a saída. Como ocorre com `printf`, o método `format` recebe uma string de formato e uma lista de argumentos. A diferença é que `printf` envia a saída formatada diretamente para o fluxo de saída padrão, enquanto o método `format` envia a saída formatada para o destino especificado pelo seu construtor (nesse programa, uma string na memória). A linha 15 invoca o método `toString` de `Formatter` para obter os dados formatados como uma string, que então é exibida em uma caixa de diálogo de mensagem.

Observe que a classe `String` também fornece um método `static` de conveniência chamado `format` que permite criar uma string na memória sem precisar antes criar um objeto `Formatter`. As linhas 11–12 e a linha 15 na Figura G.24 poderiam ter sido substituídas por

```
String s = String.format( "%d = %#o = %#x", 10, 10, 10 );
JOptionPane.showMessageDialog( null, s );
```

G.14 Conclusão

Este apêndice resumiu como exibir saída formatada com vários caracteres e flags de formato. Exibimos números decimais utilizando os caracteres de formato `d`, `o`, `x` e `X`; números de ponto flutuante utilizando os caracteres de formato `e`, `E`, `f`, `g` e `G`; e as datas e horas em vários formatos utilizando os caracteres de formato `t` e `T` e seus caracteres de sufixo de conversão. Você aprendeu a exibir saída com larguras e precisão de campo. Introduzimos os flags `+`, `-`, espaço, `#`, `0`, vírgula e `(` utilizados juntamente com os caracteres de formato para produzir a saída. Também demonstramos como formatar a saída com a classe `Formatter`.

Resumo

Seção G.2 Fluxos

- Normalmente, a entrada e saída são realizadas com fluxos, que são sequências de bytes.
- Normalmente, o fluxo de entrada padrão está conectado ao teclado e o fluxo de saída padrão está conectado à tela do computador.

Seção G.3 Formatando a saída com `printf`

- A string de formato de `printf` descreve os formatos em que os valores de saída aparecem. O especificador de formato consiste em índice de argumentos, flags, larguras de campo, precisões e caracteres de conversão.

Seção G.4 Imprimindo números inteiros

- Os inteiros são impressos com os caracteres de conversão `d` para inteiros decimais, `o` para inteiros na forma octal e `x` (ou `X`) para inteiros na forma hexadecimal. `X` exibe letras maiúsculas.

Seção G.5 Imprimindo números de ponto flutuante

- Valores de ponto flutuante são impressos com os caracteres de conversão `e` (ou `E`) para notação exponencial, `f` para notação de ponto flutuante regular e `g` (ou `G`) para notação `e` (ou `E`) ou notação `f`. Para o especificador de conversão `g`, o caractere de conversão `e` será utilizado se o valor for menor que 10^{-3}, maior ou igual a 107; caso contrário, o caractere de conversão `f` é utilizado.

Seção G.6 Imprimindo strings e caracteres

- O caractere de conversão `c` imprime um caractere.
- O caractere de conversão `s` (ou `S`) imprime uma string de caracteres. O caractere de conversão `S` exibe a saída em letras maiúsculas.

Seção G.7 Imprimindo datas e horas

- O caractere de conversão `t` (ou `T`) seguido por um caractere de sufixo de conversão imprime a data e hora em várias formas. O caractere de conversão `T` exibe a saída em letras maiúsculas.
- O caractere de conversão `t` (ou `T`) requer que o argumento seja do tipo `long`, `Long`, `Calendar` ou `Date`.

Seção G.8 Outros caracteres de conversão

- O caractere de conversão `b` (ou `B`) gera saída da representação de string de um `boolean` ou `Boolean`. Esses caracteres de conversão também imprimem `"true"` para referências não `null` e `"false"` para referências `null`. O caractere de conversão `B` imprime em letras maiúsculas.
- O caractere de conversão `h` (ou `H`) retorna `null` a uma referência `null` e uma representação de `String` do valor de código de hash (na base 16) de um objeto. Códigos de hash são utilizados para armazenar e recuperar objetos em `Hashtables` e `HashMaps`. O caractere de conversão `H` imprime em letras maiúsculas.
- O caractere de conversão `n` imprime o separador de linha específico à plataforma.
- O caractere de conversão `%` é utilizado para exibir um literal `%`.

Seção G.9 Imprimindo utilizando precisão e larguras de campo

- Se a largura de campo for maior que o objeto que está sendo impresso, o objeto será alinhado à direita no campo.
- As larguras de campo podem ser utilizadas com todos os caracteres de conversão, exceto a conversão de separador de linha.
- A precisão utilizada com os caracteres de conversão de ponto flutuante `e` e `f` indica o número de dígitos que aparece depois do ponto de fração decimal. A precisão utilizada com o caractere de conversão de ponto flutuante `g` indica o número de dígitos significativos que aparece.
- A precisão utilizada com o caractere de conversão `s` indica o número de caracteres a ser impresso.
- A largura e precisão de campo podem ser combinadas colocando-se a largura de campo, seguida por um ponto de fração decimal, seguida pela precisão entre o sinal de porcentagem e o caractere de conversão.

Seção G.10 Utilizando flags na string de formato `printf`

- O flag `-` é alinhado à esquerda do seu argumento em um campo.
- O flag `+` imprime um sinal de adição para valores positivos e um sinal de subtração para valores negativos.
- O flag espaço imprime um espaço que precede um valor positivo. O flag espaço e o flag `+` não podem ser utilizados juntos em um caractere de conversão de inteiros.
- O `flag #` prefixa `0` para os valores octais e `0x` para valores hexadecimais.
- O flag `0` imprime zeros à esquerda para um valor que não ocupa todo o seu campo.
- O flag de vírgula `(,)` utiliza o separador de milhares específicos à localidade para exibir números inteiros e números de ponto flutuante.
- O flag `(` coloca um número negativo dentro de parênteses.

Seção G.11 Imprimindo com índices de argumento

- Um **índice de argumentos** é um inteiro decimal opcional seguido por um sinal `$` que indica a posição do argumento na lista de argumentos.
- Os índices de argumentos permitem que os programadores reordenem a saída de modo que os argumentos na lista de argumentos não necessariamente estejam na ordem dos seus especificadores de formato correspondentes. Os índices de argumentos também ajudam a evitar a duplicação de argumentos.

Seção G.11 Formatando a saída com a classe `Formatter`

- A classe `Formatter` (no pacote `java.util`) fornece as mesmas capacidades de formatação do `printf`. `Formatter` é uma classe utilitária que permite aos programadores imprimir saída formatada para vários destinos, incluindo componentes GUI, arquivos e outros fluxos de saída.
- O método `Formatter format` gera saída formatada de dados para o destino especificado pelo construtor de `Formatter`.
- O método `static format String` formata dados e retorna os dados formatados como uma `String`.

Terminologia

-, flag, 1099
, (vírgula), flag, 1099
(, flag, 1099
#, flag, 1099
%, caractere de conversão, 1096
%f, especificador de formato, 1093
%s, especificador de formato, 1094
+, flag, 1099
0 (zero), flag, 1099
a, caractere de sufixo de conversão, 1095
A, caractere de sufixo de conversão, 1095
b, caractere de conversão, 1096
B, caractere de conversão, 1096
b, caractere de sufixo de conversão, 1095
B, caractere de sufixo de conversão, 1095
c, caractere de conversão, 1094
C, caractere de conversão, 1094
c, caractere de sufixo de conversão, 1094
Calendar, classe, 1096
caractere de conversão, 1091
caractere de sufixo de conversão, 1094

caracteres de conversão de inteiros, 1092
composições de data e hora, 1094
d, caractere de conversão, 1092
d, caractere de sufixo de conversão, 1095
D, caractere de sufixo de conversão, 1095
Date, classe, 1096
e, caractere de sufixo de conversão, 1095
espaço, flag, 1099
especificadores de formato, 1091
f, caractere de conversão, 1093
F, caractere de sufixo de conversão, 1095
fluxo de erro padrão, 1091
format, método da classe Formatter, 1104
formato, string, 1091
Formatter, classe, 1091
g, caractere de conversão, 1093
G, caractere de conversão, 1093
getInstance, método de Calendar, 1096
h, caractere de conversão, 1096
H, caractere de conversão, 1096
H, caractere de sufixo de conversão, 1095
I, caractere de sufixo de conversão, 1095
índice de argumentos, 1096
j, caractere de sufixo de conversão, 1095
k, caractere de sufixo de conversão, 1095
l, caractere de sufixo de conversão, 1095
m, caractere de sufixo de conversão, 1095
M, caractere de sufixo de conversão, 1095
n, caractere de conversão, 1096
notação científica computadorizada, 1092
notação exponencial, 1092
o, caractere de conversão, 1092
p, caractere de sufixo de conversão, 1095
P, caractere de sufixo de conversão, 1095
r, caractere de sufixo de conversão, 1095
R, caractere de sufixo de conversão, 1095
s, caractere de conversão, 1094
S, caractere de conversão, 1094
S, caractere de sufixo de conversão, 1095
t, caractere de conversão, 1094
T, caractere de conversão, 1094
T, caractere de sufixo de conversão, 1095
texto fixo, 1091
x, caractere de conversão, 1092
X, caractere de conversão, 1092
y, caractere de sufixo de conversão, 1095
Y, caractere de sufixo de conversão, 1095
Z, caractere de sufixo de conversão, 1095

Exercícios de autorrevisão

G.1 Preencha as lacunas em cada uma das alternativas ou em cada um dos itens:

a) Lidamos com todas as entradas e saídas na forma de _____.

b) O fluxo _____ normalmente está conectado ao teclado.

c) O fluxo _____ normalmente está conectado à tela do computador.

d) O método _____ de `System.out` pode ser utilizado para formatar texto que é exibido na saída padrão.

e) O caractere de conversão _____ pode ser utilizado para gerar saída de um inteiro decimal.

f) Os caracteres de conversão _____ e _____ são utilizados para exibir inteiros na forma octal e hexadecimal, respectivamente.

g) O caractere de conversão _____ é utilizado para exibir um valor de ponto flutuante na notação exponencial.

h) Os caracteres de conversão `e` e `f` são exibidos com os dígitos _____ de precisão à direita do ponto de fração decimal se nenhuma precisão for especificada.

i) Os caracteres de conversão _____ e _____ são utilizados para imprimir strings e caracteres, respectivamente.

j) O caractere de conversão _____ e o caractere de sufixo de conversão _____ são utilizados para imprimir tempo no relógio de 24 horas como `hour:minute:second`.

k) O flag _____ faz com que a saída seja alinhada à esquerda em um campo.

l) O flag _____ faz com que os valores sejam exibidos com um sinal de adição ou um sinal de subtração.

m) O índice de argumentos _____ corresponde ao segundo argumento na lista de argumentos.

n) A classe _____ tem a mesma capacidade de `printf`, mas permite que programadores imprimam a saída formatada para vários destinos além do fluxo de saída padrão.

G.2 Localize o erro em cada um dos seguintes itens e explique como eles podem ser corrigidos.

a) A instrução a seguir deve imprimir o caractere `'c'`.

```
System.out.printf( "%c\n", "c" );
```

b) A instrução a seguir deve imprimir `9,375%`.

```
System.out.printf( "%.3f%", 9.375 );
```

c) A instrução a seguir deve imprimir o terceiro argumento na lista de argumentos.

```
System.out.printf( "%2$s\n", "Mon", "Tue", "Wed", "Thu", "Fri" );
```

d)
```
System.out.printf( ""A string in quotes"" );
```

e)
```
System.out.printf( %d%d, 12, 20 );
```

f)
```
System.out.printf( "%s\n", 'Richard' );
```

G.3 Escreva uma instrução para cada um dos seguintes tópicos:

a) Imprima `1234` alinhado à direita em um campo de 10 dígitos.

b) Imprima `123,456789` na notação exponencial com um sinal (+ ou -) e 3 dígitos de precisão.

c) Imprima `100` na forma octal precedida por `0`.

d) Dado um objeto `calendar` de `Calendar`, imprima uma data formatada como mês/dia/ano (cada com dois dígitos).

e) Dado um objeto `calendar` de `Calendar`, imprima uma hora no relógio de 24 horas como `hour:minute:second` (cada com dois dígitos) utilizando o índice de argumentos e caracteres de sufixo de conversão para formatar a hora.

f) Imprima `3,333333` com um sinal (+ ou -) em um campo de 20 caracteres com uma precisão de 3.

Respostas dos exercícios de autorrevisão

G.1 a) Fluxos. b) Entrada padrão. c) Saída padrão. d) `printf`. e) `d`. f) `o`, `x` ou `X`. g) `e` ou `E`. h) 6. i) `s` ou `S`, `c` ou `C`. j) `t`, `T`. k) - (sinal de subtração). l) + (sinal de adição). m) `US$ 2.n)` `Formatter`.

G.2 a) Erro: o caractere de conversão `c` espera um argumento do tipo primitivo `char`.
Correção: para imprimir o caractere `'c'`, altere `"c"` para `'c'`.

b) Erro: tentar imprimir o caractere literal `%` sem utilizar o especificador de formato `%%`.
Correção: utilize `%%` para imprimir um caractere `%` literal.

c) Erro: o índice de argumentos não inicia em 0; por exemplo, o primeiro argumento é `1$`.
Correção: para imprimir o terceiro utilize o argumento `3$`.

d) Erro: tentar imprimir o caractere literal " sem utilizar a sequência de escape `\"`.
Correção: substitua cada citação no conjunto interno de aspas por `\"`.

e) Erro: a string de formato não é incluída entre aspas duplas.
Correção: inclua `%d %d` entre aspas duplas.

f) Erro: a string a ser impressa é incluída entre aspas simples.
Correção: utilize aspas duplas em vez de aspas simples para representar uma string.

G.3 a)
```
System.out.printf( "%10d\n", 1234 );
```
b)
```
System.out.printf( "%+.3e\n", 123.456789 );
```
c)
```
System.out.printf( "%#o\n", 100 );
```
d)
```
System.out.printf( "%tD\n", calendar );
```
e)
```
System.out.printf( "%1$tH:%1$tM:%1$tS\n", calendar );
```
f)
```
System.out.printf( "%+20.3f\n", 3.333333 );
```

Exercícios

G.4 Escreva a(s) instrução(ões) para cada um dos seguintes itens:

a) Imprima o inteiro `40000` alinhado à direita em um campo de 15 dígitos.

b) Imprima `200` Com e sem sinal.

c) Imprima `100` na forma hexadecimal precedido por `0x`.

d) Imprima `1,234` com três dígitos de precisão em um campo de 9 dígitos com zeros no início.

G.5 Mostre o que é impresso por cada uma das instruções a seguir. Se uma instrução estiver incorreta, indique por quê.

a)
```
System.out.printf( "%-10d\n", 10000 );
```
b)
```
System.out.printf( "%c\n", "This is a string" );
```
c)
```
System.out.printf( "%8.3f\n", 1024.987654 );
```
d)
```
System.out.printf( "%#o\n%#X\n", 17, 17 );
```
e)
```
System.out.printf( "% d\n%+d\n", 1000000, 1000000 );
```
f)
```
System.out.printf( "%10.2e\n", 444.93738 );
```
g)
```
System.out.printf( "%d\n", 10.987 );
```

G.6 Encontre o(s) erro(s) em cada um dos segmentos de programa a seguir. Mostre a instrução corrigida.

a)
```
System.out.printf( "%s\n", 'Happy Birthday' );
```
b)
```
System.out.printf( "%c\n",'Hello' );
```
c)
```
System.out.printf( "%c\n", "This is a string" );
```
d) A seguinte instrução deve imprimir `"Bon Voyage"` com aspas duplas:
```
System.out.printf( ""%s"", "Bon Voyage" );
```
e) A seguinte instrução imprime "Today is Friday":
```
System.out.printf( "Today is %s\n", "Monday", "Friday" );
```
f)
```
System.out.printf( 'Enter your name: ' );
```
g)
```
System.out.printf( %f, 123.456 );
```
h) A instrução a seguir deve imprimir a hora atual no formato `"hh:mm:ss"`:
```
Calendar dateTime = Calendar.getInstance();
System.out.printf( "%1$tk:1$%tl:%1$tS\n", dateTime );
```

G.7 (***Imprimindo datas e horas***) Escreva um programa que imprime datas e horas nos formatos a seguir:
```
GMT-05:00 04/30/04 09:55:09 AM
GMT-05:00 April 30 2004 09:55:09
```

```
2004-04-30 day-of-the-month:30
2004-04-30 day-of-the-year:121
Fri Apr 30 09:55:09 GMT-05:00 2004
```

[*Nota*: dependendo da sua localização, talvez haja outros fusos horários além do GMT-05:00.]

G.8 Escreva um programa para testar os resultados da impressão do valor inteiro `12345` e do valor de ponto flutuante `1,2345` em vários campos de vários tamanhos.

G.9 **(*Arredondando números*)** Escreva um programa que imprime o valor `100,453627` arredondado para o dígito mais próximo, décimo, centésimo, milésimo e décimo de milésimo.

G.10 Escreva um programa que gera a saída de uma palavra digitada partir do teclado e determina seu comprimento. Imprima a palavra utilizando um tamanho duas vezes maior que o comprimento da largura de campo.

G.11 **(*Convertendo temperaturas Fahrenheit em Celsius*)** Escreva um programa que converte valores inteiros de temperatura em Fahrenheit entre `0` e `212` graus para temperaturas em graus Celsius de pontos flutuantes com três dígitos de precisão. Utilize a fórmula

```
celsius = 5.0 / 9.0 * ( fahrenheit - 32 );
```

para realizar o cálculo. A saída deve ser impressa em duas colunas alinhadas à direita, cada coluna deve conter 10 caracteres cada e as temperaturas em graus Celsius devem ser precedidas por um sinal para valores positivos e negativos.

G.12 Escreva um programa para testar todas as sequências de escape na Figura G.23. Para as sequências de escape que movem o cursor, imprima um caractere antes e depois da sequência de escape de modo que fique claro o local para onde o cursor se moveu.

G.13 Escreva um programa que utiliza o caractere de conversão `g` para gerar a saída do valor `9876,12345`. Imprima o valor com precisões no intervalo entre 1 e 9.

Apêndices na Web

Os apêndices a seguir estão disponíveis em `www.prenhall.com/deitel_br` como documentos PDF:

- Apêndice H, "Sistemas de numeração"
- Apêndice I, "`GroupLayout`"
- Apêndice J, "Java Desktop Integration Components (JDIC)"
- Apêndice K, "Mash ups"
- Apêndice L, "Unicode®"
- Apêndice M, "Criando documentação com `javadoc`"
- Apêndice N, "Manipulação de bits"
- Apêndice O, "Instruções `break` e `continue` rotuladas"
- Apêndice P, "UML 2: Tipos de diagramas adicionais"
- Apêndice Q, "Padrões de design"

Esses arquivos podem ser visualizados no programa Adobe® Reader® (`get.adobe.com/reader`).

Se você ainda não estiver registrado em seu site Web, dirija-se a `www.deitel.com` e clique no link **Register-** abaixo de nosso logotipo no canto superior esquerdo da página. Preencha suas informações. Não há nenhum custo para se registrar e não compartilhamos nossas informações com ninguém. Enviaremos somente correios eletrônicos de gerenciamento de conta a não ser que você se registre separadamente em nosso boletim por e-mail gratuito *Deitel® Buzz Online* em `www.deitel.com/newsletter-/subscribe.html`. Depois de se registrar, você receberá um correio eletrônico de confirmação com o seu código de verificação. Você precisará desse código para fazer a assinatura em `www.deitel.com` pela primeira vez. Configure seu cliente de correio eletrônico para permitir e-mails da `deitel.com`, de modo a assegurar que o e-mail de confirmação não seja filtrado com mensagem maliciosa.

Em seguida, dirija-se a `www.deitel.com` e faça a assinatura utilizando o link **Login** debaixo de nosso logotipo no canto superior esquerdo da página. Dirija-se a `www.deitel.com/books/jhtp8/`. Você encontrará os links para esses apêndices sob o título **Download Code Examples and Other Premium Content for Registered Users** na coluna esquerda da página.

Índice remissivo

Símbolos

^=, operador de atribuição OU exclusivo sobre bits, LI
^, OU exclusivo sobre bits, **XLIII**
^, OU lógico booleano exclusivo, operador, 136, **138**
  tabela-verdade, 138
_, caractere curinga de SQL, **904**, 905
-, (sinal de subtração), flag de formatação, **126**
-, flag, **1099**
-=, operador de atribuição de subtração, 102
,, (vírgula), flag de formatação, **127**
!, NÃO lógico, operador, 136, **138**
  tabela-verdade, 138
!=, não igual a, 43
? (argumento de tipo curinga), **688**
?:, operador condicional ternário, **86**, 104
., ponto separador, **60**
, (vírgula), flag, **1099**
", aspa dupla, sequência de escape, 35
(, flag, **1099**
{, chave esquerda, 31
}, chave direita, 31
@, símbolo, **XXXIV**
*, caractere curinga de SQL, **903**
*, curinga em um nome de arquivo, 60
*, multiplicação, 41, 42
*=, operador de atribuição de multiplicação, 102
/, barras em tags de fechamento, **735**
/** */, comentário de documentação Java, **XXXIV**
//** */, comentário de documentação Java, 30
/* */, comentário tradicional, **29**
/=, operador de atribuição de divisão, 102
//, comentário de fim de linha, **29**
\", 1103
\', sequência de escape de caractere de aspas simples, 1103
\b, sequência de escape, 1103
\f, avanço de formulário, sequência de escape, 1103
\n, nova linha, sequência de escape, 35, 1103
\r, retorno de carro, sequência de escape, 35, 1103
\t, tabulação horizontal, sequência de escape, 35
\\, caractere de barra invertida, sequência de escape, 1103
&&, E condicional, operador, **136**
  tabela-verdade, 136
&, E lógico booleano, operador, 136, **137**
&, E sobre bits, **XLIII**
&=, operador de atribuição E sobre bits, LI
#, caractere, **XXXIX**
#, flag, **1099**, 1100
%, módulo, 41, 42
%, caractere de conversão, **1096**
%a, especificador de formato, 1093
%A, especificador de formato, 1093
%b, especificador de formato, **138**, 1096, 1097
%B, especificador de formato, 1096
%c, especificador de formato, 54, 1094
%C, especificador de formato, 1094
%, caractere curinga de SQL, **904**
%d, especificador de formato, **39**, 1091, 1092
%e, especificador de formato, 1092
%E, especificador de formato, 1092
%%, especificador de formato, 1097
%f, especificador de formato, 53, **71**, **1093**
%g, especificador de formato, 1093
%G, especificador de formato, 1093
%h, especificador de formato, 1097
%H, especificador de formato, 1096
%n, especificador de formato, 1097
%n, formatar o especificador (separador de linha), **562**
%o, especificador de formato, 1092
%=, operador de atribuição de resto, 102
%s, especificador de formato, **36**, 1091, **1094**
%S, especificador de formato, 1094
%t, especificador de formato, 1094
%T, especificador de formato, 1094
%x, especificador de formato, 1092
%X, especificador de formato, 1092
+, adição, 41, 42
+=, operador de atribuição de adição, 131
+, flag, **1099**, 1100
+=, operador de atribuição de adição, **102**
+=, operador de atribuição de concatenação de strings, 527
+=, operador de atribuição de subtração, 102
++, pré-incremento/pós-incremento, **102**
<, operador ("maior que"), **43**
<>, colchetes angulares para elementos XML, **735**
<, menor que, 43
<<=, operador de atribuição de deslocamento para a esquerda., LI
<<, deslocamento de bits para a esquerda, **XLIII**, XLIV
<=, menor que ou igual a, 43
==, é igual a, 43
=, operador de atribuição, 39
==, para determinar se duas referências referenciam o mesmo objeto, 298
>>, deslocamento para a direita com sinal, **XLIII**, XLIV
>>>, deslocamento para a direita sem sinal, **XLIII**, XLIX
>, maior que, 43
>=, maior que ou igual a, 43
>>=, operador de atribuição de deslocamento para a direita com sinal, LI
>>>=, operador de atribuição de deslocamento para a direita sem sinal, LI
||, OU condicional, operador, 136, **137**
  tabela-verdade, 137
|, OU inclusivo lógico booleano, operador, 136, **137**
|=, operador de atribuição OU inclusivo sobre bits, LI
|, OU inclusivo sobre bits, **XLIII**
~, (complemento de bits), **XLIII**

Números

0, flag, **196**
0, formato, flag, 244
0x, (prefixo hexadecimal), 1100
0 (zero), flag, **1099**, 1101
15, jogo do, exercício, 766
127.0.0.1, (endereço de IP localhost), **871**, 948

A

abordagem de blocos de construção para criar programas, 7, 16
abordagem de código ativo, **2**
abordagem de dividir para conquistar, 591
abordagem iterativa (não recursiva), 591

abreviações em inglês, 5
abreviando expressões de atribuição, 102
abrir um arquivo, **554**
abs, método de Math, 157
absolute, método de ResultSet, **919**
abstract, palavra-chave, **309**
AbstractButton, classe, **435**, 438, 777
  addActionListener, método, 438
  addItemListener, método, **440**
  isSelected, método, **779**
  setMnemonic, método, **777**
  setRolloverIcon, método, **438**
  setSelected, método, **778**
AbstractCollection, classe, **664**
Abstract Factory, padrão de design, LIX, LX, LXVII
AbstractList, classe, **664**
AbstractMap, classe, **664**
AbstractPageBean, classe, **954**, 958, 1050
AbstractQueue, classe, **664**
AbstractSequentialList, classe, **664**
AbstractSessionBean, classe, **979**
AbstractSet, classe, **664**
AbstractTableModel, classe, **915**, 919
  fireTableStructureChanged, método, **919**
Abstract Window Toolkit (AWT), **423**
  pacote, **163**
Abstract Window Toolkit Event Package, **163**
ação, 85, 88, **951**
ação a executar, **82**
ação de um objeto, **375**
accept, método da classe ServerSocket, **866**, 871
Account, classe (estudo de caso ATM), 365, 367, 370, 372, 377, 382, 383, 384, 385, 386, 406
acessibilidade, 425
acesso a pacotes, **267**
acesso a pacotes, métodos, 267
acesso simultâneo à Collection por múltiplas threads, 663
acoplamento, **LXV**
ActionEvent, classe, **432**, 435, 470, 751
  getActionCommand, método, **433**, 438
ActionListener, interface, **432**, 435
  actionPerformed, método, 432, 467, 470
actionPerformed, método da interface ActionListener, 432, 467, 470
ACTIVATED, constante da classe aninhada EventType, **865**
Ada, linguagem de programação, **8**, 804
Ada Lovelace, 8
Adapter, padrão de design, LIX, LXIII
addActionListener, método
  da classe AbstractButton, 438
  da classe JTextField, **432**
addAll, método
  Collections, 645, **652**
  List, **643**
add, método
  ArrayList<T>, **221**, 862
  BigInteger, **595**
  ButtonGroup, **442**
  JFrame, 300, **427**
  JFrame, classe, **107**
  JMenu, **777**
  JMenuBar, **778**
  LinkedList, **644**
  List, **641**, 643
addFirst, método da classe LinkedList, **644**
addGap, método da classe GroupLayout.Group, **XIII**
addGap, método da classe GroupLayout.ParallelGroup, **XIII**
addGap, método da classe GroupLayout.SequentialGroup, **XIII**
addItemListener, método da classe AbstractButton, **440**
addKeyListener, método da classe Component, **459**
addLast, método da classe LinkedList, **644**
addListSelectionListener, método da classe JList, **447**
addMouseListener, método da classe Component, **452**
addMouseMotionListener, método da classe Component, **452**
Add Binding Attribute, 959, 996, 1006
addPoint, método da classe Polygon, 501, **503**
AddressBook, aplicativo, exercício de modificação do, 1016
AddressBook, aplicativo, modificação do, 1016
addSeparator, método da classe JMenu, **778**
addTab, método da classe JTabbedPane, **789**
addTableModelListener, método de TableModel, **915**
addTrayIcon, método da classe SystemTray, **XXIV**
addWindowListener, método da classe Window, **773**
ADF Faces, 952
adiamento indefinido, **808**, 839, 857
adição, operador de atribuição composta, +=, **102**
Adicionando serialização de objeto ao aplicativo de desenho MyShape (exercício), 587
adicionando uma referência de serviço Web a um aplicativo, **1024**
adicionar uma referência de serviço Web a um aplicativo NetBeans, 1025
adicionar um handler de evento em Netbeans, XVI
"Adivinhe o número", jogo, 187, 479
adquirir o bloqueio, **812**
Agendamento de evento, XXVII
agendamento de prioridade, 807
agendamento de rodízio, **807**
agendamento preemptivo, **808**
agendando threads, **806**
Agile Alliance (www.agilealliance.org), 18
Agile Manifesto (www.agilemanifesto.org), 18
Agile Software Development, **18**
Agregação de ouvintes de evento, XXVIII
agregação na UML, **369**
agregação, símbolo (na UML), **89**
.aif, extensão de arquivo, **756**, **758**
.aiff, extensão de arquivo, **756**, **758**
AIFF, formato de arquivo do Macintosh (extensões .aif ou .aiff), **756**
Ajax, aplicativo Web, 1006
Ajax (Asynchronous JavaScript and XML), **18**, **1005**, 1006
Ajax, bibliotecas, 1005
Ajax, compatível com
  aplicativos Web, xxii
  bibliotecas de componentes, 995
  componentes JSF, xxii
alfabetando, 519
algoritmo, **82**, 85, 89, 95, 596, LI
  classificação de bucket, 634
  classificação por flutuação, 634
  classificação por inserção, **625**
  classificação por intercalação, **628**
  classificação por seleção, **622**
  em Java Collections Framework, 645
  pesquisa binária, **619**
  pesquisa binária recursiva, 635
  pesquisa linear, **616**
  quicksort, 635
algoritmo de avaliação de expressão pós-fixa, 717
algoritmo de classificação por intercalação, **628**, 632
algoritmo de classificação por seleção, **622**, 625
algoritmo de conversão de infixo para pós-fixo, 717
algoritmo de pesquisa binária, **619**, 622, 651
algoritmo de pesquisa linear, 616, 622
algoritmos de classificação
  classificação de bucket, 634
  classificação por flutuação, 634
  classificação por inserção, **625**
  classificação por intercalação, **628**
  classificação por seleção, **622**
  quicksort, 635
algoritmos de pesquisa
  pesquisa binária, **619**
  pesquisa binária recursiva, 635
  pesquisa linear, **616**
  pesquisa linear recursiva, 635
alinhado à direita, 461
alinhado à esquerda, **126**, 428, 461
alinhamento à direita, 1091, 1099
alinhamento à esquerda, 1091
alinhando componentes em GroupLayout, XIII
alinhando pontos de fração decimal na saída, 1091
allClasses-frame.html, gerado por javadoc, XLI
alocação dinâmica de memória, **696**
alterando a aparência e funcionamento de uma GUI baseada no Swing, 784
alto-falante, 756
altura de uma fonte, **493**
altura de um retângulo em pixels, 487
ALU (*arithmetic and logic unit*), **3**
Amazon, XXVI
Amazon Web Services, XXVI
ambiente de desenvolvimento integrado, **9**
American Standard Code for Information Interchange (ASCII), conjunto de caracteres, 239
American Standard Code for Information Interchange, conjunto de caracteres (ASCII), 134
American Standard Code for Information Interchange, conjunto de caracteres ASCII, 553, XXIX
amostras de cor, **491**
Ampliação de Imagem, exercício, 765
ampliando, 765
analisar os requisitos de um projeto, **16**
análise de texto, 546
análise e design orientado a objetos (*object-oriented analysis and design* — OOAD), **16**
Analog Clock, exercício, 766
anchor, campo da classe GridBagConstraints, **793**
âncora (a), elemento, 736
AND, (em SQL), 909
ângulos de arco negativo, 498
ângulos de arco positivo e negativo, 498
  animação, 730, 743, 752, 765
  www.animationfactory.com, 762
ângulos inicial de um arco, **498**
animação, exercício, 765
animando uma série de imagens, 748
Animator, applet, 724
aninhamento, 85
aninhamento de instruções de controle, **84**
Anotações, xxiv
  @Consumes, **1029**
  @GET, **1029**
  injeção de dependência, **1037**
  @Override, **284**
  Path, **1028**
  @PathParam, **1029**
  @Produces, **1029**
  @Resource, **1037**
  @WebMethod, **1022**

@WebParam, **1022**
@WebService, **1021**
anular um método de superclasse, **281**, 284
Apache Derby, xxii, 926
Apache Software Foundation, 18
Apache Tomcat (tomcat.apache.org), 1021
Apagando aleatoriamente uma imagem, exercício, 765
aparência de blocos de construção, 142
aparência e comportamento, 423, 425, 460, 781
  Nimbus, **420**
aparência e comportamento de uma GUI baseada no Swing, 781
aparência e comportamento de um aplicativo, 423
aparência e funcionamento plugável (*pluggable look-and-feel* — PLAF), **425**, **769**
API (application programming interface), **37**, 155
API de preferências, **661**
APIs comumente utilizadas em mashups, XXVII
aplicativo, **29**, 31, 59
  argumentos da linha de comando, **158**
aplicativo cliente/servidor distribuído, 4
aplicativo de consulta para o banco de dados books, 944
Aplicativo de desenho Interativo, exercício, 480
aplicativo *de n* camadas, **950**
aplicativo de múltiplas camadas, **950**
aplicativo de terminal (Mac OS X), **10**
aplicativo robusto, **336**
aplicativos comerciais, 552
aplicativo Web
  Ajax, 1006
  tradicional, 1005
append, método da classe StringBuilder, **529**
appendRow, método da classe CachedRowSetDataProvider, **1004**
Apple Computer, Inc., 4, XXIX
applet, **723**, 726, 732, 862
  arquivo .class, 729
  arrastável, **748**
Applet, classe
  getAppletContext, método, **863**
  getAudioClip, método, **756**
  getCodeBase, método, **756**
  getParameter, método, **862**
  play, método, **756**
  showStatus, método, **753**
applet-desc, elemento de um documento JNLP, **736**
applet, elemento XHTML, **729**
applet arrastável, 733, **748**
AppletContext, interface, **860**
  showDocument, método, **860**, 863
applet que desenha uma string, 728
applets, diretório
  applets de exemplo no JDK, 723
applets em domínio público, 862
appletviewer, contêiner de applets, **723**, 724
  Applet, menu, 725
  Quit, item de menu, 725
  Reload, item de menu, **725**
application-desc, elemento de um documento JNLP, 736
  height, atributo, 736
  main-class, atributo, 736
  name, atributo, 736
  width, atributo, 736
ApplicationBean, **952**
Aprimorando a classe Date (exercício), 275
Aprimorando a classe Time2 (exercício), 275
aprimorar desempenho da classificação por flutuação, 634
Arc2D, classe, 484
  CHORD, constante, **506**
  OPEN, constante, **506**
  PIE, constante, **506**
Arc2D.Double, classe, **503**, 513
arco, **498**, 724
arco, ângulo, **498**
arco em forma de torta, 506
ArcTest, applet, 724
área ativa, **753**
área de desenho personalizada, 455
área dedicada de desenho, **455**
área de exibição, 729
área de um círculo, 187
área rígida de classe Box, **791**
args, parâmetro, 218
argumento de tipo, **679**
argumento para um método, **32**, 61
argumentos reais de tipo, **674**
ArithmeticException, classe, **337**, 342
arquivo, **552**, 553
arquivo binário, **554**
arquivo de acesso sequencial, 552, **553**, 558, 867
Arquivo de Calendário/Anotações, exercício de, 766
arquivo de classe, **33**
arquivo de contas a receber, 587
arquivo de folha de pagamento, 553
arquivo de leitura, 574
arquivo de texto, **554**
arquivo de transação, 586
arquivo mestre, 586
arquivos de intercalação de áudio/vídeo da Microsoft, 758
arrastando o mouse para destacar, 470
arrastar a caixa de rolagem, 444
array, **190**, 552, 862, LI
  length, variável de instância, **191**
  passando arrays para métodos, 203
  passando elementos de array para métodos, 203
array bidimensional, **209**, 210
array bidimensional com três linhas e quatro colunas, 210
ArrayBlockingQueue, classe, 823, 830
  size, método, **824**
array classificado, 697
arraycopy, método da classe System, 219
array de apoio, **644**
array de arrays de uma dimensão, 209, 210
array dinamicamente redimensionável, 862
ArrayIndexOutOfBoundsException, classe, 199, 336, 503
ArrayList, classe, **639**, 651
ArrayList<T>, classe genérica, **221**, 687, 862, 1045
  add, método, **221**, 862
  clear, método, 221
  contains, método, 221, **222**
  get, método, **222**
  indexOf, método, 221
  isEmpty, método, 253
  remove, método, 221, **222**
  size, método, **222**
  toString, método, 688
  trimToSize, método, 221
array multidimensional, 209, 210
array redimensionável, 862
  implementação da List, 639
Arrays, classe, **219**
  asList, método, **644**
  binarySearch, método, **219**
  equals, método, **219**
  fill, método, **219**
  sort, método, **219**, **619**
  toString, método, 541, 616
arredondando, 1091
arredondando um número, 41, 93, 157, 185
arredondar um número, 127
arredondar um número de ponto flutuante para fins de exibição, 98
artefatos do lado do cliente, **1025**
artefatos do lado do servidor, **1021**
Artist, exercício, 766
árvore, 656, 709, 726
árvore binária, **695**, 713
  classificação, **713**
  exclusão, 719
  pesquisa, 719
árvore componente, **953**
árvore de pesquisa binária, **709**, 713
árvore fortemente empacotada, **714**
árvore fortemente equilibrada, **714**
ascendente de uma fonte, **493**
ASCII (American Standard Code for Information Interchange), conjunto de caracteres, **553**
asList, método da classe Arrays, **644**
asList, método de Arrays, 644
aspas duplas, ", 32, 35
aspas francesas (« e ») na UML, **69**
aspas simples, caractere, 516, 904
assembler, **5**
assert, instrução, **354**, 1069
AssertionError, classe, **354**
assertiva, **354**
assinatura, 175
assinatura de um método, **175**
associação, **16**
associação (na UML), **368**, 369, 393
  nome, 368
associação na UML, **16**
associatividade
  da direita para a esquerda, 104
  da esquerda para a direita, 104
associatividade de operadores, **41**, 46
  da direita para a esquerda, 41
  da esquerda para a direita, 46
associatividade dos operadores, 104
asterisco (*), **41**
asterisco (*), caractere curinga de SQL, 903
ativação em um diagrama de sequência na UML, **385**
atividade (na UML), **83**, **365**, 374, 375
atividades paralelas, 8
ativo de software, 16
ATM, classe (estudo de caso ATM), 368, 371, 372, 374, 377, 382, 383, 384, 391
ATM (*automated teller machine*), estudo de caso, 361, 364
ATMCaseStudy, classe (estudo de caso ATM), 415
ATM, sistema, 365, 366, 370, 374, 377, 391
ator no caso de uso na UML, **364**
atribuir um valor a uma variável, 39
atributo, 391, 393
  compartimento em um diagrama de classes, 372
  declaração na UML, 372, 374
  de tipo na UML, **373**
  de uma classe, **6**
  de um elemento XHTML, **729**
  de um objeto, 15
  na UML, 15, 60, 368, 370, 372, **373**, 374, 375, 397
  nome na UML, 373
atualização automática, 733
atualização de página parcial, **1006**
.au, extensão de arquivo, **756**, **758**
AudioClip, interface, **756**
  loop, método, **756**
  play, método, **756**

stop, método, **756**
Áudio dinâmico e caleidoscópio gráfico, exercício, 766
Australian Botanical Gardens (www.anbg.gov.au/anbg/index.html), 761
authorISBN, tabela do banco de dados books, 901
Author:, nota, **XXXVII**
-author, opção, **XL**
authors, tabela do banco de dados books, 901
@author, tag javadoc, **XXXVII**
autoboxing, 534, **638**, 676
autoboxing um int, 676
auto commit, estado, 939
autoComplete, atributo de um Text Field, **1010**
AutoComplete, interface, 1012
  getOptions, método, **1010**, 1012
autoComplete, propriedade de um componente JSF Text Field, 1006
autocomplete, campo de texto, 1006
autoCompleteExpression, atributo de um Text Field, **1010**
autodocumentando, **38**
Automatic Jigsaw Puzzle Generator, exercício, 766
autoridade certificadora, 733
auto-unboxing, **638**
avaliação da esquerda para a direita, 42
avaliação e certificação em Java, xxvii
avaliando expressões, 717
.avi, extensão de arquivo, **758**
await, método da interface Condition, **836**, 838
awaitTermination, método da interface ExecutorService, **814**
AWT (Abstract Window Toolkit), **423**
  componentes, 423
AWTEvent, classe, 433

B

B2B (business-to-business), transações, 1019
Babbage, Charles, 8
b, caractere de conversão, **1096**
B, caractere de conversão, **1096**
Background Audio, exercício, 765
Backpack, XXVII
Backpack API, XXVII
BalanceInquiry, classe (estudo de caso ATM), 367, 370, 371, 372, 373, 375, 377, 383, 384, 385, 393, 395, 396
Balking, padrão de design, LIX, LXVII
banco de dados, **553**, **899**, 903
  tabela, **900**
banco de dados MySQL, 911
banco de dados relacional, 899, **900**
BankDatabase, classe (estudo de caso ATM), 367, 370, 372, 377, 378, 382, 383, 384, 385, 386, 391, 393
BarChart, applet, 724
barra de asteriscos, 196
barra de menus, **419**, **773**, 777
  de um JComboBox, **444**
barra de rolagem horizontal, diretiva, 470
barra de status, 728
barra de título, **419**, 425
barra de título de janela interna, 786
barra de título de uma janela, 422, **772**
barra invertida, sequência de escape (\\), 35
barras invertidas (\), 1103
base, XLIII
baseado em evento, **428**
BASELINE, constante de alinhamento em GroupLayout, XIII
BasePlusCommissionEmployee, classe que estende CommissionEmployee, 316
Basic (Beginner's All-Purpose Symbolic Instruction Code), **8**
Basic (Beginner's All-Purpose Symbolic Instruction Code), 695
Basic Latin, bloco, XXXIII
BasicStroke, classe, **484**, 505, 506
  CAP_ROUND, constante, **506**
  JOIN_ROUND, constante, **506**
BCPL, linguagem de programação, 6
bean de página, **952**
Beginner's All-Purpose Symbolic Instruction Code (Basic), 8
Bell Laboratories, 6
biblioteca de classes, **279**, 297
biblioteca de classes Java, **155**
biblioteca de tags, **951**, 953
bibliotecas de tags personalizados, **951**
BigDecimal, classe, 70, 127, **593**
  documentação (java.sun.com/javase/6/docs/api/java/math/BigDecimal.html), 127
BigInteger, classe, **593**, 841
  add, método, **595**
  compareTo, método, **594**
  multiply, método, **594**
  ONE, constante, **594**, 595
  subtract, método, **595**
  ZERO, constante, **595**
binário, 188
  sistema numérico de base 2, II
binarySearch, método
  de Arrays, **219**, 221
  de Collections, 645, **651**, 652
BindException, classe, **871**
BindingProvider, interface, **1045**
  getRequestContext, método, **1045**
bit (dígito binário), **552**
BitSet, classe, **XLIII**, LI
  and, método, **LI**
  clear, método, **LI**
  equals, método, **LI**
  get, método, **LI**
  or, método, **LI**
  set, método, **LI**
  size, método, **LI**
  toString, método, **LI**
  xor, método, **LI**
bits vagos, XLIX
bitwise, operadores, 136, XLIV
  ^, (OU exclusivo sobre bits), **XLIII**
  &, (AND sobre bits), **XLIII**
  <<, (deslocamento de bits para a esquerda), **XLIII**
  >>, (deslocamento para a direita com sinal), **XLIII**
  >>>, (deslocamento para a direita sem sinal), **XLIII**
  |, (OU inclusivo sobre bits), **XLIII**
  ~, (complemento), **XLIII**
Blackjack, 1035
  jogo, 896
  Web Service Modification, 1065
_blank, frame-alvo, **863**
B, linguagem de programação, 6
Blink, applet, 724
BlockingQueue, interface, **823**
  put, método, **823**, 824
  take, método, **823**, 824
bloco de construção aninhado, 144
Bloco de Notas, 8
bloco rotulado, LIV
blocos de construção, 82
blocos de construção empilhados, 144
blog, **18**
Blogger ATOM, feed, XXVII
Blogging, XXVII
blogosfera, **18**
bloquear, **866**
bloquear o incremento de um JSlider, **769**
bloqueia até conexão ser recebida, 871
bloqueia um objeto, 827
bloqueio de monitor, **812**
bloqueio intrínseco, **812**
BlueJ (www.blueJ.org), 9
Boa prática de programação, **7**
Boas práticas de programação, visão geral, xxv
body, elemento XHTML, **729**
Bohm, C., 83, 144
BOLD, constante da classe Font, **491**, 492
Booch, Grady, 16
books, banco de dados, 900
  relacionamentos de tabela, 902, 945
boolean
  expressão, **86**, 1082
  promoções, 162
Boolean
  atributo na UML, **371**
  classe, **638**
boolean, tipo primitivo, **86**, 1069, 1070, 1082
BorderLayout, classe, **300**, **452**, 459, 463, 470
  CENTER, constante, 300, **452**, 463, 465
  EAST, constante, 300, **452**, 463
  NORTH, constante, 300, **452**, 463
  SOUTH, constante, 300, **452**, 463
  WEST, constante, 300, **452**, 463
botão, **419**, 435
botão de alternação, **435**
botão de comando, **435**
botão de estado, **438**
botão de opção, **435**, 440
botão de opção, grupo, **440**
botão do meio do mouse, 454
botão do mouse, 725
botão, rótulo, **436**
BOTH, constante da classe GridBagConstraints, **793**
Box, classe, **470**, 789, 791
  createGlue, método, **792**
  createHorizontalBox, método, **470**, 791
  createHorizontalGlue, método, **791**
  createHorizontalStrut, método, **791**
  createRigidArea, método, **791**
  createVerticalBox, método, **791**
  createVerticalGlue, método, **791**
  createVerticalStrut, método, **791**
  X_AXIS, constante, **792**
  Y_AXIS, constante, **792**
boxing, conversão, **638**, 676
BoxLayout, classe, **470**, 789
BoxLayout, gerenciador de layout, 789
braile, leitor de tela, 425
break, 1069
break, instrução, **132**, 134, 153
break, instrução rotulada, LIV
  saindo de uma aninhada for instrução, LIV
Bridge, padrão de design, LIX, LXIII
brilho, **491**
browse, método da classe Desktop, **XXII**
buffer, **577**, **818**
buffer circular, **830**
buffer compartilhado, 818
BufferedImage, classe, **506**
  createGraphics, método, **506**
  TYPE_INT_RGB, constante, **506**
BufferedInputStream, classe, **577**

BufferedOutputStream, classe, **577**, LXVIII
  flush, método, **577**
BufferedReader, classe, **578**
BufferedWriter, classe, **578**
buffer limitado, **830**
Builder, padrão de design, LX
Build, opção, no NetBeans, 1022
busca (fetch), 238
business-to-business (B2B), transações, **1019**
ButtonGroup, classe, **440**, 773, 778
  add, método, **442**
Button componente JSF
  primary, propriedade, **996**
  reset, propriedade, **996**
Button, componente JSF, 961, **964**
byte, **553**
Byte, classe, **638**
byte, palavra-chave, 1070
ByteArrayInputStream, classe, 577
ByteArrayOutputStream, classe, 577
byte, tipo primitivo, 129, **553**, 1069, XLIII
  promoções, 162
Bytecode, **10**, 33, 695

C

cabeça de uma fila, **695**, 707
cabeçalho da instrução de repetição for, **122**
cabeçalho de método, **59**
Caça-níqueis, exercício, 766
cache, **949**
CachedRowSet, interface, **924**, 998, 1001
  close, método, 926
CachedRowSetDataProvider, classe, **998**, 1004
  appendRow, método, 1004
  commitChanges, método, **1004**
  refresh, método, **1004**
caixa eletrônico (*automated teller machine* — ATM), 361, 364
  estudo de caso, xxi
  interface com o usuário, 361
caixa de combinação, **419**, 443
caixa de diálogo, **74**, **421**, 422, 777
caixa de diálogo modal, **422**, **778**
caixa de rolagem, **444**
caixa de seleção, **435**, 440
caixa de seleção, rótulo, **440**
Calculadora de frequência cardíaca alvo, 80
calculadora de pegada de carbono, 26
cálculo aritmético, 41
cálculos, 4, 47, 83
cálculos fatoriais com um método recursivo, 592, 593
cálculos matemáticos, 7
cálculos monetários, 127
Calendar, classe, **1096**
  getInstance, método, **1096**
Callable, interface, **851**
  call, método, **851**
CallableStatement, interface, **938**
call, método da interface Callable, **851**
camada de cliente, **950**, LXX
camada de dados, **950**
camada de informações, **950**, LXIX
camada em um aplicativo de múltiplas camada, **950**
camada inferior, **950**
camada intermediária, **950**, LXIX
camada superior, **950**
caminho absoluto, **555**, 556, 558
caminho geral, **506**
caminho para um recurso, 948
caminho relativo, **555**
campo, **16**, **63**, **553**
  valor inicial padrão, **65**
campo de texto, **75**
campo de uma classe, 172, 553
campos ocultos, 172
cancel, método da classe SwingWorker, **850**
CANCEL_OPTION, constante de JFileChooser, **581**
Cancel, botão, 75
CannotRealizePlayerException, exceção, **759**
canRead, método de File, 556
canto superior esquerdo de um componente GUI, 105, 484
canWrite, método de File, 556
c, opção do comando jar, **734**
capacidade da StringBuilder, 526
capacidade de reutilização, 678, 695
capacidade de reutilização de software, **156**
capacity, método
  da classe StringBuilder, **527**
CAP_ROUND, constante da classe BasicStroke, **506**
Capítulo sobre arrays na *Java Language Specification* (java.sun.com/docs/books/jls/third_edition/html/arrays.html), 299
captura
  uma exceção, 339
  uma exceção de superclasse, 344
Capturando exceções com escopos externos, exercício, 359
Capturando exceções com superclasses, exercício, 359
Capturando exceções utilizando a classe Exception, exercício, 359
capture ou declare, requisito, **343**
caractere, 163, **553**
  conjunto, 54
  constante, **134**
  literal, **516**
  set, **553**
caractere de barra (/) em tags de fechamento, **735**
caractere de conversão, **1091**
caractere de escape, **35**, 908
caractere de espaço em branco, 30, 525, 535, 536
caractere de sufixo de conversão, **1094**
  a, **1095**
  A, **1095**
  b, **1095**
  B, **1095**
  c, **1094**
  d, **1095**
  D, **1095**
  e, **1095**
  F, **1095**
  H, **1095**
  I, **1095**
  j, **1095**
  k, **1095**
  l, **1095**
  m, **1095**
  M, **1095**
  p, **1095**
  P, **1095**
  r, **1095**
  R, **1095**
  S, **1095**
  T, **1095**
  y, **1095**
  Y, **1095**
  Z, **1095**
caractere especial, 38, **516**
caractere palavra (expressões regulares), **536**
caracteres
  string de, 32
Caracteres aleatórios, exercício, 512
caracteres de conversão
  %, 1097
  a, 1093
  A, 1093
  b, **1096**, 1097
  B, **1096**
  c, **1094**
  C, **1094**
  d, **1092**
  e, 1092
  E, 1092
  f, **1093**
  g, **1093**
  G, **1093**
  h, 1097
  H, **1096**
  n, 1097
  o, **1092**
  s, **1094**
  S, **1094**
  t, **1094**
  T, **1094**
  x, **1092**
  X, **1092**
caracteres de conversão de inteiros, **1092**
caractere separador, **558**
caracteres finais de espaço em branco, 525
CardTest, applet, 724
carregador de classe, **10**, **267**, 427
carregando, **10**
Carregando e executando um AudioClip, 756
carregando e exibindo uma imagem em um applet, 743
carregar outra página Web em um navegador, 753, 755
case, palavra-chave, **132**
case, palavra-chave, 1069
CashDispenser, classe (estudo de caso ATM), 367, 369, 372, 377, 386, 404
caso básico, **591**, 595, 599
caso de uso na UML, **364**
casos de uso, diagrama na UML, **364**, 365
casos de uso, modelagem, **364**
cassino, 164, 168
catálogo de endereços, aplicativo Web multicamada baseado em banco de dados, xxii
catch
  bloco, 340, 341, 342, 345, 348, 351
  cláusula, **340**, 1069
  palavra-chave, **340**
catch, bloco, correspondendo, 341
ceil, método de Math, 157
Celsius, 479, 1108
  equivalente de uma temperatura em Fahrenheit, 187
CENTER, constante
  BorderLayout, **452**, 463, 465
  FlowLayout, **463**
  GridBagConstraints, **793**
  GroupLayout, XIII
centralizado, 461
certificado de segurança, 733
certificado digital, 733
Chain-of-Responsibility, padrão de design, LIX, LXII, LXV
chamada assíncrona, **384**
chamada de método, **57**, 156, 159
chamada por referência, **205**
chamada por valor, **205**

chamada recursiva indireta, **591**
Chamadas de método feitas dentro da chamada fibonacci(3), 596
chamadas de método na pilha, método de execução do programa, 597
chamada síncrona, **384**
Chamando atenção para uma imagem, exercício, 765
ChangeEvent, classe, **772**
ChangeListener, interface, **772**
  stateChanged, método, **772**
char
  tipo primitivo, 129
char
  array, 517
  palavra-chave, 1069, 1070
  promoções, 162
Character, classe, 516, 531, **638**
  charValue, método, **534**
  digit, método, **533**
  forDigit, método, **533**
  isDefined, método, **533**
  isDigit, método, **533**
  isJavaIdentifierPart, método, **533**
  isJavaIdentifierStart, método, **533**
  isLetter, método, **533**
  isLetterOrDigit, método, **533**
  isLowerCase, método, **533**
  isUpperCase, método, **533**
  static, conversão, métodos, 533
  toLowerCase, método, **533**
  toUpperCase, método, **533**
CharArrayReader, classe, **578**
CharArrayWriter, classe, **578**
charAt, método
  da classe String, **518**
  da classe StringBuilder, **528**
char, tipo primitivo, **38**
CharSequence, interface, **541**
charValue, método da classe Character, **534**
chat cliente-servidor, **868**
chave de classificação, **615**
chave de pesquisa, **615**
chave direita, }, 31, 37, 91, 97
chave esquerda, {, 31, 37
chave estrangeira, **901**, 903
chave primária, **900**, 902
chaves ({ e }), 88, 97, 122, 129, 193
  não requerido, 131
chave/valor, par, 659, 976, 982
chegada de mensagem de rede, 342
CHORD, constante da classe Arc2D, **506**
ciclo de execução de instrução, 238
ciclo de vida de software, **364**
ciclo de vida de uma thread, 805
ciclo de vida do processamento de evento, **956**, 958
  destroy, método, **956**, 958, 1004
  init, método, **956**, 958
  preprocess, método, **956**, 958
  prerender, método, **956**, 958
Círculos concêntricos que utilizam a classe Ellipse2D.Double exercício, 512
Círculos concêntricos que utilizam o método drawArc, exercício, 512
círculo sólido cercado por um círculo vazio (na UML), **84**
círculo sólido incluído em um círculo aberto (para representar o final de um diagrama de atividades da UML), 375
círculo sólido (na UML), **84**
círculo sólido (para representar um estado inicial em um diagrama da UML) na UML, **374**, 375
Círculos que utilizam classe Ellipse2D.Double, exercício, 512
circunferência, 53, 741
CJK Unified Ideographs, bloco, XXX
clareza, 2, 11
.class, arquivo
  separado um para cada classe, 246
class, 553
Class, classe, 298, **320**, 427, 919
  getName, método, 298, **320**
  getResource, método, **427**
.class, extensão de arquivo, **756**
class, palavra-chave, **30**, 59, 1069
.class, arquivo, **10**, **33**
ClassCastException, classe, 336, 677
classe, **7**, 15, 155, 372, 377, 378, 391
  arquivo, **33**
  campo, **63**
  class, palavra-chave, 59
  construtor, **60**, 68, 393
  construtor padrão, **68**
  declaração, **30**, 728
  declarando com um método, **58**
  get, método, 248
  instanciar um objeto, **58**
  nome, **30**, 265, 393
  ocultamento de dados, **64**
  set, método, 248
  variável de instância, **58**, 64, 158
classe abstrata, 306, **309**, 311, 322, LXII
classe adaptadora, **452**
classe aninhada, **431**, 784
  relacionamento entre uma classe interna e sua classe de primeiro nível, 440
classe auto-referencial, **695**, 696
classe básica, **279**
classe concreta, **309**
classe de applet compilado, 729
classe de caractere predefinida (expressões regulares), **536**
classe definida pelo programador, **30**
classe de primeiro nível, **431**
classe derivada, **279**
classe genérica, **221**, 678
classe interior anônima, 432, **444**, 456
classe interna, **431**, 440, 456, 778
  anônimo, **444**
  objeto de, 440
  relacionamento entre uma classe interna e sua classe de primeiro nível, 440
classe não pode estender uma classe final, 321
classe parametrizada, **678**
classe proprietária, 297
classe proxy de um serviço Web, **1022**, 1025
Classe que utiliza grade Line2D.Double, exercício, 512
Classe que utiliza grade Rectangle2D.Double, exercício, 512
Classes
  AbstractButton, **435**, 438, 777
  AbstractCollection, **664**
  AbstractList, **664**
  AbstractMap, **664**
  AbstractPageBean, **954**, 1050
  AbstractQueue, **664**
  AbstractSequentialList, **664**
  AbstractSessionBean, **979**
  AbstractSet, **664**
  AbstractTableModel, **915**, 919
  ActionEvent, **432**, 435, 470, 751
  Arc2D, 484
  Arc2D.Double, **503**
  ArithmeticException, **337**
  ArrayBlockingQueue, 823, 830
  ArrayIndexOutOfBoundsException, 336
  ArrayList, **639**, 651
  ArrayList<T>, 221, 222, 687, 862
  Arrays, **219**
  AssertionError, **354**
  AWTEvent, 433
  BasicStroke, **484**, 505, 506
  BigDecimal, 70, 127, **593**
  BigInteger, **593**, 841
  BindException, **871**
  BitSet, **XLIII**
  Boolean, **638**
  BorderLayout, **452**, 459, 463, 470
  Box, **470**, 789, 791
  BoxLayout, **470**, 789
  BufferedImage, **506**
  BufferedInputStream, **577**
  BufferedOutputStream, **577**
  BufferedReader, **578**
  BufferedWriter, **578**
  ButtonGroup, **440**, 773, 778
  Byte, **638**
  ByteArrayInputStream, 577
  ByteArrayOutputStream, 577
  CachedRowSetDataProvider, **998**, 1004
  Calendar, **1096**
  ChangeEvent, **772**
  Character, 516, 531, 533, **638**
  CharArrayReader, **578**
  CharArrayWriter, **578**
  Class, 298, **320**, 427, 919
  ClassCastException, 336, 677
  Collections, **639**, 675
  Color, **176**, **484**
  Component, **424**, 449, 486, 746, 752, 789, 796
  ComponentAdapter, 453
  ComponentListener, 460
  Container, **424**, 447, 459, 465, 467
  ContainerAdapter, 453
  Cookie, 976
  DatagramPacket, **877**, 893
  DatagramSocket, **877**
  DataInputStream, **577**
  DataOutputStream, **577**
  Date, **1096**
  Desktop, **XXII**
  Dimension, **752**
  Double, **638**, 687
  DriverManager, **913**
  Ellipse2D, 484
  Ellipse2D.Double, **503**
  Ellipse2D.Float, **503**
  EmptyStackException, **655**
  EnumSet, **257**
  Error, **343**
  EventListenerList, **434**
  Exception, **343**
  ExecutionException, 842
  Executors, **810**
  File, **555**
  FileInputStream, **555**
  FileOutputStream, **555**
  FileReader, **555**, 578
  FileWriter, **555**
  FilterInputStream, **577**
  FilterOutputStream, **577**
  Float, **638**

FlowLayout, 427, 461
FocusAdapter, 453
Font, **440**, **484**, 491
FontMetrics, **484**, 494
Formatter, **555**, **1091**
Frame, 772
GeneralPath, 484, **506**
GradientPaint, **484**, 505
Graphics2D, **484**, 503, 505
Graphics, **456**, **484**, 502, 747
GridBagConstraints, **792**, 796
GridBagLayout, 789, 792, 796
GridLayout, 461, **466**
GroupLayout, 460, **XII**
GroupLayout.Group, **XIII**
GroupLayout.ParallelGroup, **XIII**
GroupLayout.SequentialGroup, **XIII**
Gson, 1033
HashMap, **658**, 862, 976, 982, 983
HashSet, **656**
Hashtable, **658**
HttpServletRequest, 978
HttpServletResponse, 976
HyperlinkEvent, **863**, 865
IllegalMonitorStateException, **825**, 836
Image, **744**
ImageIcon, **427**, 744, 751
InetAddress, **871**, 876, 879, LXVIII
InputEvent, **450**, 454, 458
InputMismatchException, **338**
InputStream, **576**, 866, 867
InputStreamReader, **578**
Integer, 422, **638**, 687
InterruptedException, **809**
ItemEvent, **440**, 442
JApplet, **728**, 773
JAXB, **1029**
JButton, 423, **436**, 437, 465
JCheckBox, 423, **438**
JCheckBoxMenuItem, 773, 778
JColorChooser, **489**
JComboBox, 423, **443**, 793
JComponent, 425, 427, 434, 443, 445, 455, 467, 484, 486, 752
JdbcRowSetImpl, **925**
JDesktopPane, **784**, 802
JDialog, **778**
JEditorPane, **863**
JFileChooser, **578**
JFrame, 772
JInternalFrame, **784**
JLabel, 423, **425**
JList, 423, 445
JMenu, **773**, 776, 786
JMenuBar, 773, 778, 786
JMenuItem, **773**, 786
JOptionPane, **74**, **421**, 799
JPanel, 423, 455, 461, 467, 748, 770
JPasswordField, **429**, 433
JPopupMenu, **779**
JProgressBar, 848
JRadioButton, **438**, 440, 442
JRadioButtonMenuItem, 773, 778
JScrollPane, 447, 470
JSlider, **769**, 771, XIII
JTabbedPane, **787**, 792
JTable, **915**
JTextArea, 459, **468**, 469, 793, 795
JTextComponent, **429**, 431, 468
JTextField, 423, **429**, 431, 433, 468
JToggleButton, **438**
KeyAdapter, 453
KeyEvent, **435**, 458
Line2D, 484, **506**
Line2D.Double, **503**
LinearGradientPaint, 505
LineNumberReader, **578**
LinkedList, **639**
ListSelectionEvent, **445**
ListSelectionModel, **447**
Long, **638**
MalformedURLException, **863**
Manager, **758**
Matcher, 516, **541**
Math, 156, 157
MemoryImageSource, 765
MouseAdapter, 452
MouseEvent, **435**, 449, 781
MouseMotionAdapter, 453, 456
MouseWheelEvent, **450**
NullPointerException, 336
Number, 687
Object, 258
ObjectInputStream, **555**, 866, 867, 871
ObjectOutputStream, **555**
OutputStream, **576**, 866, 867
OutputStreamWriter, **578**
Pattern, 516, **541**
PipedInputStream, 576
PipedOutputStream, 576
PipedReader, **578**
PipedWriter, **578**
PixelGrabber, 765
Point, **456**
Polygon, **484**, 500
PrintStream, **577**
PrintWriter, 578
PriorityQueue, **655**
Properties, **661**, 981
RadialGradientPaint, 505
Random, 163, **164**, 231
Reader, **578**
Rectangle2D, 484
Rectangle2D.Double, **503**
ReentrantLock, **835**, 836
RoundRectangle2D, 484
RoundRectangle2D.Double, **503**, 506
RowFilter, **924**
RuntimeException, **343**, 349
Scanner, **38**, **62**
ServerSocket, **866**, 871, 887
ServiceManager, **746**
Short, **638**
Socket, **866**, 876, 887, 888, LXVIII
SocketException, **877**
SplashScreen, **XXI**
SQLException, 913
SQLFeatureNotSupportedException, 919
Stack, **654**
StackTraceElement, **351**
String, **74**, 516
StringBuffer, **526**
StringBuilder, 516, **526**
StringIndexOutOfBoundsException, **523**, 528
StringReader, **578**
StringWriter, **578**, **1030**
SwingUtilities, **784**, **871**
SwingWorker, **841**
SystemColor, **505**
SystemTray, **XXIV**
TableModelEvent, 923
TableRowSorter, **923**
TexturePaint, **484**, 505, 506
Thread, **807**, 809
Throwable, **343**, 349
Timer, **751**, 752
TrayIcon, **XXIV**
TreeMap, **658**
TreeSet, **656**
Types, **914**
UIManager, **784**
UIViewRoot, **953**
UnknownHostException, **867**
UnsupportedOperationException, **644**
URL, 755
ValidatorException, 970
Vector, **639**
Window, 772
WindowAdapter, 453, **924**
Writer, **578**
classes adaptadoras utilizadas para implementar rotinas de tratamento de evento, 452
classes de evento, 433
classes empacotadoras de tipo, 532, **638**, 675
  implementa Comparable, 675
classes genéricas, **671**
classes numéricas, 638
classe utilitária que exibe representação de bits de um inteiro, XLV
classificação de bucket, 634
classificação de dados, **615**
classificação descendente (DESC), 905, 906
classificação por flutuação, 634
  aprimorando o desempenho, 634
classificação por inserção, **625**
  algoritmo, **625**, 627
classificação por intercalação, **628**
classificação, técnicas, 724
classificando, 695
  com a Comparator, 647
  ordem decrescente, 646
classificando dados, 622
classpath, **267**, 913
-classpath, opção de linha de comando para java, **267**
-classpath, opção de linha de comando para javac, **267**
CLASSPATH
  informações (java.sun.com/javase/6/docs/technotes/tools/index.html#general), 267
  variável de ambiente, 34, **267**
-classpath, linha de comando, argumento, 560
Clean and Build, opção, no NetBeans, 1022
Clean, opção, no NetBeans, 1022
clear, comando de depurador, **1088**
clear, método
  de ArrayList<T>, 221
  de BitSet, **LI**
  de List, **643**
  de PriorityQueue, **656**
clearRect, método da classe Graphics, 496
clicando a caixa de fechamento, 802
clicar com o mouse, 438, 724, 725
clicar nas setas de rolagem, 444
clicar uma guia, 726
clicar um botão, 428
cliente, **859**
  de uma classe, **16**, 377, 384
  objeto, LXVIII
cliente, conexão, 866
cliente de um objeto, **66**
C, linguagem de programação, 6
C#, linguagem de programação, **8**
C++, linguagem de programação, 6
clip-art (www.clipart.com), 762
clipe de áudio, **756**, 758, 761

clique com o botão central do mouse, 455
clique com o botão esquerdo do mouse, 454
clique de botão do mouse, 454
clique de mouse, 453
cliques com os botões esquerdo, central e direito do mouse, 453
clock, 724, 765
Clock, applet, 724
clonando objetos
  cópia em profundidade, **298**
  cópia superficial, **298**
Cloneable, interface
  documentação (java.sun.com/j2se/5.0/docs/api/java/lang/Cloneable.html), LXXI
clone, método de Object, 298
clone, objeto, 570
close, método
  da interface Connection, 915
  da interface ResultSet, 915
  de CachedRowSet, 926
  de Connection, 915
  de Formatter, **562**
  de interface Statement, 915
  de JdbcRowSet, **926**
  de ObjectOutputStream, **574**
  de ResultSet, 915
  de Socket, **867**
  de Statement, 915
closePath, método da classe GeneralPath, **508**
Cobol (COmmon Business Oriented Language), **7**
code, atributo de <applet> tag, 729
codebase, atributo do elemento jnlp, 735
codificação de caracteres, **XXIX**
código autodocumentado, **38**
código de cliente, 307
código de operação, 236
código dependente de implementação, 245
código de tecla virtual, **460**
código de versão do recurso, 345
código-fonte, **8**, 297
código-fonte aberto, software, xxii, **18**
Código Morse, 548
Código Morse em Rede, 897
código, reutilização, 279, 637
coerção
  downcast, **307**
  operador, 54, **97**, 163
coerção de tipo, **98**
colaboração na UML, **382**, 384
cola horizontal, **791**
colchete angular (<>) para elementos XML, **735**
colchetes, [], **190**, 198
colchetes angulares (< e >), **674**
colchetes mais internos, 198
coleção, **221**
coleções, **637**
  estrutura, **637**
  lineares, 696
  não-modificáveis, 637, 639
  sincronizadas, 639
coleções genéricas, xxi
coleta de lixo, **258**, 805
  automática, 345
coletando requisitos, **364**
coletor de lixo, **258**, 342, 345, 756
colisão de nomes, **265**
colisão em uma tabela de hash, **659**
Collection, interface, 637, **639**, 641, 645
  contains, método, **641**
  iterator, método, **641**
Collections, classe, **639**, 675
  addAll, método, 645, **652**
  binarySearch, método, 645, **651**, 652
  copy, método, 645, **650**
  disjoint, método, 645, **653**
  empacotador, métodos, **639**
  fill, método, 645, **650**
  frequency, método, 645, **653**
  max, método, 645, **650**
  min, método, 645, **650**
  reverse, método, 645, **650**
  reverseOrder, método, **646**
  shuffle, método, 645, **648**, 649
  sort, método, **645**
Collections, métodos reverse, fill, copy, max e min, 650
colocando chamadas de método em linha, **251**
colocar em uma pilha, **161**
Color.BLACK, 176
Color.BLUE, 176
Color, classe, **176**, **484**
  getBlue, método, **487**, 489
  getColor, método, **487**
  getGreen, método, **487**, 489
  getRed, método, **487**, 489
  setColor, método, **487**
Color, constante, 486, 488
Color.CYAN, 176
Color.DARK_GRAY, 176
Color.GRAY, 176
Color.GREEN, 176
Color.LIGHT_GRAY, 176
Color.MAGENTA, 176
Color.ORANGE, 176
Color.PINK, 176
Color.RED, 176
Color.WHITE, 176
Color.YELLOW, 176
Colorindo fotografias e imagens em preto e branco, exercício, 766
coluna, 209, 900
coluna autoincrementada, **901**, 908
coluna de identidade, 927
coluna em uma tabela de banco de dados, **900**
colunas de um array bidimensional, 209
com.google.gson.Gson, pacote, 1033
com.sun.rave.web.ui.appbase, pacote, **954**
com.sun.rowset, pacote, **925**
com.sun.webui.jsf.component, pacote, **954**
com.sun.webui.jsf.model, pacote, 1012
comentário
  de uma única linha, 31
  de única linha (fim do arquivo), 31
  fim de linha (de uma única linha), //, 29
  Javadoc, **30**, 947
  tradicional (/* */), 29
comércio eletrônico, XXVII
comissão, 115, 230
comissões sobre vendas, 230
Command Prompt, **10**
Command, padrão de design, LIX, LXII, LXV
CommissionEmployee, classe derivada de Employee, 315
commit, método da interface Connection, **939**
commitChanges, método da classe CachedRowSetDataProvider, **1004**
Comparable, interface, **645**, 712
  compareTo, método, 645
Comparable<T>, interface, 329, 521, **675**, 712
  compareTo, método, **675**
comparação lexicográfica, **520**, 521
comparando objetos String, 519
Comparator, interface, **645**
  compare, método, **647**
Comparator, objeto, 645, 650, 657, 658
  em sort, 645
compare, método de interface Comparator, **647**
compareTo, método
  da classe String, 519, 521
  de Comparable, 645
compareTo, método da classe BigInteger, **594**
compareTo, método de Comparable<T>, **675**
compartilhamento de foto, 17, XXVII
compartilhamento de tempo, **4**
compartilhando vídeo, 17, XXVII
compartimento, **60**
  em um diagrama de classe UML, 60
compartimento de operação em um diagrama de classes, 377
compensação entre espaço na memória e tempo de execução, 659
compilador, **5**
compilando um aplicativo com múltiplas classes, 60
compilar, 33
compilar um programa, **10**
compile, método da classe Pattern, **541**
complemento de bits (~), operador, **XLIII**, XLIV
complemento de dois, VIII
complemento de um, VIII, XLIX
complemento lógico, !, operador, **138**
Complex, 276
complexidade, teoria da, 596
Component, classe, **424**, 449, 486, 490, 746, 752, 789, 796, LXIII, LXIV
  addKeyListener, método, **459**
  addMouseListener, método, **452**
  addMouseMotionListener, método, **452**
  documentação (java.sun.com/javase/6/docs/api/java/awt/Component.html), 424
  getHeight, método, **747**
  getMaximumSize, método, XIII
  getMinimumSize, método, **752**, 772, XIII
  getPreferredSize, método, **752**, 772, XIII
  getWidth, método, **747**
  repaint, método, **456**
  setBackground, método, 490
  setBounds, método, 460
  setFont, método, **440**
  setLocation, método, 460, **772**
  setSize, método, 460, 772
  setVisible, método, **465**, 772
ComponentAdapter, classe, 453
componente, 6, **16**, 163, 447, 724
componente de origem, **781**
componente de um array, **190**
componente GUI de peso leve, **424**, 778
componente na UML, **16**
componente opaco, **455**
componente reutilizável padrão, **279**
componentes de peso pesado, **424**
componentes GUI Swing, **423**
componentes Java puros, 423
Componentes JSF comumente utilizados, 961
componentes reutilizáveis de software, 6, 163, **279**
ComponentListener, interface, 453, 460
comportamento, 15, 377
  de uma classe, **6**
  de um objeto, **15**
  de um sistema, 374, 375, 376, 384
comportamento de sistema, **365**
composição, **253**, 279, 281, 369, 387
  na UML, **369**
composições de data e hora, **1094**
Composite, padrão de design, LIX, LXIII, LXIV
comprimento da fila, **866**

computação, 2
computação cliente/servidor, **4**
computação corporativa crítica, 336
computação de missão crítica, 336
computação distribuída, **4**
computação pessoal, **4**
computador, **2**
computador cliente, **4**
computadores na educação, 188
Computadorização dos registros de saúde, 80
comunicação baseada em pacotes, **859**
comunicação baseada em socket, **859**
`concat`, método da classe `String`, **524**
concatenação, **160**
concatenação de string, **160**
concatenar strings, 260
conclusão de E/S de disco, 342
concorrência, 804
concorrência de conjunto de resultados, **918**
`CONCUR_READ_ONLY`, constante, **919**
`CONCUR_UPDATABLE`, constante, **919**
condição, **43**, 128
condição de guarda (na UML), **85**, 375
condição dependente, 137
condição, objeto, **836**
condição simples, 136
`Condition`, interface, **836**
  `await`, método, **836**, 838
  `signalAll`, método, **836**
  `signal`, método, **836**
conectando tabelas de banco de dados, **901**, 911
conectar ao servidor, 866, 867
conectar-se a um banco de dados, 911
conexão, **859**, 867, 876, 877, 887, 888
conexão entre cliente e servidor termina, 867
conexão entre programa Java e banco de dados, 913
configurar (*set*) um valor, 66
configurar tratamento de eventos, 431
Configure Virtual Forms..., 1007
confinamento de thread, **840**
confirmar uma transação, **939**
conflito de nomes, **265**
confundir o operador de igualdade == com o operador de atribuição =, 45
conjunto de caracteres, **553**, XXIX
conjunto de caracteres de dois bytes (double-byte character set — DBCS), **XXXI**
conjunto de caracteres multibyte (*multibyte character set* — MBCS), **XXXI**
Conjunto de chamadas recursivas para `fibonacci(3)`, 595
conjunto de constantes
  como uma interface, 322
Conjunto de inteiros, exercício, 276
conjunto vazio, 276
`Connection`, interface, **913**, 914, 918, 939
  `close`, método, 915
  `commit`, método, **939**
  `createStatement`, método, **914**, 918
  `getAutoCommit`, método, **939**
  `prepareStatement`, método, **929**
  `rollBack`, método, **939**
  `setAutoCommit`, método, **939**
constante, 262
  em uma interface, 329
  `Math.PI`, 53
constante de enumeração, **171**
constante de ponto flutuante, 125
constante identificado, **195**
construindo seu próprio compilador, 695
construindo seu próprio computador, 236
construtor, **60**, 68, 393
  chamar outro construtor da mesma classe utilizando `this`, 250
  lista de parâmetros, 68
  múltiplos parâmetros, 70
  sem argumento, **250**
  sobrecarregado, **248**
construtores não podem especificar um tipo de retorno, 69
construtores sobrecarregados, **248**
construtor padrão, **68**, 252, 284
  de uma classe interna anônima, 445
construtor sem argumento, **250**, 251
consulta, **899**, 900
consultar um banco de dados, 911
`@Consumes`, anotação, **1029**
consumidor, **818**
consumir um evento, **432**
consumir um serviço Web, **1019**, 1020, 1060
`cont`, comando de depurador, **1080**
conta de poupança, 125
contador, **90**, 94, 99
Contador de hits de página com `ApplicationBean`, 993
Contador de hits de página com `ApplicationBean` exercício, 993
Contador de hits de página com cookies, 993
Contador de hits de página com cookies, exercício, 993
contagem de cliques, 453
`Container`, classe, **424**, 447, 459, 465, 467, LXIV
  documentação (`java.sun.com/javase/6/docs/api/java/awt/Container.html`), 424
  `setLayout`, método, **427**, 460, 465, 792
  `validate`, método, **467**
`ContainerAdapter`, classe, 453
`ContainerListener`, interface, 453
`contains`, método
  de `Collection`, **641**
`contains`, método da classe `ArrayList<T>`, 221, **222**
`containsKey`, método de interface `Map`, **661**
contêiner de applets, **723**, 730
contêiner de servlets, **950**
contêiner para menus, 773
conteúdo dinâmico, **6**
contexto de imagens gráficas, **484**
`continue`, instrução, **134**, 135, 153, 1069, LIV
`continue`, instrução rotulada, LIV
  terminando uma iteração única de uma rotulada - `for` instrução, LV
controlador (na arquitetura MVC), LXIX
controle de programa, **82**
controles, **419**
convergir para um caso de base, 591
conversão explícita, **98**
conversão implícita, **98**
conversão unboxing, **638**
converter
  entre sistemas numéricos, 534
  um número binário em decimal, VI
  um número hexadecimal em decimal, VI
  um número octal em decimal, VI
  um valor integral em um valor de ponto flutuante, 162
cookie, **971**, 977, 979
  cabeçalho, 971
  data de expiração, **971**
  domínio, 979
  exclusão, 971
  expiração, 971
`Cookie`, classe, 976
  `getDomain`, método, 979
  `getMaxAge`, método, 979
  `getName`, método, 979
  `getPath`, método, 979
  `getSecure`, método, 979
  `getValue`, método, 979
`Cookies`, array, 978
coordenada horizontal, **105**, **484**
coordenadas, 728
coordenadas (0, 0), 105, 484
coordenada vertical, **105**, **484**
coordenada *x* superior esquerda, 487
coordenada *y* superior esquerda, 487
coordenas em pixels, 729
cópia em profundidade, **298**
copiando objetos
  cópia em profundidade, **298**
  cópia superficial, **298**
cópia superficial, **298**, 299
cópia temporária, 97
`copy`, método da classe `Collections`, 645, **650**
cor, 484
cor de fundo, 489, 490
cores, 176
Cores aleatórias, exercício, 513
cor, manipulação, 484
corpo
  de uma declaração de classe, **31**
  de uma instrução `if`, 43
  de um loop, 88
  de um método, **31**
corpo da definição de método, **32**
correio eletrônico, 866
correspondência de arquivo
  com de serialização de objeto, exercício, 587
  com múltiplas transações, exercício, 587
  exercício, 586
  programa, 586
correspondência de padrão, **904**
Corrida de cavalos, exercício, 766
corrigir em um sentido matemático, 140
`cos`, método de `Math`, 157
co-seno, 157
co-seno trigonométrico, 157
Cozinhando com ingredientes mais sadios, 549
`-cp`, opção de linha de comando para `java`, **267**
CPU, **4**
CraigsList (`www.craigslist.org`), 18, XXVI
craps (jogo de dados de cassino), 168, 188, 231
`createGlue`, método da classe `Box`, **792**
`createGraphics`, método da classe `BufferedImage`, **506**
`createHorizontalBox`, método da classe `Box`, **470**, 791
`createHorizontalGlue`, método da classe `Box`, **791**
`createHorizontalStrut`, método da classe `Box`, **791**
`createRealizedPlayer`, método da classe `Manager`, **758**
`createRigidArea`, método da classe `Box`, **791**
`createStatement`, método de `Connection`, **914**, 918
`createVerticalBox`, método da classe `Box`, **791**
`createVerticalGlue`, método da classe `Box`, **791**
`createVerticalStrut`, método da classe `Box`, **791**
cresça e brilhe, algoritmo, 82
Crescimento demográfico mundial, 118
criando e inicializando um array, 193
criando um banco de dados Java DB em NetBeans, 996
criando um objeto de uma classe, 59
criar uma classe reutilizável, 264
criar um aplicativo desktop em NetBeans, 1025
criar um aplicativo Web em NetBeans, 1020
criar um `Socket`, 867
criptografar, 118

critérios de seleção, **904**
Crivo de Eratóstenes, 234, **847**
Crivo de Eratóstenes, utilizando um BitSet, LI
cross-site scripting, 1030
CSS (Cascading Style Sheets)
  folha de estilo, 953, 957
  tutoriais (www.deitel.com/CSS21/), 860
ctrl, tecla, 131
<Ctrl>-d, 131
Ctrl, tecla, 447, 460
<Ctrl>-z, 131
curinga, 688
  argumento de tipo, **688**
  em um parâmetro de tipo genérico, **687**
  limite superior, 688
currentThread, método da classe Thread, 813
currentTimeMillis, método da classe System, 613
cursor, 31, 32
cursor de saída, 32, 34
cursor de tela, 35
cursos avançados de ciência da computação, xxvii
curto-circuito, avaliação, **137**
curva, 506, 724
curva complexa, 506

D

-d, opção de compilador, **265**
dados, **3**
dados compartilhados de acesso, 823
dados consistentes, 243
dados de template fixa, **951**
dados mutáveis, **817**
dados persistentes, **552**
Damas, jogo, 896
data, 163
data de expiração de um cookie, **971**
DatagramPacket, classe, **877**, 893
  getAddress, método, **879**
  getData, método, **879**
  getLength, método, **879**
  getPort, método, **879**
DatagramSocket, classe, **877**
  receive, método, **879**
  send, método, **879**
DataInput, interface, 577
DataInputStream, classe, **577**
DataOutput, interface, 577
  writeBoolean, método, 577
  writeByte, método, 577
  writeBytes, método, 577
  writeChar, método, 577
  writeChars, método, 577
  writeDouble, método, 577
  writeFloat, método, 577
  writeInt, método, 577
  writeLong, método, 577
  writeShort, método, 577
  writeUTF, método, 577
DataOutputStream, classe, **577**
Date, classe, **1096**
  exercício, 276
DateAndTime, classe, (exercício), 276
DBCS (double byte character set), **XXXI**
decimal (sistema numérico de base 2, II
decisão, **43**, 85
  símbolo (na UML), **85**
  símbolo na UML, 375
decisão lógica, 2
decisões, tomando, 48
declaração
  classe, **30**
  import, 37, 38
  método, **30**
declaração de interface, **322**
declaração de método, 159
declaração import do tipo por demanda, **266**
declaração import do tipo simples, **266**
declarar um método de uma classe, **58**
Decorator, padrão de design, LIX, LXI, LXVIII
default, caso em um switch, **132**, 133, 167
default, palavra-chave, 1069
definir uma área de desenho personalizada, 455
deitel@deitel.com, 3
Deitel Buzz Online, boletim, **19**
Deitel Buzz Online Newsletter (www.deitel.com/newsletter/subscribe.html), 3, 18
Deitel Resource Centers (www.deitel.com/ResourceCenters.html), 3, 18
Deitel, site Web (www.deitel.com), 2
delegar uma chamada de método, **706**
DELETE, instrução SQL, **909**
delete, método da classe StringBuilder, **530**
DELETE, SQL, instrução, 903
deleteCharAt, método da classe StringBuilder, **530**
del.icio.us, 17, XXVI
delimitador para tokens, **535**
demo, diretório, 726
demo de imagens gráficas, 726
demo de imagens gráficas bidimensionais, 726
demo de pacote de navegador, XXV
demo Wallpaper API, XXV
Department of Defense (DOD), 8
dependência de plataforma, 808
dependências entre capítulos, gráfico de, xxi, xxiv
dependente de estado, **818**
dependente de máquina, **5**
Deploy, opção, no NetBeans, 1022
Deposit, classe (estudo de caso ATM), 367, 369, 371, 372, 377, 383, 384, 388, 393, 395
DepositSlot, classe (estudo de caso ATM), 369, 372, 377, 384, 393
deprecated-list.html, gerado por javadoc, XLI
Deprecated, nota, **XL**
Deprecated, link na API, **1072**
@deprecated, tag javadoc, **XL**
depurador, 1078
  clear, comando, **1088**
  cont, comando, **1080**
  definição, **1078**
  erro de lógica, 1078
  exit, comando, **1084**
  -g, opção de compilador, **1079**
  inserindo pontos de interrupção, 1080
  jdb, comando, **1079**
  modo de interrupção, **1080**
  next, comando, **1084**
  ponto de interrupção, **1078**
  print, comando, **1081**, 1082
  run, comando, **1079**, 1081
  set, comando, **1081**, 1082
  step, comando, **1083**
  step up, comando, **1083**
  stop, comando, **1080**, 1081
  suspendendo a execução do programa, 1081
  unwatch, comando, 1086
  watch, comando, **1085**
depurar, **7**
descendente de uma fonte, **493**
descoberta automática de driver (JDBC 4), xxii, **913**
descobrindo um componente, 486
descrição de um serviço Web, **1024**
descriptografar, 118
desempenho da classificação e pesquisa na árvore binária, 721
desempilhando, **348**
desempilhar a pilha de chamadas de método, 348
desencadear um evento, 423
desenhando na tela, 484
desenhar imagens gráficas, 728
desenhar uma curva complexa, 724
desenhar um arco, 724
desenhar um retângulo, 732
desenho, cores, 487
desenho, formas, 484
desenho, linhas e pontos, 724
desenvolvimento de aplicativo Web, 947
desenvolvimento de GUIs para diversas plataformas, 423
desenvolvimento rápido de aplicativos Web, xxii
Designer de fogos de artifício, exercício, 766
Design, modo no NetBeans, **956**, 959, 996
design orientado a objetos (*object-oriented design* — OOD), **15**
Design, visualização no Netbeans, XIII
design padrão, LVIII, LIX, LXII, LXIII, LXIV, LXV, LXVII, LXXI
Desktop, classe, **XXII**
  browse, método, **XXII**
  getDesktop, método, **XXII**
  isDesktopSupported, método, **XXII**
  mail, método, **XXII**
  open, método, **XXII**
desktop, elemento de um documento JNLP, **735**
deslocamento de bits para a esquerda (<<), **XLIII**, XLIV
deslocamento para a direita com sinal (>>), **XLIII**, XLIV, XLIX
deslocamento para a direita sem sinal (>>>), XLIII
deslocar (números aleatórios), **165**
despachar uma thread, **806**
despacha um evento, 435
destroy, método
  de JApplet, **728**, 731
  JavaServer Faces, **956**, 958, 1004
Determinando pontos C e D para nível 1 de "Fractal de Lo", 602
diacrítico, **XXX**
diagrama da UML elidido, **368**
diagrama de atividades, 83, 85, 124, 140
  do...while, instrução, 128
  for, instrução, 124
  if...else, instrução, 85
  if, instrução, 85
  instrução sequencial, 83
  na UML, 89, **365**, 375, 389
  switch, instrução, 134
  while, instrução, 89
diagrama de atividades mais simples, 142
diagrama de atividades na estrutura de sequência, 83
diagrama de classes
  na UML, **365**, 368, 369, 371, 372, 377, 391, 393, 396, 397
  para o modelo do sistema ATM, 371, 387
diagrama de colaboração na UML, **366**, 384
diagrama de comunicação na UML, **366**, 384
diagrama de estado na UML, **374**
diagrama de estado para o objeto ATM, 374
diagrama de gráfico de estado do ciclo vida de threads, 805, 806
diagrama de interação na UML, **384**
diagrama de máquina de estado na UML, **365**, 374
diagrama de relacionamento de entidades, **902**

diagrama de sequência na UML, **366**, 384
Dialog, font, 492
DialogInput, font, 492
diálogo, **74**, **421**
diálogo de entrada, **74**, **421**
diálogo de mensagem, **74**, **421**, 422
  tipos, 422
diálogo de seleção de cor, 490
diálogo modal, 490
diâmetro, 53, 512, 741
Dica de desempenho, **7**
Dica de portabilidade, **7**
Dica de prevenção de erro, **7**
Dicas de desempenho, visão geral, xxv
dicas de ferramenta, **425**, 427
Dicas de portabilidade, visão geral, xxv
Dicas de prevenção de erro, visão geral, xxv
digitalização de imagens, 3
digit, método da classe Character, **533**
digitando um campo de texto, 428
dígito, 38, 533, 536, II
dígito binário (bit), **552**
dígito decimal, **553**
dígitos invertidos, 187
Dimension, classe, **752**
dir, comando no Windows, 724
DIRECTORIES_ONLY, constante de JFileChooser, **580**
diretiva, **951**
diretiva de imparcialidade de um bloqueio, **835**
diretivas de barra de rolagem, **470**
diretório, 555, 556
  árvore, 726
  nome, 555
  separador, **267**
diretório-pai, 556
diretório-raiz, **555**
diretório virtual, **948**
diretrizes de projeto GUI recomendadas utilizadas por GroupLayout, XIII
disco, 3, 10, 552
disco óptico, 552
disjoint, método da classe Collections, 645, **653**
dispose, método da classe Window, **772**
DISPOSE_ON_CLOSE, constante de interface WindowConstants, 772
dispositivo de armazenamento secundário, **552**
dispositivo eletrônico inteligente de consumo popular, 6
dispositivos de entrada, **3**
dispositivos de saída, **3**
dispositivos eletrônicos de consumo popular, 6
distância entre valores (números aleatórios), 167
distinção de letras maiúsculas e minúsculas, **30**
  comandos do Java, 12
distribuindo cartas, 199
DitherTest, applet, 724
dividir para conquistar, abordagem, **155**, 156
dividir por zero, 10, 337
dividir um array na classificação por intercalação, 628
divisão de inteiro, **41**
divisão de ponto flutuante, 98
divisão, operador de atribuição composta, /=, 102
divisão por zero, 94
DNS (*domain name system*), servidor, **948**
DO_NOTHING_ON_CLOSE, constante de interface WindowConstants, 772
do...while, instrução de repetição, 84, **127**, 128, 144, 1069
documentação, comentários, **XXXIV**
documentação da estrutura de coleções (java.sun.com/javase/6/docs/technotes/guides/collections/index.html), 637
Documentação da Java API
  download (java.sun.com/javase/downloads/), 11, 35, 40
  java.sun.com/javase/6/docs/api/, 11, 35, 39, **1071**
documentação de API (java.sun.com/javase/6/docs/api/), 164
documentar um programa, **29**
documento, 769, 784
documento de requisitos, 15, **361**, 364
dois maiores valores, 116
Dojo Toolkit, 995, 1012
-d, opção, **XL**
(double), coerção, **98**
Double, classe, **638**, 687
  parseDouble, método, 732
double, tipo primitivo, **38**, **70**, 95, 1069, 1070
  promoções, 162
Double Range Validator, componente JSF, **964**
doubleValue, método de Number, **688**
downcast, **307**, 319
draw3DRect, método da classe Graphics, 496, **498**
draw, método da classe Graphics2D, **505**
drawArc, método da classe Graphics, **224**, **498**, 512
drawImage, método da classe Graphics, **747**
drawLine, método da classe Graphics, **107**, 495
drawOval, método da classe Graphics, **144**, 498, 741
drawPolygon, método da classe Graphics, 501, **502**
drawPolyline, método da classe Graphics, 501, **503**
drawRect, método da classe Graphics, **144**, 495, 506, 512, 741
drawRoundRect, método da classe Graphics, **496**
drawString, método da classe Graphics, **488**, 728, 732
DrawTest, applet, 724, 726
driver, classe, 59
driver JDBC, **899**
DriverManager, classe, **913**
  getConnection, método, **913**
Dropcash, XXVII
Drop Down List, componente JSF, 961, **964**
dump de computador, 238
duplicata de datagrama, 877
duplo igual, ==, 45

E

EAST, constante
  da classe BorderLayout, **452**, 463
  da classe GridBagConstraints, **793**
eBay, XXVII
echo, caractere da classe JPasswordField, **429**
Eclipse
  vídeo de demonstração (www.deitel.com/books/jhtp8), 29
Eclipse Foundation, 18
Eclipse (www.eclipse.org), 9
ecoar um pacote de volta para o cliente, **877**
eco, caractere, **429**
Ecofont, 481
E condicional, &&, operador, **136**, 137
  tabela-verdade, 136
editar um programa, 9
editor, **8**
editor de texto, 32, 516
editor visual, 953
editor visual (NetBeans), **956**
efeito colateral, **137**
eficiência de
  classificação por flutuação, 634
  classificação por inserção, 628
  classificação por intercalação, 631
  classificação por seleção, 625
  pesquisa binária, 622
  pesquisa linear, 618
eficiente (princípio do projeto Unicode), **XXIX**
é um, relacionamento, **279**, 307
eixo *x*, **484**
eixo *y*, **484**
elemento chance, **164**
elemento de formulário oculto, 971
elemento de script, **951**
elemento de um array, **190**
elemento-raiz, 953
elemento-raiz (XML), **735**
elemento (XML), **735**
elemento XML vazio, 735
eliminação de duplicatas, **713**
Eliminação de duplicatas, exercício, 230
eliminar vazamentos de recurso, 346
Ellipse2D, classe, 484
Ellipse2D.Double, classe, **503**, 512
Ellipse2D.Float, classe, **503**
E lógico booleano, &, operador, 136, **137**
else oscilante, problema, **87**, 117
else, palavra-chave, 86, 1069
emacs, 8
embaralhamento e distribuição de cartas, 235
  por Collections método shuffle, 648
empacotador de sincronização, **663**
empacotador não modificável, **664**
empacotando objetos fluxo, **570**, 574
empacotar tipos de fluxo, 866, 867
empilhamento de instruções de controle, 85
empilhamento, regra, **142**
empilhando instruções de controle, 144
Employee, classe que implementa Payable, 325
Employee, programa de teste de hierarquia de classe, 317
Employee, superclasse abstrata, 312
EmptyStackException, classe, **655**
en.wikipedia.org/wiki/IP_address, 948
encapsulamento, **15**
Encontrar um valor mínimo em um array, exercício, 612
endereço de IP, 879, **948**
  do servidor, 876
endereço de IP (en.wikipedia.org/wiki/IP_address), 948
endereço de loopback, **871**
endereço Internet do servidor, 867
End, tecla, 458
endsWith, método da classe String, **521**
engenharia de software, 252
Enquete com alunos, exercício, 587
enqueue, operação de fila, **707**
ensureCapacity, método da classe StringBuilder, **527**
ENTERED, constante da classe aninhada EventType, **865**
Enter (ou Return), tecla, 32, 432, 726
entidade de segurança em um cálculo de juros, 125
entrada de dados, 74
entrada de dados a partir do teclado, 47
entrada/saída, 555
entrada/saída, operação, 236
entrada/saída, operações, 84
entrada/saída, pacote, 163
entrada única/saída única, instruções de controle de, 141

entrelinha de uma fonte, **493**
enum
  constante, 256
  EnumSet, classe, 257
  palavra-chave, 1069
  values, método, **257**
enum, palavra-chave, **171**
enumeração, **171**
EnumSet, classe, **257**
  java.sun.com/javase/6/docs/api/java/util/EnumSet.html, 257
  range, método, **257**
enviar dados a um servidor, 876
enviar mensagem, 67
enviar uma mensagem para um objeto, **57**
EOF (*end-of-file*), 867
EOFException, classe, **576**
equals, método
  da classe Arrays, **219**
  da classe BitSet, **LI**
  da classe Object, 298
  da classe String, 519, 520
equalsIgnoreCase, método da classe String, 519, 521
erasure, **675**, 677
erro de compilação, **30**
erro de compilador, **30**
erro de estouro, 239
erro de lógica, **9**, 39, **88**, 122, 1078
erro de lógica em tempo de execução, 39
erro de lógica fatal, **88**
erro de lógica não-fatal, **88**
Erro de programação comum, **7**
erro de sintaxe, **30**, 31, 32
erro de tempo de execução, **11**
erro em tempo de compilação, **30**
erro fatal, **11**, **88**, 239
erro não-fatal de tempo de execução, **11**
erro por um (off-by-one), **122**
Error, classe, **343**
Erros de programação comuns, visão geral, xxv
erro síncrono, **342**
E/S, aprimoramento do desempenho, 577
E/S armazenada em buffer, **577**
escalar, **203**
escalonando (números aleatórios), 165
escalonando uma imagem, 747
escape, propriedade de um componente Static Text, **965**
escopo, 123
escopo de aplicativo, **952**
escopo de sessão, **952**
escopo de solicitação, **952**
escopo de uma declaração, **172**
escopo de um parâmetro de tipo, 679
escopo de variável, **123**
Escrevendo o valor de um cheque por extenso, 547
esfera, 182
E sobre bits (&), **XLIII**
E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e operadores de complemento de bits, XLIII
espaçamento horizontal, **465**
espaçamento vertical, **465**
espaçar entre componentes em GroupLayout, XIII
espaço em branco, 30, 46
espaço em disco, 696
espaço, flag, **1099**, 1100
especialização, **279**
especialização na UML, **395**
especificação do projeto, **365**
especificadores de conversão de ponto flutuante, 1098
especificadores de formato, **1091**
  %%, 1097
  %.2f, para números de ponto flutuante com precisão, 98
  %b, 1096
  %B, 1096
  %b, para valores boolean, **138**
  %c, 54, 1094
  %d, **39**, 1091, 1092
  %e, 1093
  %E, 1093
  %f, 53, **71**, 1093
  %g, 1093
  %G, 1093
  %h, 1096
  %H, 1096
  %n, 1096
  %n, (separador de linha), **562**
  %o, 1092
  %s, 1091, 1094
  %S, 1094
  %x, 1092
  %X, 1092
especificidades, 307
espera, fila, 639, 655
espera sincronizada, estado, **806**
espera para uma nova conexão, 871
espiral, 513, 594
estação de trabalho, **4**
estado, **365**
estado *bloqueado*, **806**, 812
estado consistente, 248
estado da ação (na UML), **83**, 141, 375
estado de *espera*, **806**
estado de *execução*, **806**
estado de um objeto, 371, 374
estado *executável*, **805**, 806
estado final (na UML), **84**, 140, 375
estado inicial, 140
estado inicial (na UML), **84**, **374**, 375
estado *morto*, **806**
estado na UML, **365**, 374
estado *novo*, **805**
estado *pronto*, **806**
estender uma classe, **279**
estilo de fonte, 438, **492**
estouro, 342
estouro aritmético, 342
estouro de pilha, **162**
estrela de cinco pontas, 506
estrutura de aplicativo Web, 951
estrutura de coleções, **637**
estrutura de dados, **190**, LXVII
estrutura de dados bidimensional, 709
estrutura de dados dinâmica, **695**
estrutura de dados linear, **709**
estrutura de dados, primeiro a entrar, primeiro a sair (*first-in*, *first-out* — FIFO), **707**
estrutura de dados subjacente, 655
estrutura de dados último a entrar, primeiro a sair (*last-in-first-out* — LIFO), **704**
estrutura de um sistema, 374
estruturas de dados não-lineares, 696
estruturas de dados predefinidas, 637
estrutura vertical, **791**
Estudo de caso de engenharia de software, xxiv
estudos de caso, xxii
etapa de análise do ciclo de vida de software, 364
Euclides, algoritmo de, 187
Euler, 232
EventListenerList, classe, **434**
evento, 329, 374, **428**, 509
evento assíncrono, **342**
EventObject, classe
  getSource, método, 433
evento de gatilho pop-up, **779**, 781
evento de janela, **773**
evento de mouse, **435**, 449
evento externo, **449**
eventos de mouse, 781
EventType, classe aninhada
  ACTIVATED, constante, **865**
  ENTERED, constante, **865**
  EXITED, constante, **865**
EventType, classe aninhada de HyperlinkEvent, **865**
exceção, 199, **336**
exceção encadeada, **351**
exceção não-interceptada, **341**
exceção não-verificada, **343**
exceção verificada, **343**
Exception, classe, **343**
excluindo um item de uma árvore binária, 714
excluir uma pilha, **162**
exclusão mútua, **812**
execução sequencial, **83**
executando um aplicativo, 12
executando um AudioClip, 756
executar, 33
executar um applet em um navegador Web, 727, 730
execute, método
  de Executor, 811
  de JdbcRowSet, **925**
execute, método do Executor interface, **810**
executeQuery, método
  de PreparedStatement, **932**
  de Statement, **914**
executeUpdate, método da interface PreparedStatement, **932**
ExecutionException, classe, 842
Executor, interface, **810**
  execute, método, **810**, 811
Executors, classe, **810**
  newCachedThreadPool, método, **810**
ExecutorService, interface, **810**, 851
  awaitTermination, método, **814**
  shutdown, método, **811**
  submit, método, **851**
exibindo texto em uma caixa de diálogo, 74
exibir saída, 46
exibir uma linha de texto, 32
exists, método de File, 556
exit, comando de depurador, **1084**
exit, método da classe System, **345**, **561**
EXIT_ON_CLOSE, constante da classe JFrame, **107**
EXITED, constante da classe aninhada EventType, **865**
exp, método de Math, 157
exponenciação, 239
expressão, **39**
expressão condicional, **86**, 243
expressão de ação (na UML), **84**, 85, 375
expressão de acesso ao array, **190**
expressão de controle da instrução switch, **132**
expressão de criação de array, **191**
expressão de criação de instância de classe, **60**, 69
expressão integral, 134
expressão integral constante, **129**, 134
expressão regular, **536**, 541, 968
  ^, 538
  ?, 539
  {, *n* }, 539

{,*n*,}, 539
{,*n*,*m* }, 539
*, 539
\d, 536
\D, 536
\s, 536
\S, 536
\w, 536
\W, 536
+, 539
extends, palavra-chave, **107**, **282**, 289, 1069
extensibilidade, 307
Extensible HyperText Markup Language (XHTML), 947, 950
Extensible Markup Language (XML), **735**, 950, 1024
extensões de arquivo
.aif, **756**, 758
.aiff, **756**, 758
.au, **756**, 758
.avi, **758**
.class, 756
.gif, **744**
.jpeg, **744**
.jpg, **744**
.mid, **756**, 758
.mov, **758**
.mp3, **758**
.mpeg, **758**
.mpg, **758**
.png, **744**
.rmi, **756**, 758
.spl, **758**
.swf, **758**
.wav, **756**

F

fábrica, LXVIII
fábrica, método, LXIII
Facade, padrão de design, LIX, LXI, LXVIII
FaceBook, 18
fachada, objeto, LXVIII
factorial, método, 591, 592
Factory Method, padrão de design, LIX, LX, LXIII
Fahrenheit, 479, 1108
equivalente de uma temperatura em Celsius, 187
falando com um computador, 3
Falha de construtor, exercício, 359
false, palavra-chave, **43**, 86, 1069
f, opção do comando jar, **734**
fase, 94
fase de implementação, 398
fase de inicialização, 94
fase de processamento, 94
F, sufixo para literais float, **655**
fator de carga, **659**
fator de escalonamento (números aleatórios), 165, 167
fatorial, 118, 151, 591
f:view, elemento, **953**
fazer sua pontuação (jogo de dados *craps*), 168
fechar (ocultar) um diálogo, **422**
fechar uma janela, 425, 428
feedback visual, 438
ferramenta de desenvolvimento, 724
ferramenta de desenvolvimento de programas, 85, 95
fibonacci, método, 595
Fibonacci, série, 235
Fibonacci, série de, **594**, 595
definido recursivamente, 594
gerada com um método recursivo, 595
fila, 639, 655, **695**, 707
fila de prioridade de múltiplos níveis, **808**
fila para o servidor, 871
File, classe, **555**
canRead, método, 556
canWrite, método, 556
exists, método, 556
File, métodos, 556
getAbsolutePath, método, 556
getName, método, 556
getParent, método, 556
getPath, método, 556
isAbsolute, método, 556
isDirectory, método, 556
isFile, método, **556**
lastModified, método, 556
length, método, 556
list, método, 556
toURI, método, **761**
utilizado para obter informações sobre arquivos e diretórios, 556
File, documentação de classe (java.sun.com/javase/6/docs/api/java/io/File.html), 556
File, métodos, 556
File.pathSeparator, 558
FileContents, interface, 751
getLength, método, **746**
FileExplorer, demo, XXV
FileInputStream, classe, **555**, 570, 572, 574, 576, 663
FileNotFoundException, classe, **561**
FileOpenService, interface, **743**, 746
openFileDialog, método, **746**
openMultiFileDialog, método, **751**
FileOutputStream, classe, **555**, 570, 572, 663, LXVIII
FileReader, classe, **555**, 578
FILES_AND_DIRECTORIES, constante de JFileChooser, **580**
FILES_ONLY, constante de JFileChooser, **580**
FileWriter, classe, **555**, 578
filho direito, **709**
filho esquerdo, **709**
fill3DRect, método da classe Graphics, 496, **498**
fill, método
da classe Arrays, **219**, 220
da classe Collections, 645, **650**
da classe Graphics2D, **505**, 506, 508, 513
fillArc, método da classe Graphics, 223, **224**, **498**
fillOval, método da classe Graphics, **176**, **457**, 496, **498**
fillPolygon, método da classe Graphics, 501, **503**
fillRect, método da classe Graphics, **176**, **487**, 495, 506
fillRoundRect, método da classe Graphics, **496**
FilterInputStream, classe, **577**
FilterOutputStream, classe, **577**
filtrar um fluxo, **577**
fim da entrada de dados, 93
fim de fluxo, 867
fim de linha (de uma única linha), comentário, //, **29**, 31
fim do arquivo (*end-of-file* — EOF), 867
fim do arquivo (*end-of-file* — EOF)
indicador, **131**
fim do arquivo (end-of-file — EOF)
combinações de teclas, 562
marcador, **554**
final
classe, **321**
método, **321**
palavra-chave, 134, **158**, 195, 262, 321, 817, 1069
variável, 195
variável deve ser inicializada, 263
variável local, 445
final, Classes e métodos
java.sun.com/docs/books/tutorial/java/IandI/final.html, 321
www.ibm.com/developerworks/java/library/j-jtp1029.html, 321
final, palavra-chave, 158
final de uma fila, **695**, 707
finalize, método, **258**, 298
finally
bloco, **341**, 345, 349, 838
cláusula, **345**, 1069
palavra-chave, **341**
find, método da classe Matcher, **541**
Firefox, navegador Web, 74
fireTableStructureChanged, método de AbstractTableModel, **919**
firewall, **1019**
first, método de interface SortedSet, **658**
fita magnética, 552
flags, 1091, 1099
Flasher de Imagem, exercício, 765
Flasher de texto, exercício, 765
Flickr, 17, XXVI, XXVII
APIs, XXVI
float
promoções de tipos primitivos, 162
sufixo de literal F, **655**
tipo primitivo, **38**, 1069, 1070
Float, classe, **638**
float, tipo primitivo, **70**
Floating Dock, demo, XXV
floor, método de Math, 157
FlowLayout, classe, **427**, 461
CENTER, constante, **463**
LEFT, constante, **463**
RIGHT, constante, **463**
setAlignment, método, **463**
flush, método
da classe BufferedOutputStream, **577**
da classe Formatter, **887**
da classe ObjectOutputStream, **871**
fluxo, **347**, 1091
fluxo baseado em bytes, **554**
fluxo baseado em caracteres, **554**
fluxo, cabeçalho, **872**
fluxo, comunicações baseadas em, **859**
fluxo de bytes, **554**
fluxo de controle, 89, 97
fluxo de controle na instrução if...else, 86
fluxo de entrada padrão (System.in), 554
fluxo de erro padrão, **341**, 347, **1091**
fluxo de erro padrão (System.err), 554, 577
fluxo de saída padrão, **347**
fluxo de saída padrão (System.out), 554, 577
fluxo de trabalho, **83**
fluxo de trabalho de um objeto na UML, **375**
fluxo, processamento, 552
fluxos, **859**
fluxo, socket, **859**, 867, 882
fluxos, transmissão baseada em, 877
Flyweight, padrão de design, LXI
foco, **429**
foco para um aplicativo GUI, 769, 781
FocusAdapter, classe, 453
FocusListener, interface, 453
folha, LXIV

folha de estilo, 953, 958
font
  estilo, **491**
  nome, **491**
  tamanho, **491**
Font, classe, **440**, **484**, 491
  BOLD, constante, **491**, 492
  getFamily, método, 491, **493**
  getName, método, 491, **492**
  getSize, método, 491, **492**
  getStyle, método, 491, **493**
  isBold, método, 492, **493**
  isItalic, método, 492, **493**
  isPlain, método, 492, **493**
  ITALIC, constante, **491**, 492
  PLAIN, constante, **491**, 492
fonte
  manipulação, 484
fonte de evento, **432**, 434
fontes do Java
  Dialog, 492
  DialogInput, 492
  Monospaced, 492
  SansSerif, 492
  Serif, 492
FontMetrics, classe, **484**, 494
  getAscent, método, 494
  getDescent, método, 494
  getFontMetrics, método, **494**
  getHeight, método, 494
  getLeading, método, 494
for aninhado, instrução, 196, 211, 212, 215, LIV
fora dos limites de um array, 199
for, instrução de repetição, 84, 121, 122, 123, 124, 125, 126, 144, 1069
  aninhado, 196
  diagrama de atividades, 124
  exemplo, 124
for, instrução de repetição aprimorada, **202**
for, propriedade de um campo de entrada JSF, 965
força bruta, 233, 234
  Passeio do cavalo, 233
forDigit, método da classe Character, **533**
forma, 503
forma animada, 726
forma de linha reta (expressões aritméticas em Java), **41**
forma preenchida, 176, 506
formas, 724
formas bidimensionais, 484
formas dimensionadas aleatoriamente, 513
formatação
  exibição de dados formatados, 35
formatação de data, 1091
formatação de data/hora, 1091
formatação de inteiro decimal, 39
format, método
  da classe Formatter, 562, **1104**
  da classe String, **75**, 1104
format, método da classe String, **244**
Formatando data e hora com caractere de conversão t, 1096
Formatar saída com a classe Formatter, 1104
formato de data/hora universal, 242, 243, 244
formato de relógio de 24 horas, 242
formato exponencial, 1091
formato padrão de data/hora, 244
formato, string, **1091**, 1098
formato tabular, 193
forma tridimensional, 725
Formatter, classe, **555**, 560, **1091**, 1103
  close, método, **562**
  documentação (java.sun.com/javase/6/docs/api/java/util/Formatter.html), 1094, 1104
  flush, método, **887**
  format, método, **562**, 1104
  toString, método, 1104
FormatterClosedException, classe, **562**
formulando algoritmos, 93
formulários virtuais, 998, **1006**
  participante, **1007**
  submissor, **1006**
fornecedor de software independente (ISV), 297
Fortran (FORmula TRANslator), **7**
fração de tempo, **806**
fracionamento do tempo, 807
fractal, **600**
  curva de Koch, **601**
  exercícios, 612
  floco de neve de Koch, **601**
  "fractal de Lo" no nível, 602, 603
  "fractal de Lo" no nível 1, com C e D pontos determinados para nível 2, 602
  "fractal de Lo" no nível 2, linhas tracejadas no nível 1 fornecido, 602
  fractal estritamente auto-similar, **600**
  nível, **601**
  ordem, **601**
  profundidade, **601**
  propriedade auto-similar, **600**
Fractal, applet, 724
fractal da Curva de Koch, **601**
fractal de floco de Neve de Koch, **601**
"fractal de Lo" no nível 0, 601
"fractal de Lo" no nível 1, com C e D pontos determinados para nível 2, 602
"fractal de Lo" no nível 2, 603
"fractal de Lo" no nível 2, linhas tracejadas no nível 1 fornecido, 603
fractal estritamente auto-similar, **600**
Fractal, interface com usuário, 603
Frame, classe, 772
frame-alvo, **863**
  _blank, **863**
  _self, **863**
  _top, **863**
frame interno
  que pode ser fechado, 786
  que pode ser maximizado, 786
  que pode ser minimizado, 786
  que pode ser redimensionado, 786
frame interno maximizado, 786
frame (na UML), 385
framework, **951**
frase com verbo no documento de requisitos, 377
FreeTTS (freetts.sourceforge.net/docs/index.php), 762
frequency, método da classe Collections, **653**
frequency, método de Collections, 645
FROM, cláusula de SQL, 903
fromJson, método
  de classe Gson, **1034**
função, **16**, **156**
função de retorno de chamada, **1006**
funcionalidade de negócios (mashups), XXVII
Future, interface, **851**
Future Splash (.spl), formato de arquivo, **758**

G

gabinete de arquivos, 726
Gamma, Erich, LVIII
"Gang of Four", LVIII, LIX, LX, LXI
GCD (máximo divisor comum), 611
generalidades, 307
generalização na UML, **395**
GeneralPath, classe, 484, **506**, 512
  closePath, método, **508**
  lineTo, método, **508**
  moveTo, método, **508**
genéricos, xxi, 638, **671**
  ? (argumento de tipo curinga), **688**
  argumento de tipo curinga, **688**
  argumentos reais de tipo, **674**
  classe parametrizada, **678**
  colchetes angulares (< e >), **674**
  curingas, **687**, 688
  curinga sem um limite superior, 689
  erasure, **675**
  escopo de um parâmetro de tipo, 679
  limite superior de um curinga, 688
  limite superior de um parâmetro de tipo, **676**, 677
  limite superior padrão (Object) de um parâmetro de tipo, 679
  método, 673
  parâmetro formal de tipo, **674**
  seção de parâmetro de tipo, **674**
  tipo parametrizado, **678**
  tipo, parâmetro, **674**
  tipo, variável, **674**
Gerador de palavras cruzadas, 549
gerador de palavras para números de telefone, exercício, 587
Gerando e Percorrendo um Labirinto, exercício, 766
Gerando labirintos aleatoriamente, exercício, 613
gerenciador de layout, **427**, 452, 460, 465, XII
  BorderLayout, **452**
  FlowLayout, 427
  GridLayout, **466**
Gerencia programas de publicidade do Google AdWords, XXVII
gestão de relacionamento com o cliente (*customer relationship management* — CRM), XXVII
getAbsolutePath, método da classe File, 556
getActionCommand, método da classe ActionEvent, **433**, 438
getAddress, método da classe DatagramPacket, **879**
get, método
  da classe ArrayList<T>, **222**
  da classe BitSet, **LI**
  de interface Future, **851**
  de interface List, **641**
  de interface Map, **661**
  de interface MessageContext, 1037
@GET, anotação, **1029**
getAppletContext, método
  da classe Applet, **863**
getAscent, método da classe FontMetrics, 494
get, solicitação, 949
GET, solicitação HTTP, **948**
getAttribute, método da interface HttpSession, **1037**
getAudioClip, método da classe Applet, **756**
getAutoCommit, método da interface Connection, **939**
getBlue, método da classe Color, **487**, 489
getByName, método da classe InetAddress, **876**
getChars, método
  da classe String, **518**
  da classe StringBuilder, **528**
getClass, método da classe Object, **320**, **427**
getClass, método de Object, 298
getClassName, método da classe StackTraceElement, **351**

getClassName, método da classe UIManager.LookAndFeelInfo, **784**
getClickCount, método da classe MouseEvent, **455**
getCodeBase, método da classe Applet, **756**
getColor, método da classe Color, **487**
getColor, método da classe Graphics, 487
getColumnClass, método de TableModel, **915**, 919
getColumnClassName, método de ResultSetMetaData, **919**
getColumnCount, método de ResultSetMetaData, **914**, 919
getColumnCount, método de TableModel, **915**, 919
getColumnName, método de ResultSetMetaData, **919**
getColumnName, método de TableModel, **915**, 919
getColumnType, método de ResultSetMetaData, **914**
getConnection, método de DriverManager, **913**
getContentPane, método da classe JFrame, **447**
getControlPanelComponent, método da interface Player, **760**
getData, método da classe DatagramPacket, **879**
getDefaultSystemTray, método da classe SystemTray, **XXIV**
getDelay, método da classe Timer, **765**
getDescent, método da classe FontMetrics, 494
getDesktop, método da classe Desktop, **XXII**
getEventType, método da classe HyperlinkEvent, **865**
getFamily, método da classe Font, 491, **493**
getFileName, método da classe StackTraceElement, **351**
getFont, método da classe Graphics, 492
getFontMetrics, método da classe FontMetrics, **494**
getFontMetrics, método da classe Graphics, 494
getGreen, método da classe Color, **487**, 489
getHeight, método da classe Component, **747**
getHeight, método da classe FontMetrics, 494
getHeight, método da classe JPanel, **107**
getHostName, método da classe InetAddress, **871**
getIcon, método da classe JLabel, **428**
getIconHeight, método da classe ImageIcon, **747**
getIconWidth, método da classe ImageIcon, **747**
getImage, método da classe ImageIcon, **747**
get, método, **66**, 248, 252
getInetAddress, método da classe Socket, **871**
getInputStream, método da classe Socket, **866**, 867
getInstalledLookAndFeels, método da classe UIManager, **784**
getInstance, método de Calendar, **1096**
getInt, método de ResultSet, **914**
getKeyChar, método da classe KeyEvent, **460**
getKeyCode, método da classe KeyEvent, **460**
getKeyModifiersText, método da classe KeyEvent, **460**
getKeyText, método da classe KeyEvent, **460**
getLeading, método da classe FontMetrics, 494
getLength, método da classe DatagramPacket, **879**
getLength, método da interface FileContents, **746**
getLineNumber, método da classe StackTraceElement, **351**
getLocalHost, método da classe InetAddress, **876**, 879
getMaximumSize, método da classe Component, XIII
getMessage, método da classe Throwable, **349**
getMethodName, método da classe StackTraceElement, **351**
getMinimumSize, método da classe Component, **752**, 772, XIII
getModifiers, método da classe InputEvent, **460**
getName, método da classe Class, 298, **320**
getName, método da classe File, 556
getName, método da classe Font, 491, **492**
getObject, método da interface ResultSet, **914**, 926
getOptions, método da interface AutoComplete, **1010**, 1012
getOutputStream, método da classe Socket, **866**
getParameter, método da classe Applet, **862**
getParent, método da classe File, 556
getPassword, método da classe JPasswordField, **433**
getPath, método da classe File, 556
getPoint, método da classe MouseEvent, **456**
getPort, método da classe DatagramPacket, **879**
getPreferredSize, método da classe Component, **752**, 772, XIII
getProperty, método da classe Properties, **661**
getRed, método da classe Color, **487**, 489
getRequestContext, método da interface BindingProvider, **1045**
getResource, método da classe Class, **427**
getRow, método da interface ResultSet, **919**
getRowCount, método da interface TableModel, **915**, 919
getSelectedFile, método da classe JFileChooser, **581**
getSelectedIndex, método da classe JComboBox, **445**
getSelectedIndex, método da classe JList, **447**
getSelectedText, método da classe JTextComponent, **470**
getSelectedValues, método da classe JList, **449**
getSession, método da interface HttpSession, **1037**
getSize, método da classe Font, 491, **492**
getSource, método da classe EventObject, 433
getStackTrace, método da classe Throwable, **349**
getStateChange, método da classe ItemEvent, **445**
getStyle, método da classe Font, 491, **493**
getText, método da classe JLabel, **428**
getText, método da classe JTextComponent, 779
gettingreal.37signals.com/toc.php, 19
getURL, método da classe HyperlinkEvent, **865**
getValue, método da classe JSlider, **772**
getValueAt, método da interface TableModel, **915**, 919
getVisualComponent, método da interface Player, **760**
getWidth, método da classe Component, **747**
getWidth, método da classe JPanel, **107**
getX, método da classe MouseEvent, **452**
getY, método da classe MouseEvent, **452**
.gif, extensão de arquivo, **744**
.gif, extensão de arquivo, 427
GIF (Graphics Interchange Format), 427, 744
G.I.M.P., 743
Givezilla, XXVIII
GlassFish, servidor de aplicativo, **947**, 951, 960
  glassfish.dev.java.net, 1021
  Tester, página Web, 1023
glifo, **XXX**
Google, **17**
  AdWords, XXVII
  Blog Search, **18**
  Mapas, **18**, XXVI
  Maps, XXVI
-g, opção de linha de comando para javac, 1079
Gosling, James, 6
goto, eliminação, 83
goto, instrução, **83**
grade, 466
grade para gerenciador de layout GridBagLayout, 792
gradiente, **505**
gradiente acíclico, **505**
gradiente cíclico, **505**
GradientPaint, classe, **484**, 505, 512
gráfico, 152
gráfico de barras, 152, 196, 724
gráfico de dependências entre capítulos, xxi, xxiv
gráfico de pizza, exercício, 513
gráfico de torta, 513
gráfico, informações, 196
Gráficos de Tartaruga, 231, 512
Gráficos de tartaruga, exercício, 512
Grand, Mark, LXVII
Graphics2D, classe, **484**, 503, 505, 508, 512
  draw, método, **505**
  fill, método, **505**, 506, 508, 513
  rotate, método, **508**
  setPaint, método, **505**
  setStroke, método, **505**
  translate, método, **508**
Graphics, classe, **105**, 176, 223, 330, **456**, **484**, 502, 728, 730, 732, 747
  clearRect, método, 496
  draw3DRect, método, 496, **498**
  drawArc, método, **498**, 512
  drawImage, método, **747**
  drawLine, método, **107**, 495
  drawOval, método, 496, **498**, 741
  drawPolygon, método, 501, **502**
  drawPolyline, método, 501, **503**
  drawRect, método, 495, 506, 512, 741
  drawRoundRect, método, **496**
  drawString, método, **488**, 728, 732
  fill3DRect, método, 496, **498**
  fillArc, método, **498**
  fillOval, método, **176**, **457**, 496, **498**
  fillPolygon, método, 501, **503**
  fillRect, método, **176**, **487**, 495, 506
  fillRoundRect, método, **496**
  getColor, método, 487
  getFont, método, 492
  getFontMetrics, método, 494
  setColor, método, **176**, 506
  setFont, método, **492**
Graphics Interchange Format (GIF), **427**, **744**
graphicssoft.about.com/od/pixelbasedfreewin, 762
GraphicsTest, applet, 724
GraphLayout, applet, 724
grau, 498
graus negativos, **498**
graus positivos, **498**
gravável, 556
GridBagConstraints, classe, **792**, 796
  anchor, campo, **793**
  BOTH, constante, **793**
  CENTER, constante, **793**
  EAST, constante, **793**
  gridheight, campo, **793**
  gridwidth, campo, **793**
  gridx, campo, **793**
  gridy, campo, **793**
  HORIZONTAL, constante, **793**
  NONE, constante, **793**
  NORTH, constante, **793**
  NORTHEAST, constante, **793**
  NORTHWEST, constante, **793**
  RELATIVE, constante, **796**
  REMAINDER, constante, **796**
  SOUTH, constante, **793**
  SOUTHEAST, constante, **793**
  SOUTHWEST, constante, **793**
  variáveis de instância, 792
  VERTICAL, constante, **793**
  weightx, campo, **793**
  weighty, campo, **793**
  WEST, constante, **793**
GridBagConstraints, constantes RELATIVE e REMAINDER, 796

GridBagLayout, classe, 789, 792, 796
  setConstraints, método, **796**
GridBagLayout, gerenciador de layout, 794
gridheight, campo da classe GridBagConstraints, **793**
GridLayout, classe, 461, **466**
GridLayout, contendo seis botões, 466
Grid Panel, componente JSF, **963**
gridwidth, campo da classe GridBagConstraints, **793**
gridx, campo da classe GridBagConstraints, **793**
gridy, campo da classe GridBagConstraints, **793**
GROUP BY, 903
group, método da classe Matcher, **542**
GroupLayout, classe, 460, **XII**
  BASELINE, constante de alinhamento, XIII
  CENTER, constante de alinhamento, XIII
  diretrizes de design GUI recomendadas, XIII
  espaçamento entre componentes, XIII
  gerenciador de layout padrão no Netbeans, **XII**
  grupos, **XIII**
  layout paralelo de componentes GUI, XII
  layout sequencial de componentes GUI, XII
  LEADING, alinhando componentes, XIII
  LEADING, constante de alinhamento, XIII
  orientação horizontal sequencial, **XII**
  TRAILING, constante de alinhamento, XIII
GroupLayout.Group, classe, **XIII**
  addGap, método, **XIII**
GroupLayout.ParallelGroup, classe, **XIII**
  addGap, método, **XIII**
GroupLayout.SequentialGroup, classe, **XIII**
  addGap, método, **XIII**
grupos em GroupLayout, **XIII**
Gson, classe, 1033
  code.google.com/p/google-gson/, 1032
  Json, método, **1034**
  toJson, método, 1033
guardando código com um bloqueio, **812**
Guarded Suspension, padrão de design, LIX, LXVII
Guestbook, aplicativo, 1016
Guestbook, aplicativo, exercício, 1016
GUI portável, 164
GUIs (graphical user interfaces), 329
  componente, **419**
  ferramenta de design, **460**

H

h, caractere de conversão, **1096**
H, caractere de conversão, **1096**
handler de evento, 329, **428**
handler de exceção, **340**
handler de exceção padrão, 350
handler do formulário do lado do servidor, **949**
handshake, ponto, **866**, 876
hardcopy, impressora, 10
hardware, **3**, 5
hash bucket, 659
hashCode, método de Object, 299
hashing, **659**
HashMap, classe, **658**, 862, 976, 982, 983
  get, método, 976
  keySet, método, **661**, 985
  put, método, 976
HashSet, classe, **656**
hash, tabela, 656, 659
Hashtable, classe, **658**, 659, LXXI
hasNext, método
  da classe Scanner, **131**, 562
  de interface Iterator, **641**, 643
hasPrevious, método de interface ListIterator, **643**
headSet, método da classe TreeSet, **658**
height, atributo do elemento applet-desc, 736
height de um applet em pixels, **729**
Helm, Richard, LVIII
helpdoc.html, gerado por javadoc, **XLII**
Help, link na API, **1072**
herança, **15**, 107, **279**, 395, 397, 398
  exemplos, 280
  extends, palavra-chave, **282**, 289
  hierarquia, **280**, 311
  hierarquia MembrosDaComunidade da universidade, 280
  múltipla, 279
  único, **279**
herdar, **107**
heurística, 233
heurística de acessibilidade, 233
hidden
  campo, 971
  elemento, 971
HIDE_ON_CLOSE, constante de interface WindowConstants, 772
hierarquia de coleções, 639
hierarquia de dados, 553
Hierarquia de formas, exercício, 334
hierarquia de herança, **280**
hipotenusa de um triângulo reto, 186
Hiragana, bloco, XXXIII
Home, chave, 458
HORIZONTAL, constante da classe GridBagConstraints, **793**
HORIZONTAL_SCROLLBAR_ALWAYS, constante da classe JScrollPane, **470**
HORIZONTAL_SCROLLBAR_AS_NEEDED, constante da classe JScrollPane, **471**
HORIZONTAL_SCROLLBAR_NEVER, constante da classe JScrollPane, **471**
horizontal JSlider, componente, 769
host, **948**
host de serviço Web, **1019**
hostname, **948**
HourlyEmployee, classe derivada de Employee, 314
HousingMaps.com (www.housingmaps.com), 18
href, atributo do elemento jnlp, 735
.htm, extensão de nome de arquivo, **729**
.html, extensão de nome de arquivo, **729**
HTML, tutoriais (www.deitel.com/xhtml/), 723, 860
HTTP, códigos de status (www.w3.org/Protocols/rfc2616/rfc2616-sec10.html), 948
HTTP (Hypertext Transfer Protocol), 860, 971
  cabeçalho, **949**
  em uso com firewalls, 1019
  método, **948**
  protocolo sem estado, 971
  solicitação, tipo, **949**
  transação, 948
HttpServletRequest, classe, 978
HttpServletResponse, classe, 976
HttpSession, interface, 1037
  getAttribute, método, **1037**
  getSession, método, **1037**
  setAttribute, método, **1037**
HugeInteger, Classe, 276
  exercício, 276
hyperlink, 863, **865**
HyperlinkEvent, classe, **863**, 865
  EventType, classe aninhada, **865**
  getEventType, método, **865**
  getURL, método, **865**
HyperlinkListener, interface, **865**
  hyperlinkUpdate, método, **865**
Hyperlink, componente JSF, 961, **964**
hyperlinkUpdate, método da interface HyperlinkListener, **865**
Hypertext Transfer Protocol (HTTP), **860**, 948, 971

I

IBM, computador pessoal, 4
IBM Corporation, 4, 7, XXIX
IBM, Enterprise Mashup Tool Enterprise da, XXVIII
Icon, interface, **427**
ícone, 422
ícones de bandeja, **XXIV**
ID de evento, **435**
IDE, **9**
identificador, **30**, 36
identificador válido, 38
IDENTITY, palavra-chave (SQL), **928**
IDEs
  NetBeans, 1019
IEEE 754 (grouper.ieee.org/groups/754/), 1070
IEEE 754, ponto flutuante, 1070
if...else aninhado, instrução de seleção, **86**, 87
if...else, instrução de seleção dupla, 84, **85**, 97, 129, 144
  diagrama de atividades, 85
if, instrução de seleção única, **43**, 84, 85, 129, 144, 1069
  diagrama de atividades, 85
ignorando elemento zero, 199
IllegalMonitorStateException, classe, **825**, 836
IllegalStateException, classe, **565**
Image, classe, **744**
ImageIcon, classe, **300**, **427**, 744, 751
  getIconHeight, método, **747**
  getIconWidth, método, **747**
  getImage, método, **747**
  paintIcon, método, **751**
imagem, 730, 743, 761
ImageMap, applet, 724
imagens gráficas, 455, 724, 726, 743
imagens gráficas bidimensionais, 503
imagens gráficas coloridas, de alta resolução e tridimensionais, 743
imagens gráficas em uma maneira independente de plataforma, 484
ImageObserver, interface, **747**
Image, componente JSF, 961, **963**
impasse, **836**, 839, 856, LXVII
implantando um serviço Web, 1022
implementação abstrata, 664
implementação de coleção, 664
implementação de uma função, 313
implementar múltiplas interfaces, **449**
implementa uma interface, **306**, 322, 327
implements, 1069
implements, palavra-chave, **322**, 325
Impondo privacidade com criptografia, 118
import, declaração, **37**, 38, 63, 264, 1069
imprimindo árvores, 720
imprimindo uma árvore binária em um formato de árvore de duas dimensões, 714
imprimir em múltiplas linhas, 35
imprimir uma linha de texto, **32**
imprimir um array, 612
Imprimir um array de para trás para frente, exercício, 612
Imprimir um array, exercício, 612
imprimir um array recursivamente, 612
inanição, **808**
incluir na pilha, operação, 701

incrementar
  uma variável de controle, 121
incremento, 125
  de uma instrução for, 123
  de uma variável de controle, **120**
  expressão, 135
  operador, ++, 102
incremento e decremento, operadores, 102
index-all.html, gerado por javadoc, **XLII**
index.html, gerado por javadoc, XLI
indexOf, método da classe ArrayList<T>, 221
indexOf, método da classe String, **522**
IndexOutOfBoundsException, classe, 650
indicador de classe JSlider, **769**
índice de argumentos, 1091, **1096**, 1102
índice de array fora dos limites, 199, 342
índice de massa corpórea (IMC), 26
  calculadora, 26
índice de um JComboBox, **444**
índice (subscript), **190**
índice zero, **190**
InetAddress, classe, **871**, 876, 879, LXVIII
  getByName, método, **876**
  getHostName, método, **871**
  getLocalHost, método, **876**, 879
informações de caminho, **555**
informações de nível de classe, **258**
informações persistentes, 971
informações sobre fontes, 484
informações voláteis, **3**
information, elemento de um documento JNLP, **735**
inicialização no começo de cada repetição, 100
inicializador de array, **193**
  aninhado, **210**
  para array multidimensional, 210
inicializadores de array aninhados, 210
inicializando arrays bidimensionais nas declarações, 211
inicializar um applet, 730
inicializar uma variável em uma declaração, **38**
inicializar variáveis de instância do applet, 732
iniciar uma ação, 773
init, método, **956**, 958
  de JApplet, **728**, 730, 732
initComponents, método autogerado em Netbeans, XVI
_init, método, 956, 976
injeção de dependência, **1037**
INNER JOIN, cláusula de SQL, 903, **907**
input, 971
InputEvent, classe, **450**, 454, 458
  getModifiers, método, **460**
  isAltDown, método, 454, **460**
  isControlDown, método, **460**
  isMetaDown, método, 454, **460**
  isShiftDown, método, **460**
InputMismatchException, classe, **338**, 340
InputStream, classe, 570, **576**, 663, 866, 867
  read, método, **746**
InputStreamReader, classe, **578**
inserir caracteres literais na saída, 1091
INSERT, instrução SQL, **908**
insert, método da classe StringBuilder, **530**
INSERT, SQL, instrução, 903
instanceof, operador, **319**, 1069
instância de uma classe, 64
instância (não- static) método, 259
instanciar, **16**
instanciar um objeto de uma classe, **58**
instrução, **32**, 59
instrução auxiliada por computador (*computer-assisted instruction* — CAI), 188
  Monitorando o desempenho do aluno, 188
  Níveis de dificuldade, 188
  Reduzindo a fadiga do aluno, 188
  Variando os tipos de problema, 188
instrução de atribuição, 39
instrução de controle, **82**, 84, 85, 597
  aninhamento, **84**
  empilhamento, **84**
instrução de declaração de variável, **38**
instrução rotulada, **LIV**
Instruções, 94
  aninhadas, **99**
  break, **132**, 134, 135, 153
  break rotulada, LIV
  continue, **134**, 153, LIV
  continue rotulada, **LV**
  de controle aninhadas, **99**, 142, 144
  de controle de entrada única/saída única, **84**
  de controle empilhadas, 141
  do...while, 84, **127**, 128, 144
  empilhamento de instruções de controle, **84**
  for, 84, **121**, 122, 123, 124, 126, 144
  for aprimorada, **202**
  if, **43**, 84, 85, 129, 144
  if...else, 84, **85**, 86, 97, 129, 144
  if...else aninhadas, **86**, 87
  instrução de controle, **82**, 84, 85
  instrução vazia, 88
  loop, **84**
  problema dos resultados do exame com instruções de controle aninhadas, 100
  profundamente aninhadas, 143
  repetição, **83**, 84, **88**
  return, 156, 161
  seleção, **83**, **84**
  seleção dupla, **84**, 99
  seleção múltipla, **84**
  seleção única, **84**
  switch, 84, **129**, 133, 144
  switch, instrução de múltipla seleção, 167
  vazias (um ponto-e-vírgula, ;), **46**, 88, 129
  while, 84, 89, 92, 97, 120, 144
instruções de controle aninhadas, 167, LIV
int, tipo primitivo, **38**, 95, 102, 129, 1069, 1070
  promoções, 162
Integer, classe, 219, 422, **638**, 687
  parseInt, método, 219, 422
  toBinaryString, método, **XLV**
integerPower, método, 186
integração com o desktop, 733
integridade de dados, 253
inteiro, **36**
  array, 193
  divisão, **93**
  quociente, 41
  valor, 38
inteiro binário, 117
inteiro decimal, 1092
inteiro hexadecimal, 1092
inteiro octal, 1092
inteiros
  sufixo L, 655
inteiros alinhados à direita, 1098
Intel, 758
interação entre um cliente de serviço Web e um serviço Web, 1025
interações entre objetos, 382, 384
intercalação na UML, 375
intercalar dois arrays, 628
interface, **15**, 306, 322, 327, **914**
  declaração, **322**
  implementando mais de uma por vez, **452**
  interface de tags, **570**
interface, palavra-chave, **322**, 1069
interface com o usuário, 950, LXI
interface de múltiplos documentos (*multiple document interface* — MDI), 769, **784**
interface de ponto de extremidade de serviço (SEI), **1022**
interface de ponto de extremidade de serviço (*service endpoint interface* — SEI), 1022, 1024
interface de programas aplicativos (*applications programming interface* — API), **7**, **155**
interface de tags, 322, **570**
interface genérica, **675**
interface gráfica com usuário (*graphical user interface* — GUI), **74**, 163, **419**
  componente, **74**
  ferramenta de design, **460**
Interfaces, **321**
  ActionListener, **432**, 435
  AppletContext, **860**
  AudioClip, **756**
  BlockingQueue, **823**
  CachedRowSet, **924**, 998
  Callable, **851**
  CallableStatement, **938**
  ChangeListener, **772**
  CharSequence, **541**
  Collection, 637, **639**, 645
  Comparable, 329, 521, **645**, **675**, 712
  Comparator, **645**
  ComponentListener, 453
  Condition, **836**
  Connection, **913**, 914, 918
  ContainerListener, 453
  DataInput, 577
  DataOutput, 577
  Executor, **810**
  ExecutorService, **810**, 851
  FileOpenService, **743**, 746
  FocusListener, 453
  Future, **851**
  HyperlinkListener, **865**
  Icon, **427**
  ImageObserver, **747**
  ItemListener, **440**, 778
  Iterator, **639**
  JdbcRowSet, **924**
  KeyListener, **435**, 453, 458, 459
  LayoutManager2, **463**
  LayoutManager, **460**, 463
  List, 637, **643**
  ListIterator, **639**
  ListSelectionListener, **447**, 863
  Lock, **835**
  Map, 637, 658
  MessageContext, **1037**
  MouseInputListener, **449**, 452
  MouseListener, **435**, 449, 452, 781
  MouseMotionListener, **435**, 449, 452
  MouseWheelListener, **450**
  ObjectInput, **570**
  ObjectOutput, **570**
  Player, **758**
  PreparedStatement, 938
  PropertyChangeListener, **850**
  Queue, 637, **639**, 655, 823
  RequestContext, 1045
  ResultSet, **914**
  ResultSetMetaData, **914**
  RowSet, **924**
  Runnable, 329, **808**, 888

Serializable, 329, **570**
Servlet, **950**
Set, 637, **639**, 656
SortedMap, **658**
SortedSet, **657**
Statement, 914
SwingConstants, 329, **428**, 772
TableModel, **915**
WebServiceContext, **1037**
WindowConstants, **772**
WindowListener, 452, **773**, **924**
interfaces com usuário baseadas em multithreading, xxiv
internacionalização, 163
internalVirtualForms, propriedade de um Table, **998**
Internet, **5**, 860
Internet Explorer, 74
Internet, telefonia, 18
interpretador, **6**
Interpreter, padrão de design, LXII
interrupt, método da classe Thread, **809**
InterruptedException, classe, **809**
interseção de dois conjuntos, 276
interseção teórica, 276
intervalo de sono, **806**
introdução
java.sun.com/new2java/, 8
Introdução ao Java (java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/GettingStartedTOC.fm.html), 914
invocar um método, **67**, 156
invokeLater, método da classe SwingUtilities, **871**
IOException, classe, **574**
isAbsolute, método de File, 556
isActionKey, método da classe KeyEvent, **460**
isAltDown, método da classe InputEvent, 454, **460**
isBold, método da classe Font, 492, **493**
isCancelled, método da classe SwingWorker, **847**
isControlDown, método da classe InputEvent, **460**
isDefined, método da classe Character, **533**
isDesktopSupported, método da classe Desktop, **XXII**
isDigit, método da classe Character, **533**
isDirectory, método de File, 556
isEmpty, método
ArrayList, 253
Map, **661**
Stack, **655**
isFile, método de File, **556**
isItalic, método da classe Font, 492, **493**
isJavaIdentifierPart, método da classe Character, **533**
isJavaIdentifierStart, método da classe Character, **533**
isLetter, método da classe Character, **533**
isLetterOrDigit, método da classe Character, **533**
isLowerCase, método da classe Character, **533**
isMetaDown, método da classe InputEvent, 454, **460**
isPlain, método da classe Font, 492, **493**
isPopupTrigger, método da classe MouseEvent, **781**
isRunning, método da classe Timer, **751**
isSelected, método
AbstractButton, **777**
JCheckBox, **440**
isShiftDown, método da classe InputEvent, **460**
isUpperCase, método da classe Character, **533**
ITALIC, constante da classe Font, **491**, 492
item de menu, **773**, 777
ItemEvent, classe, **440**, 442
getStateChange, método, **445**
ItemListener, interface, **440**, 778
itemStateChanged, método, 440, 779
itemStateChanged, método da interface ItemListener, 440, 779
iteração, **92**, 597
de um loop, **120**, 135
iterador, **637**
iterador bidirecional, **643**
Iterator, interface, **639**
hasNext, método, **641**
next, método, **641**
remove, método, **641**
iterator, método de interface Collection, **641**
Iterator, padrão de design, LIX, LXII, LXXI

J

Jacobson, Ivar, 16
Jacopini, G., 83, 144
janela, 74, 105, 107, **772**
janela de comando, 32, 724, 725, 729
janela de terminal, **32**
janela, eventos, **773**
janela filha, 769, 786, 787
janela filha em uma interface de múltiplos documentos, **784**
janela pai, 74, 422, 769
janela pai em uma interface de múltiplos documentos, **784**
janela pai especificada como nulo, 777
janela pai para uma caixa de diálogo, **777**
janelas, sistema, **424**
JApplet, classe, **728**, 730, 773
destroy, método, **728**
init, método, **728**, 732
paint, método, **728**, 732
start, método, **728**, 732
stop, método, **728**
jar, comando, **734**
c, opção, **734**
f, opção, **734**
v, opção, **734**
jar, elemento de um documento JNLP, **736**
JAR, arquivo, 747, 753
Java2D, applet, 726
Java2D, diretório, 726
Java2D API, 484, **503**, 726, 743
Java 2D, documentação (java.sun.com/javase/6/docs/technotes/guides/2d), 503
Java 2D, formas, 503
Java 3D, 767
Java 3D API, **743**, 762
Java 3D, documentação (java.sun.com/javase/technologies/desktop/java3d/), 743
java.applet, pacote, 163
java.awt, classe, 772
java.awt.color, pacote, 503
java.awt.event, pacote, 163, 433, 452, 460
java.awt.font, pacote, 503
java.awt.geom, pacote, 163, 503
java.awt.image, pacote, 503
java.awt.image.renderable, pacote, 503
java.awt, pacote, 163, **423**, 486, 500, 503, 744, 752, 781
java.awt.print, pacote, 503
java.beans, pacote, 850
Java Abstract Window Toolkit (AWT), pacote, **163**
Java Abstract Window Toolkit Event, pacote, **163**
java.com, 723
java, comando, **10**, 12, **29**
-splash, opção, **XXI**
Java Advanced Imaging API, **743**
java, elemento de um documento JNLP, **736**
java, interpretador, 33
java.io, documentação de pacote (java.sun.com/javase/6/docs/api/java/io/package-tree.html), 555
java.io, hierarquia de pacote (java.sun.com/javase/6/docs/api/java/io/package-tree.html), 555
java.io, pacote, 163, **555**
java.lang, pacote, **38**, 157, 163, 282, 298, 516, 808, XL
importado em cada programa Java, 39
java.math, pacote, 70, 593
Java, ambiente de desenvolvimento, 9, 724
java.net, pacote, 163, **859**
Java API, **155**, 329
documentação (java.sun.com/javase/6/docs/api/index.html), 11
visão geral (java.sun.com/javase/6/docs/api/overview-summary.html), 164
Java API Interfaces, 329
Java, applet, 728
Java Applet Package, **163**
Java Application Programming Interface (Java API), **7**, **37**, **155**, 163
Java Architecture for XML Binding (JAXB), 1029
Java archive (JAR), arquivo, 734
java.sql, pacote, 163, 913
java.swing, pacote, 728
java.text, pacote, 163
java.util.concurrent.locks, pacote, 835, 836
java.util.concurrent, pacote, 164, 810, 823, 851
java.util, pacote, **37**, 163, 221, 638, 654, 687, 696, 1096
Calendar, classe, **1096**
Date, classe, **1096**
java.util.prefs, pacote, **661**
java.util.regex, pacote, 516
JavaBean, **951**, 952, 953
Java, biblioteca de classe, **7**, **37**
javac, compilador, 33
Java, compilador, **10**
Java Concurrency Package, **164**
Java Database Connectivity (JDBC), **899**
Java DB, xxii, 899, **926**, 995
Developer's Guide (developers.sun.com/docs/javadb/10.4.2.0/devguide/derbydev.pdf), 928
Java, depurador, 1078
javadoc, opções
-author, **XL**
-d, **XL**
-link, **XL**
javadoc, programa utilitário, **30**
javadoc, tag, **XXXIV**
javadoc, tags
{@link}, **XL**
@author, **XXXVII**
@deprecated, **XL**
@param, **XXXVIII**
@return, **XXXIX**
@see, **XXXVII**
@since, **XL**
@throws, **XXXVIII**
@version, **XL**
Javadoc, comentário, 30, 947
Java EE 5, xxii, xxvii, 951
java.sun.com/javaee, 950
Java Enterprise Edition (Java EE), **3**, 950
.java, extensão, **10**
.java, extensão de nome de arquivo, 58
Java Foundation Classes (JFC), **423**
Java HotSpot, compilador, **10**

Java Como Programar, 8/*E*
recursos para o aluno, xxvi
recursos para o professor, xxvii
Java How to Program, 8/*E*
site Web (www.deitel.com/books/jhtp8/), 2
Java Image I/O API, **743**
Java Input/Output Package, **163**
Java Language Package, **163**
Java Language Specification, capítulo sobre arrays (java.sun.com/docs/books/jls/third_edition/html/arrays.html), 299
Java Media Framework (JMF)
API, 758
download (java.sun.com/javase/technologies/desktop/media/jmf/2.1.1/download.html), 758
java.sun.com/javase/technologies/desktop/media/jmf/, 758
Java Media Framework (JMF), API, **743**, 756
Java Media Framework Package, **164**
Java Micro Edition (Java ME), **3**
Java, navegador da Web compatível com, 723
Java Networking Package, **163**
Java Network Launch Protocol (JNLP), 723, 733, **734**
Java Plug-in, **723**
Java, repositório de aparência e funcionamento (java.sun.com/developer/techDocs/hi/repository), 762
Java, repositório de imagens gráficas de aparência e comportamento, 762
Java Resource Centers (www.deitel.com/ResourceCenters.html), 33
JavaScript, 952
JavaScript assíncrono e XML (Ajax), **1005**
JavaScript Object Notation (JSON), 1020
Java SE 6, xxii, xxvii
documentação de API (java.sun.com/javase/6/docs/api/), 164
Visão geral do pacote (java.sun.com/javase/6/docs/api/overview-summary.html), 164
Java SE Development Kit (JDK), 8, 30, 33
JavaServer Faces (JSF), xxii, **951**
Button, componente, 961, **964**
componentes, **952**, 954
Drop Down List, componente, 961, **964**
for, propriedade de um campo de entrada, 965
Grid Panel, componente, **963**
Hyperlink componente, 961, **964**
Image, componente, 961, **963**
Label, componente, 961, **964**
Message, componente, **964**
Message Group, componente, **999**
Meta, componente, 960
Radio Button Group, componente, 961, **964**
Static Text, componente, **953**, 961
Table componente, **995**, 997
Text Area, componente, **978**
Text Field, componente, 961, **964**
JavaServer Faces (JSF), elementos
webuijsf:staticText, 1001
webuijsf:table, 1001
webuijsf:tableColumn, 1001
webuijsf:tableRowGroup, 1001
JavaServer Faces (JSF) Expression Language, **953**
JavaServer Pages (JSP), 951
ação, **951**
declaração XML, 953
diretiva, **951**
elemento de script, **951**
elemento-raiz, 953
xmlns, atributos, **953**
JavaServer Pages Standard Tag Library (JSTL), **951**
Java, site Web (java.sun.com), 164
Java Sound API, **743**, 762
Java Speech API, **743**, 762
Java Speech API (java.sun.com/products/java-media/speech), 762
Java Standard Edition 6 (Mustang), xxvi
Java Standard Edition (Java SE), **3**
Java Swing Event Package, **164**
Java Swing GUI Components Package, **164**
Java, tecnologias de desktop (java.sun.com/javase/technologies/desktop/), 423
Java Text Package, **163**
Java™ Language Specification (java.sun.com/docs/books/jls/), 41
Java Utilities Package, **163**
Java Virtual Machine (JVM), **10**, 11, 29
Java Web Start, 723, **733**
atualização automática, 733
integração com o desktop, 733
javaws, comando, 736
visão geral (java.sun.com/javase/6/docs/technotes/guides/javaws/), 737
javax.faces.validators, pacote, **964**
javax.jnlp, pacote, 734, 743, 746
javax.media, pacote, 164, **758**
javax.servlet.http, pacote, **950**, 976, 978
javax.servlet.jsp, pacote, 951
javax.servlet.jsp.tagext, pacote, 951
javax.servlet, pacote, **950**
javax.sql.rowset, pacote, **924**
javax.swing.event, pacote, 164, **433**, 447, 452, 772
javax.swing, pacote, **74**, 164, 421, **423**, 427, 433, 435, 470, 486, 728, 744, 772, 784, 786
javax.swing.table, pacote, 915, **923**
JAXB, classe, **1029**
marshal, método, **1030**
unmarshal, método, **1031**
JAXB (Java Architecture for XML Binding), **1029**
JAX-RS, **1018**
JAX-WS, **1018**, 1025
JAX-WS, pacote, **164**
JBoss Application Server (www.jboss.com/products/platforms/application), 1021
JBuilder (www.codegear.com), 9
JButton, classe, 423, **436**, 437, 465
JCheckBox, botões e eventos de item, 438
JCheckBox, classe, 423, **438**
isSelected, método, **440**
JCheckBoxMenuItem, classe, 773, 778
JColorChooser, classe, **489**, 491
showDialog, método, **490**
JColorChooser, Exercício de diálogo, 513
JComboBox, classe, 423, **443**, 793
getSelectedIndex, método, **445**
setMaximumRowCount, método, **444**
JComboBox, que exibe uma lista de nomes de imagens, 443
JComponent, classe, 425, 427, 434, 443, 445, 455, 467, 484, 486, 752, LXIV
documentação (java.sun.com/javase/6/docs/api/javax/swing/JComponent.html), 425
paintComponent, método, **107**, **455**, 484, 751, 770, 772
repaint, método, **486**
setForeground, método, **778**
setOpaque, método, **455**, 457
setToolTipText, método, **427**
JCreator (www.jcreator.com), 9
jdb, comando, **1079**
JDBC
API, **899**, 911, 938
driver, 899
JDBC 4, xxii
jdbc:mysql://localhost/books, 913
JDBC, documentação (java.sun.com/javase/technologies/database/index.jsp), 900
JDBC, drivers (devapp.sun.com/product/jdbc/drivers), 900
JDBC, drivers (developers.sun.com/product/jdbc/drivers/), 900
JDBC, informações (java.sun.com/javase/technologies/database/index.jsp), 900
JDBC, pacote, **163**
JdbcRowSet, interface, **924**
close, método, **926**
execute, método, **925**
setCommand, método, **925**
setPassword, método, **925**
setUrl, método, **925**
setUsername, método, **925**
JdbcRowSetImpl, classe, **925**
JDesktopPane, classe, **784**, 802
JDesktopPane, documentação (java.sun.com/javase/6/docs/api/javax/swing/JDesktopPane.html), 787
JDialog, classe, **778**
JDIC Incubator Projects, XXV
JDIC (Java Desktop Integration Components)
addTrayIcon, método da classe SystemTray, **XXIV**
browse, método da classe Desktop, **XXII**
demo de pacote de navegador, XXV
Demos, XXV
demo Wallpaper API, XXV
Desktop, classe, **XXII**
FileExplorer, demo, XXV
Floating Dock, demo, XXV
getDefaultSystemTray, método da classe SystemTray, **XXIV**
getDesktop, método da classe Desktop, **XXII**
ícones de bandeja, **XXI**
isDesktopSupported, método da classe Desktop, **XXII**
JDIC Incubator Projects, XXV
mail, método da classe Desktop, **XXII**
open, método da classe Desktop, **XXII**
pacote de demonstração TrayIcon, XXIV
removeTrayIcon, método da classe SystemTray, **XXIV**
-splash, opção de linha de comando para o comando java, **XXI**
SplashScreen, classe, **XXI**
SystemTray, classe, **XXIV**
tela de splash, **XXI**
TrayIcon, classe, **XXIV**
JDK, 8, 33
demo, diretório, 723, 726
JEditorPane, classe, **863**
setPage, método, **865**
Jesse James Garrett, 1005
JFC (Java Foundation Classes), 423
JFileChooser, classe, **578**
CANCEL_OPTION, constante, **581**
FILES_AND_DIRECTORIES, constante, **580**
FILES_ONLY, constante, **580**
getSelectedFile, método, **581**
setFileSelectionMode, método, **580**
showOpenDialog, método, **581**
JFileChooser, diálogo, 578
JFrame, classe, **107**, 177, 772
add, método, **107**, **427**
EXIT_ON_CLOSE, **428**
getContentPane, método, **447**

setDefaultCloseOperation, método, **107**, **428**, **772**
setJMenuBar, método, **778**
setSize, método, **107**, **428**
setVisible, método, **107**, **428**
JFrame, classe EXIT_ON_CLOSE, constante, **107**
JFrame.EXIT_ON_CLOSE, **428**
jGRASP (www.jgrasp.org), 9
JInternalFrame, classe, 784
documentação (java.sun.com/javase/6/docs/api/javax/swing/JInternalFrame.html), 787
JIT, compilador, **10**
JLabel, classe, **299**, 300, 423, **425**
documentação (java.sun.com/javase/6/docs/api/javax/swing/JLabel.html), 425
getIcon, método, **428**
getText, método, **428**
setHorizontalAlignment, método, **428**
setHorizontalTextPosition, método, **428**
setIcon, método, **427**
setText, método, **428**
setVerticalAlignment, método, **428**
setVerticalTextPosition, método, **428**
JList, classe, 423, 445
addListSelectionListener, método, **447**
getSelectedIndex, método, **447**
getSelectedValues, método, **449**
setFixedCellHeight, método, **449**
setFixedCellWidth, método, **449**
setListData, método, **449**
setSelectionMode, método, **447**
setVisibleRowCount, método, **446**
JMenu, classe, **773**, 776, 786
add, método, **777**
addSeparator, método, **778**
JMenu, e mnemônicos, 773
JMenuBar, classe, 773, 778, 786
add, método, **778**
JMenuItem, classe, **773**, 786
JMF (Java Media FrameWork) API, 758
JNLP, 746, 747, 752
FileOpenService, **743**, 746
main-class, **734**
ServiceManager, classe, **746**
jnlp, elemento de um documento JNLP, **735**
codebase, atributo, 735
href, atributo, 735
jnlp.jar, 747, 752
JNLP, documentação (java.sun.com/javase/6/docs/jre/api/javaws/jnlp), 746
JNLP, documento, 734
applet-desc, elemento, **736**
application-desc, elemento, 736
desktop, elemento, **735**
information, elemento, **735**
jar, elemento, **736**
java, elemento, **736**
jnlp, elemento, **735**
offline-allowed, elemento, **735**
resources, elemento, **736**
shortcut, elemento, **735**
title, elemento, **735**
vendor, elemento, **735**
JNLP (Java Network Launch Protocol), 734
jogo, 164
Jogo de bilhar, exercício, 766
jogo de dados, 168
Jogo de dados (*craps*), 231
jogos de cartas, 199
Johnson, Ralph, LVIII
JOIN_ROUND, constante da classe BasicStroke, **506**
Joint Photographic Experts Group (JPEG), **427**, **744**
JOptionPane, classe, 74, **421**, 422, 799
constantes para diálogos de mensagens, 423
documentação (java.sun.com/j2se/1.5.0/docs/api/javax/swing/JOptionPane.html), 422
documentação (java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html), 422
PLAIN_MESSAGE, constante, **422**
showInputDialog, método, **74**, **422**
showMessageDialog, método, **74**, **422**
JOptionPane, constantes para diálogos de mensagens
JOptionPane.ERROR_MESSAGE, 423
JOptionPane.INFORMATION_MESSAGE, 423
JOptionPane.PLAIN_MESSAGE, 423
JOptionPane.QUESTION_MESSAGE, 423
JOptionPane.WARNING_MESSAGE, 423
JPanel, classe, **105**, 107, 423, 455, 461, 467, 748, 770, LXIV
getHeight, método, **107**
getWidth, método, **107**
JPasswordField, classe, **429**, 433
getPassword, método, **433**
.jpeg, extensão de nome de arquivo, 427
.jpeg, extensão de arquivo, **744**
JPEG (Joint Photographic Experts Group), 427, 744
.jpg, extensão de nome de arquivo, 427
.jpg, extensão de arquivo, **744**
JPopupMenu, classe, **779**
show, método, **781**
JProgressBar, classe, 848
JRadioButton, classe, **438**, 440, 442
JRadioButtonMenuItem, classe, 773, 778
JScrollPane, classe, 447, 470
HORIZONTAL_SCROLLBAR_ALWAYS, constante, **470**
HORIZONTAL_SCROLLBAR_AS_NEEDED, constante, **471**
HORIZONTAL_SCROLLBAR_NEVER, constante, **471**
setHorizontalScrollBarPolicy, método, **470**
setVerticalScrollBarPolicy, método, **470**
VERTICAL_SCROLLBAR_ALWAYS, constante, **470**
VERTICAL_SCROLLBAR_AS_NEEDED, constante, **471**
VERTICAL_SCROLLBAR_NEVER, constante, **471**
JScrollPane, diretivas de barra de rolagem, 470
JSF (JavaServer Faces), **951**
JSlider, classe, **769**, 771, XIII
documentação (java.sun.com/javase/6/docs/api/javax/swing/JSlider.html), 772
getValue, método, **772**
incremento de bloco, **769**
marcador, **769**
marcas de medida menores, 769
marcas de medida significativas, 769
marcas de verificação, **769**
setInverted, método, **770**
setMajorTickSpacing, método, **772**
setPaintTicks, método, **772**
tiques de aderência, **769**
Json, método da classe Gson, **1034**
JSON (JavaScript Object Notation), **1020**
JSON (www.json.org), 1020
jsp:root, elemento, **953**
JSP, contêiner, **951**
.jsp, extensão de nome do arquivo, **952**
JSP (JavaServer Pages), **951**
JSTL (JavaServer Pages Standard Tag Library), **951**
JTabbedPane, classe, **787**, 792
addTab, método, **789**
SCROLL_TAB_LAYOUT, constante, **792**
TOP, constante, **792**
JTable, classe, **915**
classificando e filtrando, xxiv
RowFilter, **924**
setRowFilter, método, **924**
setRowSorter, método, **923**
TableRowSorter, **923**
JTextArea, classe, 459, **468**, 469, 793, 795
setLineWrap, método, **470**
JTextArea não editável, 468
JTextComponent, classe, **429**, 431, 468
getSelectedText, método, **470**
getText, método, 779
setDisabledTextColor, método, **460**
setEditable, método, **431**
setText, método, 470
JTextField, classe, 423, **429**, 431, 433, 468
addActionListener, método, **432**
JTextField, e JPasswordFields, 429
JToggleButton, classe, **438**
JumpingBox, applet, 724
junções de linha, **505**
juros compostos, 125, 152
just-in-time, compilação, **10**
just-in-time (JIT), compilador, **10**

K

Kelvin, temperatura, escala, 479
KeyAdapter, classe, 453
KeyEvent, classe, **435**, 458
getKeyChar, método, **460**
getKeyCode, método, **460**
getKeyModifiersText, método, **460**
getKeyText, método, **460**
isActionKey, método, **460**
KeyListener, interface, **435**, 453, 458
keyPressed, método, **458**, 460
keyReleased, método, **458**
keyTyped, método, **458**
Keypad, classe (estudo de caso ATM), 365, 367, 369, 377, 382, 383, 384, 385, 393, 394, 416
keyPressed, método da interface KeyListener, **458**, 460
keyReleased, método da interface KeyListener, **458**
keySet, método
da classe HashMap, **661**, 985
da classe Properties, 663
keyTyped, método da interface KeyListener, **458**
KIS ("keep it simple"), 11
Koenig, Andrew, 336

L

Label, componente JSF, 961, **964**
Labirintos de qualquer de tamanho, exercício, 613
Lady Ada Lovelace, 8
LAMP, **19**
Lançamento de dados, 231
lançar uma exceção, **337**, 340
largura, 495
largura de arco e altura de arco para retângulos arredondados, 498
largura de campo, **126**, 1091, 1098
largura de um retângulo em pixels, 487
last, método da classe SortedSet, **658**
last, método de ResultSet, **919**
lastIndexOf, método da classe String, **522**
lastModified, método da classe File, 556
L, sufixo para long literais, **655**
Latim de porco, 546
layout, **300**
layoutContainer, método da interface LayoutManager, **463**

layout de página, software, 516
LayoutManager2, interface, **463**
LayoutManager, interface, **460**, 463
  layoutContainer, método, **463**
layout padrão do painel de conteúdo, 470
layout paralelo de componentes GUI, XII
layout sequencial de componentes GUI, XII
LEADING, constante de alinhamento em GroupLayout, XIII
Lea, Doug, LXVII
Lebre e a tartaruga, A, 234, 513
Lebre e a tartaruga, A, exercício, 513
LEFT, constante da classe FlowLayout, **463**
legibilidade, 29, 100
Leilões, XXVII
Leis de Morgan, 152
lendo um arquivo em um servidor Web, 863
length, campo de um array, 191
length, método da classe String, **518**
length, método da classe StringBuilder, **527**
length, método de File, 556
length, variável de instância de um array, **191**
Length Validator, 999
Length Validator, componente JSF, **964**, 968
letra, **553**
letra maiúscula, 30, 38
letra minúscula, 30, 553
liberar um bloqueio, 827
liberar um recurso, 345
LIFO (*last-in*, *first-out* - último a entrar, primeiro a sair), **162**, 681
ligando o servidor a uma porta, **866**, 877
LIGHTWEIGHT_RENDERER, constante da classe Manager, **759**
LIKE, cláusula de SQL, 905, 906
LIKE, operador (SQL), **904**
Limericks, 546
limericks aleatórios, 546
Limericks, exercício, 765
limite de crédito numa conta-corrente, 115
limite superior, 676
  de um curinga, 688
limite superior de um parâmetro de tipo, **676**, 677
limite superior padrão (Object) de um parâmetro de tipo, 679
Line2D, classe, 484, **506**
Line2D.Double, classe, **503**, 512
LinearGradientPaint, classe, 505
LineNumberReader, classe, **578**
lineTo, método da classe GeneralPath, **508**
linguagem assembly, **5**
linguagem de alto nível, **5**
linguagem de logotipo, 231
linguagem de máquina, **5**
linguagem de máquina, programação, 236
linguagem de modelagem gráfica (UML), xxiv
linguagem de programação procedural, **16**
linguagem extensível, **17**, **60**
linguagem híbrida, 6
linguagem natural de um computador, 5
linguagem orientada a objetos, **16**
linguagens fortemente tipadas, **105**
linha, 484, 495, 500, 902, 904, 905, 908
linha de base da fonte, 492
linha de comando, **32**
linha de comando, argumento, **158**, **218**, 219
linha de vida de um objeto em um diagrama de sequência da UML, **385**
linha em branco, 30, 95
linha em uma tabela de banco de dados, **900**
linha pontilhada (na UML), **84**
Linhas aleatórias utilizando a classe Line2D.Double, exercício, 512
linhas a serem recuperadas, 903
linhas conectadas, 501
linhas de guia (Netbeans), **XIV**, XV
linhas de um array bidimensional, 209
linha separadora em um menu, 778
linhas espessas, 503
linhas tracejadas, 503
link, 696, 709
{@link}, tag javadoc, **XL**
link de árvore na API, **1072**
link de índice na API, **1072**
-link, opção, **XL**
LinkedList, classe, **639**, 651, 669
  add, método, **644**
  addFirst, método, **644**
  addLast, método, **644**
links da API
  Deprecated, **1072**
  Help, **1072**
  Índice, **1072**
  Tree, **1072**
Linux, 5, 8, **19**, 32, 562, 723
lista, 443
lista de argumentos, 1092
lista de argumentos de comprimento variável, **217**
lista de compras, 88
lista de parâmetros, **61**, 68
lista de parâmetros de métodos, 217
lista de seleção múltipla, **445**, 447
lista de seleção única, **445**
lista drop-down, 423, **443**
listagens classificadas, 17
lista inicializadora, **193**
List, interface, 637, 639, **643**, 645, 650
  addAll, método, **643**
  add, método, **641**, 643
  clear, método, **643**
  get, método, **641**
  listIterator, método, **643**
  size, método, **641**, 643
  subList, método, **643**
  toArray, método, **644**
list, método da classe Properties, **663**
list, método de File, 556, 558
  dos argumentos, **36**, 38
  dos parâmetros, 159
listas indexadas, 721
lista vinculada, **695**, 696, LXXI
lista vinculada individualmente, **697**
ListIterator, interface, **639**
  hasPrevious, método, **643**
  previous, método, **643**
  set, método, **643**
listIterator, método de interface List, **643**
ListSelectionEvent, classe, **445**
ListSelectionListener, interface, **447**, 863
  valueChanged, método, **447**
ListSelectionModel, classe, **447**
  MULTIPLE_INTERVAL_SELECTION, constante, 447
  SINGLE_INTERVAL_SELECTION, constante, **447**
  SINGLE_SELECTION, constante, **447**
literais
  ponto flutuante, **70**
literal de caractere, **516**
literal de ponto flutuante, **70**
  double, por padrão, 70
literal de string, **516**
Live-Code, abordagem, xxv
load, método da classe Properties, **663**
local, 163
localhost, 948
localhost, (127.0.0.1), endereço, **871**
localização, **425**, **XXIX**
localização de uma variável na memória do computador, 40
Lock, interface, **835**
  lock, método, **835**, 838
  newCondition, método, **836**, 837
  unlock, método, **835**, 838
lock, método da interface Lock, **835**, 838
log, método de Math, 157
logaritmo, 157
logaritmo natural, 157
lógica da apresentação, **950**
lógica do controlador, **950**
lógica do negócio, **950**
loja física, 970
long
  sufixo de literal L, **655**
Long, classe, **638**
long, palavra-chave, 1069, 1070
long, promoções, 162
Long Range Validator, componente JSF, **964**
LookAndFeelInfo, classe aninhada da classe UIManager, **784**
lookingAt, método da classe Matcher, **541**
lookup, método da classe ServiceManager, **746**
loop, 89, 90, 92, 94, 765
  aninhado dentro de um loop, 99
  condição de continuação, 84
  contador, 120
  corpo, 127
  infinito, **89**, 97
  instrução, **84**, **88**
loop, método da interface AudioClip, **756**
loop, condição de continuação, **120**, 121, 122, 123, 124, 127, 128, 135, 199
loop infinito, **89**, 97, 123, 879, 882
Lord Byron, 8
losango (na UML), **83**
losango na UML, 153
losangos sólidos (representando composição) na UML, **369**
losangos vazios (representando agregação) na UML, **369**
lote, arquivo, **562**
Lovelace, Ada, 8
ls, comando no UNIX, 724

M

Macintosh, 484
Macintosh AIFF, formato de arquivo (extensões .aif ou aiff), 758
Macintosh, aparência e comportamento, 781
Mac OS X, 5, 8, 32, 562
Macromedia Flash 2, filmes (.swf), 758
Macromedia Flash, filmes (.swf), **758**
mail, método da classe Desktop, **XXII**
main-class, atributo do elemento applet-desc, 736
main-class especificada em um documento JNLP, **734**
main, método, 31, 33, 60
maiúsculas de fim (de uma linha), **505**
MalformedURLException, classe, **863**
Malha, jogo, exercício, 766
Manager, classe, **758**
  createRealizedPlayer, método, **758**
  LIGHTWEIGHT_RENDERER, constante, **759**
  setHint, método, **759**
Mandelbrot, Benoit, 600

manipulação de bits, XLIII
manipulação de fonte, 484
manutenciabilidade, 695
mapa de imagem, 724, 743, **753**
Map, interface, 637, **658**
  containsKey, método, **661**
  get, método, **661**
  isEmpty, método, **661**
  put, método, **661**
  size, método, **661**
Mapas, XXVII
mapeamentos de tipos SQL para tipos Java, 914
mapeando APIs, XXVI
Máquina Analítica, 8
máquina virtual (VM), **10**
marcado para coleta de lixo, 261
marcador de visibilidade na UML, **391**
marcas de medida em um JSlider, **769**
marcas de medidas secundárias da classe JSlider, 769
marcas de tique importantes da classe JSlider, 769
margem de uma janela, **772**
marshal, método da classe JAXB, **1030**
mashup, aplicativo, XXVI
mashups, xxii, **18**
Mashups
  Agendamento de evento, XXVII
  Agregação de ouvintes de evento, XXVIII
  Amazon, XXVII
  Amazon Web Services, XXVI
  APIs comumente utilizadas, XXVII
  Backpack, XXVII
  Backpack API, XXVII
  Blogger ATOM, feed, XXVII
  Blogging, XXVII
  comércio eletrônico, XXVII
  compartilhamento de foto, XXVII
  compartilhamento de vídeo, XXVII
  Craigslist, XXVI
  de aplicativos, XXVI
  del.icio.us, XXVI
  Dropcash, XXVII
  eBay, XXVII
  Flickr, XXVI, XXVII
  Flickr APIs, XXVI
  funcionalidade de negócios, XXVII
  Gerencia programas de publicidade do Google AdWords, XXVII
  gestão de relacionamento com o cliente (*customer relationship management* — CRM), XXVII
  Givezilla, XXVIII
  Google AdWords, XXVII
  Google Maps, XXVI, XXVII
  IBM, Enterprise Mashup Tool Enterprise da, XXVIII
  Leilões, XXVII
  mapeando APIs, XXVI
  mashups populares, XXVI
  Mashups Resource Center, XXVII
  Microsoft, XXVI
  Microsoft Virtual Earth, XXVI
  Organizador de angariação de fundos, XXVII
  Pagamentos, XXVII
  PayPal, XXVII
  Pesquisa local, XXVII
  Podbop, XXVIII
  ProgrammableWeb, XXVII
  questões de desempenho e confiabilidade, XXVIII
  reutilização de software, XXVII
  RSS feed, XXVI
  RSS Resource Center, XXVIII
  SalesForce, XXVII
  Salesforce.com, XXVII
  serviços Web, XXVI
  Smashforce, XXVIII
  social bookmarking, XXVII
  Strmz, XXVIII
  Technorati APIs, XXVIII
  TypePad ATOM, XXVII
  Upcoming.org, XXVII
  Web 2.0, XXVI
  Yahoo! Maps, XXVIII
  Yahoo! Search, XXVIII
  YouTube, XXVII
mashups populares, XXVI
Mashups Resource Center, XXVII
Matcher, classe, 516, **541**
  find, método, **541**
  group, método, **542**
  lookingAt, método, **541**
  matches, método, **541**
  replaceAll, método, **541**
  replaceFirst, método, **541**
matcher, método da classe Pattern, **541**
matches, método da classe Matcher, **541**
matches, método da classe Pattern, **541**
matches, método da classe String, **536**
Math, classe, **126**, 156, 157
  abs, método, 157
  ceil, método, 157
  cos, método, 157
  E, constante, **157**
  exp, método, 157
  floor, método, 157
  log, método, 157
  max, método, 157
  min, método, 157
  PI, constante, **157**, 182
  pow, método, 126, 156, 157, 182
  random, método, 164
  sqrt, método, 157, 162
  tan, método, 157
Math.E, **157**
Math.PI, **157**
Math.PI, constante, 53, 512, 741
Matisse GUI, designer (Netbeans), **XII**
max, algoritmo, 650
max, método da classe Collections, 645, **650**
max, método de Math, 157
maximizar uma janela, 425, 787
máximo divisor comum (MDC), 187, 611
  exercício, 611
MBCS (multi-byte character set), **XXXI**
MDI (Multiple Document Interface), 769, **784**
mecanismo de extensão, **267**
  estendendo Java com bibliotecas de classes adicionais, **267**
  java.sun.com/javase/6/docs/technotes/guides/extensions/, 267
mecanismo de extensão de tags, **951**
Mecanismo de extensão do Java (java.sun.com/javase/6/docs/technotes/guides/extensions/), 267
média, 42, 89, 91
média aritmética, 42
média áurea, **594**
Mediator, padrão de design, LXII
meia palavra, 239
membro não-static, 259
membros de acesso a pacotes de uma classe, 268
memento, objeto, **LXII**
Memento, padrão de design, LIX, LXII
memória, **3**
memória, buffer, 577
memória, consumo, 598, 637
memória, posição, 40
memória principal, **3**
memória reivindicada, 261
memória, utilização, 659
memória, vazamento, 258, 345
MemoryImageSource, classe, 765
menor de vários inteiros, 151
menor inteiro em um grupo, 741
mensagem, 15, 67, 724, 728
mensagem aninhada na UML, **385**
mensagem de retorno na UML, 385
mensagem na UML, **382**, 383, 384, 385
menu, **419**, 468, 773
menu pop-up sensível ao contexto, **779**
MessageContext, interface, **1037**
  get, método, 1037
Message, componente JSF, **964**
Message Group, componente JSF, **999**
metadados, **914**
Meta, chave, 455
metal, aparência, 769, **781**
Meta, componente JSF, 960
método, **7**, 16, **31**, 391
  assinatura, **175**
  declaração, 31
  lista de parâmetros, **61**
  parâmetro, **61**, 62
  static, 126
  tipo de retorno, 65
  variável local, **63**
método abstrato, **309**, 311, 313, 398, 434, 1071
método auxiliar, **133**, 713
método chamador, **59**, 65, 156
método de acesso, **252**
método de classe, **156**
método de comparação natural, **645**
método de consulta, **252**
método exponencial, 157
método genérico, **671**, 673, 677
método modificador, **252**
método predicado, **253**
Método que utiliza grade drawLine, exercício, 512
Método que utiliza grade drawRect, exercício, 512
métodos chamados automaticamente durante execução do applet, 730
métodos de ciclo de vida, 954, 956, 958
  destroy, **956**, 958
  init, **956**, 958
  preprocess, **956**, 958
  prerender, **956**, 958
métodos de tratamento de evento de janela, 452
métodos de visualização de intervalo, **643**, 657
métodos empacotadores da classe Collections, **639**
métodos implicitamente final, 321
método sobrecarregado, 672
método utilitário, **133**
métrica de fonte, **493**
  altura, 495
  ascendente, 495
  descendente, 495
  entrelinha, 495
Microsoft, XXVI, XXIX
Microsoft Audio/Video Interleave (.avi), formato de arquivo, **758**
Microsoft Internet Explorer, 74
Microsoft SQL Server, 899
Microsoft Virtual Earth, XXVI
Microsoft Windows, 131, 484, 772, 781
Microsoft Windows, aparência e funcionamento, 781

.mid, extensão de arquivo, **756**, **758**
MIDI (Musical Instrument Digital Interface), formato de arquivo (extensões .mid ou .rmi), 756, 758
MIME (Multipurpose Internet Mail Extensions), **949**, 971
min, algoritmo, 650
min, método da classe Collections, 645, **650**
min, método de Math, 157
minimizar frame interno, 786
minimizar uma janela, 425, 773, 787
m por *n*, array, **209**
mnemônico, **425**, **773**, 776
modelo de evento de delegação, **434**
modelo de retomada de tratamento de exceções, **341**
modelo de retomada do tratamento de exceções, **341**
modelo de segurança de caixa de areia, **733**
modelo de um sistema de software, 368, 374, 396
modelo em cascata, **364**
modelo iterativo, **364**
modelo (na arquitetura MVC), LXIX
Model-View-Controller (MVC), LXIX, LXX
Modificação da forma de registro, exercício, 993
Modificação de serviço Web de reserva de companhia aérea, 1065
Modificação do sistema de contas a pagar, exercício, 334
Modificação do sistema de folha de pagamento, exercício, 334
modificador de acesso, **59**, 64, 391
  private, **64**, 245, 281
  protected, 245, **281**
  public, **59**, 245, 281
modificador de acesso na UML
  + (publicã), 60
  - (private), 67
modificando a representação interna dos dados de uma classe, exercício, 275
modo de interrupção, **1080**
modos de seleção, **447**
modularizando um programa com métodos, 156
módulo, 41, **155**
módulos em Java, 155
moeda, lançamento, 165, 187
MoleculeViewer, applet, 724
monitor, **812**
monitorar eventos do mouse, 450
monitor de exibição, 484
Monospaced, fonte do Java, 492
Morse Code Web Service, 1065
Morse, código, 897
Motif-style (UNIX), aparência e funcionamento, **769**, 781
mouse, 3, 419, 724
MouseAdapter, classe, 452
  mousePressed, método, 857
mouseClicked, método da interface MouseListener, 449, 452
mouse de múltiplos botões, 454
mouse de três botões, 454
mouse de um, dois ou três botões, 454
mouseDragged, método da interface MouseMotionListener, 449, 455
mouseEntered, método da interface MouseListener, 449
MouseEvent, classe, **435**, 449, 781
  getClickCount, método, **455**
  getPoint, método, **456**
  getX, método, **452**
  getY, método, **452**
  isAltDown, método, **455**
  isMetaDown, método, **455**
  isPopupTrigger, método, **781**
mouseExited, método da interface MouseListener, 449
MouseInputListener, interface, **449**, 452
MouseListener, interface, **435**, 449, 452, 781
  mouseClicked, método, 449, 452
  mouseEntered, método, 449
  mouseExited, método, 449
  mousePressed, método, 449, 781
  mouseReleased, método, 449, 781
MouseMotionAdapter, classe, 453, 456
MouseMotionListener, interface, **435**, 449, 452
  mouseDragged, método, 449, 455
  mouseMoved, método, 449, 455
mouseMoved, método da interface MouseMotionListener, 449, 455
mousePressed, método da classe MouseAdapter, 857
mousePressed, método da interface MouseListener, 449, 781
mouseReleased, método da interface MouseListener, 449, 781
MouseWheelEvent, classe, **450**
MouseWheelListener, interface, **450**
  mouseWheelMoved, método, **450**
mouseWheelMoved, método da interface MouseWheelListener, **450**
.mov, extensão de arquivo, **758**
moveTo, método da classe GeneralPath, **508**
Mozilla Foundation, 18
.mp3, extensão de arquivo, **758**
MPEG-1, vídeos (extensões .mpeg ou .mpg), **758**
.mpeg, extensão de arquivo, **758**
MPEG Layer 3 Audio (.mp3), formato de arquivo, **758**
.mpg, extensão de arquivo, **758**
mudando de diretório, 33, 724
muitos para um, mapeamento, **658**
multicast, **860**, 893
multimídia, **743**
múltiplas declarações de classe em um arquivo de código-fonte, 248
MULTIPLE_INTERVAL_SELECTION, constante de interface ListSelectionModel, 447
multiplicação, *, operador, **41**
multiplicação, operador de atribuição composta, *=, 102
multiplicidade, **368**
multiply, método da classe BigInteger, **594**
multiprocessador, **4**
multiprogramação, **4**
Multipurpose Internet Mail Extensions (MIME), **949**, 971
multitarefa, **8**
multithreading, 8, 639, **804**, LXVII
mundo virtual, 18
Musical Instrument Digital Interface (MIDI), formato de arquivo (extensões .mid ou .rmi), **756**
Musical Instrument Digital Interface (MIDI), formato de arquivo (.mid ou .rmi extensões), 758
MVC (Model-View-Controller), LXIX
MyShape, hierarquia, 330
MyShape, hierarquia com MyBoundedShape, 331
MySpace, 18
MySQL, **19**, 899
MySQL 5.0 Community Edition, **910**
MySQL Connector/J, **910**
mysqld-nt.exe, 910
MySQL (www.mysql.com), 910
MySQL (www.mysql.com/products/mysql/), 910

N

n, caractere de conversão, **1096**
name, atributo do applet-desc elemento, 736
não-ambíguo, princípio do projeto Unicode, **XXIX**
NASA Multimedia Gallery, 761
NASA, multimídia (www.nasa.gov/multimedia/highlights/index.html), 761
native, palavra-chave, 1069
navegabilidade bidirecional na UML, **391**
navegador, 74
navegador Web, 74, 723, 863
  executar um applet, 727, 730
navegando, 862
Navigator, janela no NetBeans, **960**
negação lógica, !, **138**
NervousText, applet, 724
NetBeans IDE, **947**, 1019, XII
  adicionar uma referência de serviço Web a um aplicativo, 1025
  adicionar um handler de evento, XVI
  criar um aplicativo desktop, 1025
  criar um aplicativo Web, 1020
  criar um banco de dados Java DB, 996
  criar um novo projeto, XIII
  Design, visualização, XIII
  editor visual, **956**
  grade de alinhamento, XII
  GroupLayout, **XII**
  initComponents, método autogerado, XVI
  linhas de guia, **XIV**, XV
  New JFrame Form, diálogo, 1026
  New Web Service Client diálogo, 1025
  New Web Service diálogo, 1021
  Palette, janela, XIII, XIV
  Properties, janela, XIII, XV
  Services, guia, 997
  Show Line Numbers, 958
  Source, visualização, XIII
  vídeo de demonstração (www.deitel.com/books/jhtp8), 29
  vinculando um JSF Table a uma tabela de banco de dados, 997
  Web Application, projeto, **1020**
Netbeans IDE, ferramenta de design, xxiv
Netbeans Matisse GUI, designer, **XII**
NetBeans (www.netbeans.org), 9
new, palavra-chave, **38**, **60**, 191, 1069
newCachedThreadPool, método da classe Executors, **810**
newCondition, método da interface Lock, **836**, 837
New JFrame Form diálogo, 1026
New Web Service Client diálogo, 1025
New Web Service diálogo, 1021
new Scanner(System.in), expressão, 38
New to Java Center (java.sun.com/new2java/), 8
next, método de ResultSet, **914**
next, método da classe Scanner, **62**
next, método de interface Iterator, **641**
nextDouble, método da classe Scanner, **72**
nextInt, método da classe Random, **164**, 167
nextLine, método da classe Scanner, **62**
Nimbus, aparência e comportamento, **420**, 781
  swing.properties, xxxii, 420
níveis de aninhamento, LIV
nível de recuo, 86
nó, LXIV
nó em uma lista, **696**

nó filho, **709**
nó folha, **709**
  em uma árvore de pesquisa binária, 713
NomeDaClasse .this, 778
nome de classe, **30**
  completamente qualificado, 63
nome de classe completamente qualificado, **63**, **265**
nome de domínio Internet em ordem inversa, 265
nome de fonte, **492**
nome de host, 876
nome de papel na UML, **369**
nome de um array, **191**
nome de uma variável, **40**
nome de um parâmetro, 862
nome qualificado, **907**
nomes de diretório na declaração package, 265
nome simples de uma classe, **265**
NONE, constante da classe GridBagConstraints, **793**
nó pai, 709, 719
NoPlayerException, exceção, **759**
nó raiz, **709**
NORTH, constante da classe BorderLayout, **452**, 463
NORTH, constante da classe GridBagConstraints, **793**
NORTHEAST, constante da classe GridBagConstraints, **793**
NORTHWEST, constante da classe GridBagConstraints, **793**
nós irmãos, **709**
NoSuchElementException, classe, **562**, 565
notação algébrica, 41
Notação Big O, **618**, 622, 625, 628, 632
notação científica, 1092
notação científica computadorizada, **1092**
notação de infixo, 717
notação exponencial, **1092**
notação pós-fixa, 717
nota (na UML), **84**
notifyAll, método da classe Object, 299, **825**, 827
notify, método da classe Object, 299
notify, método da classe Object, **825**
nova linha, caractere (\l), **34**
nova linha, sequência de escape, , 239
nova linha, sequência de escape, \n, 35, 38, 516
null, 1069
null, palavra-chave, 65, **67**, 74, 192, 422, 696
null, palavra reservada, 105
NullPointerException, classe, 336
Number, classe, 687
  doubleValue, método, **688**
número complexo, 276
número de coluna em um conjunto de resultados, 904
número de identificação de empregado, 553
número de ponto flutuante de precisão dupla, **70**
número de ponto flutuante de precisão simples, **70**
número de porta, **866**, 867, 876, 877, 882
número de porta do servidor, 876
número não especificado de argumentos, 217
número perfeito (exercício), 187
número primo, 234
número pseudoaleatório, **164**, 167
número real, 38
números aleatórios, 167
  deslocar um intervalo, **165**
  diferença entre valores, 167
  elemento de chance, **164**
  escalonamento, **165**
  fator de escalonamento, **165**, 167
  geração, 199
  geração para criar orações, 546
  número pseudoaleatório, **164**
  processando, 163
  semear, **164**
  valor de deslocamento, **165**, 167
  valor de semente, **167**
Números complexos, exercício, 276
Números racionais, exercício, 276

O

O(1), tempo, **618**
O(log *n*), tempo, **622**
O(*n*), tempo, **618**
O(*n*2ÿ), tempo, **618**
O(*n* log *n*), tempo, **632**
Object, classe, 258, 279, **282**, 576
  clone, método, 298
  equals, método, 298
  finalize, método, 298
  getClass, método, 298, **320**, **427**
  hashCode, método, 299
  notifyAll, método, 299, **825**, 827
  notify, método, 299, **825**
  toString, método, 284, 299
  wait, método, 299, **825**
ObjectInput, interface, **570**
  readObject, método, **570**
ObjectInputStream, classe, **555**, 570, 574, 866, 867, 871
Object Management Group (OMG), **17**
  www.omg.org, 17
Object Management Group (www.omg.org), **17**
ObjectOutput, interface, **570**
  writeObject, método, **570**
ObjectOutputStream, classe, **555**, 570, 663, LXVIII
  close, método, **574**
  flush, método, **871**
objeto, 2
objeto-assunto, LXV
objeto de entrada padrão (System.in), **38**
objeto de evento, 434
objeto de saída padrão (System.out), **32**
objeto desserializado, **570**
objeto de uma classe derivada, 307
objeto de uma classe derivada é instanciado, 296
objeto empacotador (coleções), **663**
objeto imutável, **260**
objeto, orientação, 15
objeto originador, **LXII**
objeto (ou instância), 15, 384, 728
objeto proxy, **LXI**
objeto serializado, **570**
objeto zelador, **LXII**
Observable, classe, LXV
Observações de engenharia de software, **7**
Observações de engenharia de software, visão geral, xxv
Observações sobre a aparência e comportamento, **7**
Observer, interface, LXV
observer, objeto, LXV
Observer, padrão de design, LIX, LXII, LXV
obter (*get*) um valor, **66**
ocultamento de dados, **64**
ocultamento de informações, **15**, 64
ocultar detalhes da implementação, 156, 245
offer, método da classe PriorityQueue, **655**
offline-allowed, elemento de um documento JNLP, **735**
Oito rainhas, exercício, 234, 612
  abordagens de força bruta, 234
OK, botão, 75
OMG (Object Management Group), 17
ON, cláusula, **907**
ONE, constante da classe BigInteger, **594**, 595
OOAD (*object-oriented analysis and design*), 16
OOD (*object-oriented design*), xxiv, **15**, 361, 365, 366, 371, 374
OOP (*object-oriented programming*), **16**, 242, 279
opções de compilador
  -d, **265**
opções mutuamente exclusivas, **440**
OPEN, constante da classe Arc2D, **506**
open, método da classe Desktop, **XXII**
openFileDialog, método da interface FileOpenService, **746**
openMultiFileDialog, método da interface FileOpenService, **751**
openStream, método da classe URL, **1034**
operação, **60**
  na UML, 60
operação atômica, **816**, **939**
Operação de um objeto, **15**
operação física de entrada, **577**
operação física de saída, **577**
operação na UML, 368, 377, 380, 393, 394, 397
operação para desenfileiramento de fila, **707**
operação para enfileiramento de fila, **707**
operações concorrentes, **804**
operações de carregar/armazenar, 236
operações de deslocamento de bits, L
operações lógicas de entrada, **577**
operações lógicas de saída, **577**
operações paralelas, **804**
operador, **39**
operador condicional, ?:, 86, 104
operador de comparação, 329
operador de decremento, --, **102**
operador de igualdade == para comparar objetos String, 519
operador de resto, %, **41**
Operadores
  ^, OU lógico booleano exclusivo, 136, **138**
  --, pré-decremento/pós-decremento, **103**
  -=, operador de atribuição de subtração, 102
  !, NÃO lógico, 136, **138**
  ?:, operador condicional ternário, 86, 104
  *=, operador de atribuição de multiplicação, 102
  /=, operador de atribuição de divisão, 102
  &&, E condicional, 136
  &, E lógico booleano, 136, **137**
  %=, operador de atribuição de resto, 102
  +=, operador de atribuição de adição, **102**, 131
  ++, pré-incremento/pós-incremento, 103
  ++, pré-incremento/pré-incremento, 103
  =, atribuição, **39**, 45
  ||, OU condicional, 136, **137**
  |, OU inclusivo lógico booleano, 136, **137**
  atribuição, =, 45
  binário, **39**, 41, 138
  coerção, **97**
  complemento lógico, !, **138**
  de atribuição composta, 102, 104
  E condicional, &&, **136**, 137
  E lógico booleano, &, 136, **137**
  exponenciação, 126
  incremento, ++, 102
  incremento e decremento, 102
  lógicos, 138
  módulo, %, 41
  multiplicação, *, **41**
  multiplicativos, *, / e %, **98**
  negação lógica, !, **138**
  operador condicional, ?:, **86**, 104
  operador de decremento, --, 102

operadores lógicos, **136**, 138, 140
OU condicional, ||, 136, 137
OU inclusivo lógico booleano, |, **137**
OU lógico booleano exclusivo, ^, 136, **138**
pós-decremento, **102**
pré-decremento, **102**
pré-incremento, **102**
resto, %, 41
ternário, **86**
unário, 138
operadores de atribuição de bits, **L**
^=, (OU exclusivo sobre bits), LI
&=, (AND sobre bits), LI
<<=, (deslocamento de bits para a esquerda), LI
>>=, (deslocamento para a direita com sinal), LI
>>>=, (deslocamento para a direita sem sinal), LI
|=, (OU inclusivo sobre bits), LI
operador unário, **97**
coerção, **97**
operando, **39**, 98, 237
or, método da classe BitSet, **LI**
Oracle Corporation, 899, XXIX
ordem, 82
ordem crescente, 219
ASC, em SQL, 905, 906
Ordem de bloclos catch, exercício, 359
ordem de classificação, 657, 658
ordem decrescente, 219
ordem de handlers de exceção, 359
ordem em que as ações devem ser executadas, **82**
ordenação de registros, 903
ORDER BY, cláusula de SQL, 903, **905**, 906
Organizador de angariação de fundos, XXVII
orientação horizontal sequencial em GroupLayout, **XII**
orientado para a ação, **16**
otimização do compilador, 127
OU condicional, ||, operador, 136, **137**
tabela-verdade, 137
OU exclusivo sobre bits (^) operador, **XLIII**, XLIV
OU inclusivo sobre bits (|) operador, **XLIII**
OU lógico booleano exclusivo, ^, operador, 136, **138**
tabela-verdade, 138
OU lógico booleano inclusivo, |, operador, **137**
OutOfMemoryError, 696
OutputStream, classe, 570, **576**, 866, 867
OutputStreamWriter, classe, **578**
ouvinte de evento, 329, 432, 452
classe adaptadora, 452
interface, 431, 432, 433, 435, 449, 452
ouvinte registrado, 434
ouvir eventos, **431**, 432
oval, 495, 498, 728
oval preenchida com cores que mudam gradualmente, 505
oval unida por um retângulo, 498
@Override, anotação, **284**

P

package, declaração, **264**
package-list, gerado por javadoc, XLII
package, palavra-chave, 1069
pack, método da classe Window, **787**
pacote, **37**, 163, 264, **859**, 877, XXXIX
pacote básico, 33
pacote de datagrama, **859**, 877
pacote de rede, 163
pacote, estrutura de diretórios, 264
pacote, nome, 63
pacote opcional, **267**
pacote padrão, **63**, 264
Pacotes, 155
com.google.gson.Gson, 1033
com.sun.rave.web.ui.appbase, **954**
com.sun.webui.jsf.component, **954**
com.sun.webui.jsf.model, 1012
java.applet, 163
java.awt, 163, **423**, 486, 503, 744, 752, 772, 781
java.awt.color, 503
java.awt.event, 163, 433, 452, 460
java.awt.font, 503
java.awt.geom, 163, 503
java.awt.image, 503
java.awt.image.renderable, 503
java.awt.print, 503
java.beans, 850
java.io, 163, **555**
java.lang, **38**, 157, 163, 282, 298, 516, 808
java.math, 70, 593
java.net, 163, **859**
java.sql, 163, 913
java.swing, 728
java.text, 163
java.util, **37**, 163, 221, 687
java.util.concurrent, 164, 810, 823, 851
java.util.concurrent.locks, 835, 836
java.util.prefs, **661**
java.util.regex, 516
javax.faces.validators, **964**
javax.jnlp, 734, 743, 746
javax.media, 164, **758**
javax.servlet, **950**
javax.servlet.http, **950**, 976, 978
javax.servlet.jsp, 951
javax.servlet.jsp.tagext, 951
javax.sql.rowset, **924**
javax.swing, 164, 421, **423**, 427, 435, 470, 486, 744, 772, 784, 786
javax.swing.event, 164, **433**, 447, 452, 772
javax.swing.table, 915, **923**
pacote padrão, **63**
pacotes da Java API, **163**
pacote, visão geral (java.sun.com/javase/6/docs/api/overview-summary.html), 164
padrão, 506
padrão arquitetônico Layers, LXX
padrão de 1 e 0 s, 553
padrão de design Visitor, LXII
padrão de preenchimento, 506
padrão, princípio do projeto Unicode, XXIX
padrões arquitetônicos, LIX, LXIX, LXX
padrões de design, xxii, **18**
padrões de design comportamentais, LIX, LXI, LXIV
padrões de design criacionais, **LIX**, LXII, LXVII, LXX
padrões de design de concorrência, **LIX**, LXVII
padrões de design estruturais, LIX, **LXI**, LXIII, LXVIII
padrões de design Java, xxvii
padrões de design, elementos de software orientado a design reutilizáveis, LVIII
Pagamentos, XXVII
Page Down, tecla, 458
Page Up, tecla, 458
página inicial, 961
paginationControls, propriedade de um Table, **998**
página Web, 74
painel, 467
Painel de caracteres rolantes, exercício, 766
painel de conteúdo, 447, 778
setBackground, método, **447**
Painel de imagens rolantes, exercício, 766
painel de vidro, 447
paint, método da classe JApplet, **728**
paint, método de JApplet, 730, 732
Paint, objeto, **505**
paintComponent, método da classe JComponent, **107**, **455**, 484, 751, 770, 772
paintIcon, método da classe ImageIcon, **751**
palavra-chave, **30**, 84
palavra reservada, **30**, 83, 1069
false, 86
null, 65, **67**, 105
true, 85
Palavras-chave
abstract, **309**
boolean, **86**, 1082
break, **132**
case, **132**
catch, **340**
char, **38**
class, **30**, 59
continue, **134**
default, **132**
do, 84, **127**
double, **38**, **70**
else, 84
enum, **171**
extends, **107**, **282**, 289
false, **86**, 1069
final, 134, **158**, 195, 817
finally, **341**
float, **38**, **70**
for, 84
if, 84
implements, **322**
import, **37**
instanceof, **319**
int, 38
interface, **322**
new, 38, **60**, 191
null, **67**, 192, 1069
private, **64**, 245, 252
public, 31, 58, 59, 64, 245
reservadas, mas não utilizadas pelo Java, 1069
return, **65**, 156, 161
static, **74**, 126, 156
super, **281**, 297
switch, 84
synchronized, **812**
tabela de palavras-chave e palavras reservadas, 1069
this, **246**, 259
throw, **348**
true, **86**, 1069
try, **340**
void, **31**, 59
while, 84, **127**
Palavras-chave do Java, 1069
palavras cruzadas, exercício, 766
palavras e frases descritivas, 371, 372
Palette, 959, 961
Palette no NetBeans, **957**
palíndromo, 117, 611
Palíndromos, exercício, 611
papel na UML, 369
param, elemento, **862**
Parameters:, nota
nota, **XXXVIII**
parâmetro, **61**, 62
formal, **159**
parâmetro de applet, **862**
parâmetro de exceção, **341**
parâmetro de operação na UML, 63, 377, 380, 381
parâmetro de saída para CallableStatement, **938**

parâmetro formal, **159**
parâmetro formal de tipo, **674**
parâmetro na UML, 62, 377, 380, 381
@param, tag javadoc, **XXXVIII**
parênteses, **31**, 41
  aninhados, **41**
  redundantes, **43**
parênteses aninhados, **41**
parênteses desnecessários, 43
parseDouble, método de Double, 732
parseInt, método da classe Integer, 75, 145, 219, 422
parte imaginária, 276
parte real, 276
partes intercambiáveis padronizadas, 16
parte superior, 655
parte superior de uma pilha, **695**
participante de um formulário virtual, **1007**
Pascal, Blaise, 8
passagem de mensagem na UML, 385
passando arrays para métodos, 203
passando elementos de array para métodos, 203
passando opções para um programa, 218
passar por referência, **205**
passar por valor, 203, **205**
passeio completo, 513
Passeio do cavalo, 232, 513
  abordagem de força bruta, 233
  exercício, 513
  teste do passeio fechado, 234
passeio fechado, 234, 513
passo de partição em quicksort, 635
pasta de arquivo, 726
@Path, anotação, **1029**
PATH, variável de ambiente, xxxi, 33
@PathParam, anotação, **1029**
pathSeparator, campo static de File, 558
Pattern, classe, 516, **541**
  compile, método, **541**
  matcher, método, **541**
  matches, método, **541**
Payable, declaração de interface, 323
Payable, diagrama de classe UML de hierarquia de interface, 323
Payable, processamento de programa de teste de interface Invoice e Employee polimorficamente, 328
PayPal, XXVII
pedagogia baseada na introdução antecipada de classes e objetos, xxi
peek, método da classe PriorityQueue, **655**
peek, método da classe Stack, **655**
pequenos círculos (na UML), **83**
pequeno símbolo de losango (para representar uma decisão em um diagrama de atividades da UML), 375
percorrer uma árvore, 713
percorrer um array, **211**
percurso em uma árvore binária em ordem de nível, **714**, 720
percurso na ordem, **709**
percurso na pós-ordem, **710**, 712
percurso na pré-ordem, **710**
Percurso para sair de um labirinto utilizando reversão recursiva, exercício, 612
persistente, **4**
persistente Hashtable, 661
personalização, **970**
pesquisa, 199
pesquisa binária de um array, **199**
pesquisa de dados, **615**
pesquisa de DNS, **948**
Pesquisa local, XXVII
pesquisando, 695
Pesquisa sequencialmente um item em um array, 617
PHP, **19**
PI, 512
PIE, constante da classe Arc2D, **506**
pilha, **161**, 678, **695**, 703
  estouro de pilha, **162**
  pilha de chamadas de método, **162**
  programa, pilha de execução, **162**
pilha de chamadas de método, **162**
pilha de execução do programa, **162**
pilha, operação
  colocar na pilha (*push*), **704**
  remover da pilha (*pop*), **704**
pior cenário de tempo de execução para um algoritmo, 618
pipe, **576**
PipedInputStream, classe, **576**
PipedOutputStream, classe, **576**
PipedReader, classe, **578**
PipedWriter, classe, **578**
PixelGrabber, classe, 765
pixel (*picture element*), **105**, **484**
placa de som, 756
PLAF (*pluggable look-and-feel*), 769
PLAIN, constante da classe Font, **491**, 492
PLAIN_MESSAGE, 422
plataforma .NET, **8**
play, método da classe Applet, **756**
play, método da interface AudioClip, **756**
Player, interface, **758**
  getControlPanelComponent, método, **760**
  getVisualComponent, método, **760**
  start, método, **760**
.png, extensão de nome de arquivo, 427
.png, extensão de arquivo, **744**
PNG (Portable Network Graphics), 427, 744
Podbop, XXVIII
Point, classe, **456**
POJO (Plain Old Java Object), **1021**
polígono, **500**, 503
polígonos fechados, 501
polilinha, **500**
polilinhas, 500
polimorfismo, 134, 301, **305**, 395, 396, 402
polinômio, 42, 43
polinômio de segundo grau, 42, 43
poll, método da classe PriorityQueue, **655**
Polygon, classe, **484**, 500
  addPoint, método, 501, **503**
ponta de seta em um diagrama de sequência na UML, 385
pontilhamento, 724
ponto, **492**, 726
ponto ativo (mapa de imagens), 724
ponto de entrada, 140
ponto de inserção, 221, 651, 697
ponto de interrupção, **1078**
  inserindo, 1080, 1081
  listando, 1088
  removendo, 1088
ponto de lançamento, **338**
ponto de saída, 140
  de uma instrução de controle, 84
ponto-e-vírgula (;), **32**, 38, 46
ponto flutuante, número, **70**, 93, 95, 97, 655, 732, 1093
  divisão, 98
  double, tipo primitivo, **70**
  float, tipo primitivo, **70**
  precisão dupla, **70**
  precisão simples, **70**
pontos ativos no bytecode, **10**
ponto separador (.), **60**, 74, 126, 157, 258, 503
pool de threads, **810**, 867
pop, método da classe Stack, **655**
pôquer, 235
Pôquer, jogo, 896
porcentagem (%), caractere curinga de SQL, 904
porta, **866**
portabilidade, 11, 492, XXXI
Portable Network Graphics (PNG), **427**, **744**
porta de conexão, 866
portável, **10**
pós-condição, **354**
pós-decrementar, **103**
pós-decremento, operador, **102**
posição do indicador de classe JSlider, 772
posição, número, 190
posicionamento absoluto, **959**
pós-incrementar, **103**, 104
pós-incremento, operador, **102**
postback, **956**
PostgreSQL, 899
potência de 3 maiores que 100, 89
potência (expoente), 157, 186
pow, método da classe Math, 127, 156, 157, 182
precedência, **41**, 46, 104, 595
  gráfico, 41
  operadores aritméticos, 41
  tabela, 98
precedência de operadores, 41, 595
  regras, **41**
  tabela de precedência de operadores, 98
precedência de operadores, tabela, 1066
precisão, **71**, 1091, 1093
  formato de um número de ponto flutuante, 98
precisão de um número de ponto flutuante formatado, **71**
precisão de um valor de ponto flutuante, 70
pré-condição, **354**
pré-decrementar, **103**
pré-decremento, operador, **102**
predicado, **904**
predicado, método, 701
preencher com cor, 484
pré-incrementar, **103**, 104
pré-incrementar e pós-incrementar, 103
PreparedStatement, interface, **927**, 929, 938
  documentação (java.sun.com/javase/6/docs/api/java/sql/PreparedStatement.html), 927
  executeQuery, método, **932**
  executeUpdate, método, **932**
  setString, método, **927**, 932
prepareStatement, método da interface Connection, **929**
preprocess, método (JSF), **956**, 958
prerender, método (JSF), **956**, 958
previous, método de interface ListIterator, **643**
primary, propriedade de um JSF Button, **996**
primeiro refinamento, 99
primeiro refinamento no refinamento passo a passo de cima para baixo, 93
primos, 187, 669
princípio do menor privilégio, **262**
print, comando de depurador, **1081**
printArray, método genérico, 674
printf, método de System.out, 1091
printStackTrace, método da classe Throwable, **349**
PrintStream, classe, **577**, 663
PrintWriter, classe, **562**, 578
prioridade de thread, **807**

PriorityQueue, classe, **655**
  clear, método, **656**
  offer, método, **655**
  peek, método, **655**
  poll, método, **655**
  size, método, **656**
private
  campo, 252
  dados, 252
  modificador de acesso, **64**, 245, 281
  palavra-chave, 252, 391, 1069
private, palavra-chave, **64**
private static
  membro de classe, 258
probabilidade, 164
probabilidade igual, 166
problema da média de classe, 89, 90, 95
problema da média de classe geral, 93
problema de concorrência, LXVII
problema dos resultados do exame, 100
procedimento, **156**
procedimento para resolver um problema, 82
procedure armazenada, **938**
processador de dois núcleos, 4
processador de múltiplos núcleos, **4**
processador de quatro núcleos, 4
processador de texto, 516, 522
processamento de arquivo, 555
processamento de dados comercial, 586
processamento de transação, **939**
processamento em lote, **4**
processamento polimórfico
  de coleções, 639
processamento polimórfico de exceções relacionadas, 344
processando polimorficamente Invoice e Employee s, 328
processo baseado em evento, 509
processo de design, **16**
processo de implementação, 378, 391
processo de refinamento, 93
processo do projeto, 361, 364, 378, 381
@Produces, anotação, **1029**
produto de inteiro ímpar, 151
produtor, **818**
produtor/consumidor, relacionamento, **818**, 830
Professor de digitação
  Aprimorando uma habilidade crucial na era dos computadores, 481
programa, 3
programação concorrente, **805**
programação de jogos, xxii, xxvii
Programação de jogos, **18**
programação estruturada, 2, 3, **8**, **83**, 120, 136, 140
  resumo, 140
programação orientada a objetos (*object-oriented programming* — OOP), 2, 3, 6, 16, 242, 279
programa de computador, **3**
Programa de conversão métrica, 548
programa de gerenciamento de vídeo, 306
programa de jogo de xadrez, 896
Programa de Planta Baixa, exercício, 766
programador, **3**
programador de computador, **3**
programa, pilha de execução, 704
programa, princípios de construção, 147
programar no específico, 305
programar no geral, 305, 334
programas gráficos gratuitos (www.freebyte.com/graphicprograms), 762
programa tolerante a falha, **336**
programa tradutor, **5**
ProgrammableWeb, XXVII
Projects, janela, 957
Projeto de acessibilidade
  Reconhecimento da fala, 767
  Speech Synthesis, 767
projeto orientado a objetos (*object-oriented design* — OOD), 361, 365, 366, 371, 374, 391
projeto orientado a objetos utilizando a UML, xxiv
projetos de programação, xxii, xxvii
promoção, **98**
  regras, 98, **162**
promoção de argumentos, **162**
promoções para tipos primitivos, 162
prompt, **38**
Prompt de Comando, 32
Properties, classe, **661**, 983
  getProperty, método, **661**
  keySet, método, 663
  list, método, **663**
  load, método, **663**
  setProperty, método, **661**
  store, método, **663**
Properties, janela, 959
propertyChange, método da interface PropertyChangeListener, 850
PropertyChangeListener, interface, **850**
  propertyChange, método, 850
propriedade auto-similar de um fractal, **600**
propriedade de uma classe JavaBean, 951
proteção de cheque, exercício, 547
proteção de privacidade, 970
protected, modificador de acesso, 245, **281**, 1069
Protetor de tela com formas, exercício, 512
Protetor de tela, exercício, 512
Protetor de tela para um número aleatório de linhas, exercício, 512
Protetor de tela utilizando a API Java2D, exercício, 512
Protetor de tela utilizando Timer, exercício, 512
protocolo de comunicação (jdbc), 913
protocolo de estado (HTTP), **971**
protocolo seguro, 979
prototipagem rápida, xxii
Prototype, padrão de design, LIX, LX, LXX
provedor de dados, 998
Proxy, padrão de design, LIX, LXI
pseudocódigo, **16**, **82**, 85, 90, 99
  algoritmo, 94
  primeiro refinamento, **93**, 99
  segundo refinamento, **94**, 100
public
  interface, **242**
  método, 243, 245
  método encapsulado em um objeto, 245
  modificador de acesso, 245
  serviços de uma classe, **242**
  static, membros de classe, 258
  static, método, 258
public
  abstract, método, 322
  final, static dados, 322
  membro de uma subclasse, 281
  método, 107
  modificador de acesso, 58, 59, 64, 281
  palavra-chave, 391, 393, 394, 1069
public, classe, 31
Publicando um serviço Web com um aplicativo Java SE 6 padrão (today.java.net/pub/a/today /2007/07/03/jax-ws-web-services-without-ee-containers.html), 1022
public, palavra-chave, **31**, **64**
publicar um serviço Web, **1019**, 1020, 1022, **1060**
push, método da classe Stack, **655**
put, método
  de interface BlockingQueue, **823**, 824
  de interface Map, **661**
  de interface RequestContext, **1045**

Q

quadrados eliminados posicionando uma rainha no canto superior esquerdo de um tabuleiro de xadrez, 612
quadro delimitador para um oval, 741
quadro de pilha, **162**
quantificadores utilizados em expressões regulares, **539**
quantificador ganancioso (expressões regulares, **539**
quantificador preguiçoso (expressões regulares), **539**
quantificador relutante (expressões regulares), **539**
quantum, **806**
quebra de linha, **470**
QUESTION_MESSAGE, 422
Questionário sobre o aquecimento global, 153
Queue, interface, 637, **639**, 655, 823
quicksort, algoritmo, 635
QuickTime (.mov), formato de arquivo, **758**
Quilometragem de combustível, 115

R

RadialGradientPaint, classe, 505
radianos, 157
radical (base) de um número, **533**, **534**
Radio Button Group, componente JSF, 961, **964**
raio, 512, 741
raio de um círculo, 187
raiz quadrada, 157
ramificação, LXIV
Random, classe, 163, **164**, 231
  java.sun.com/javase/6/docs/api/java/util/Random.html, 164
  nextInt, método, **164**, 167
  setSeed, método, 168
random, método da classe Math, 164
range, método da classe EnumSet, **257**
rastreamento de pilha, **337**
rastreando clientes, 970
Rational, classe, 276
Rational Software Corporation, 17, 365
Rational Software Corporation™, 365
RDBMS (*relational database management system*), 950
read, método da classe InputStream, **746**
Reader, classe, **578**
readObject, método de ObjectInput, **570**
readObject, método de ObjectInputStream, 576
Read/Write Lock, padrão de design, LIX, LXVII
realização na UML, **323**
realizando operações concorrentemente, 804
realizando uma tarefa, 59
realizar uma ação, 32
realizar um cálculo, 47
recarregar uma página Web inteira, 1005
receber dados de um servidor, 876
receber uma conexão, 871
receive, método da classe DatagramSocket, **879**
recomendações curriculares da ACM/IEEE e do Computer Science Advanced Placement Examination, xxii

reconhecendo clientes, 971
reconhecimento de fala, 767
recorrendo texto em uma JTextArea, 470
Rectangle2D, classe, 484
Rectangle2D.Double, classe, **503**
Rectangle, Classe (exercício), 275
recuo, 86, 87
recuperar-se de um erro, 199
recursão
  algoritmo recursivo de pesquisa binária, 635
  algoritmo recursivo de pesquisa linear, 635
  avaliação recursiva, 592
  avaliação recursiva de 5!, 592
  chamada recursiva, **591**, 595
  gerando recursivamente números de Fibonacci, 595
  imprimir recursivamente uma lista de trás para frente, 719
  método factorial recursivo, 592
  método power recursivo, exercício, 611
  método recursivo, **590**
  overhead, 598
  passo de recursão, **591**, 595
  pesquisar recursivamente uma lista, 719
  retorno recursivo, **607**
Recursão, exercícios
  Encontrar o valor mínimo em um array, 612
  Fractais, 612
  Gerador de labirintos aleatório, 613
  Imprimir um array, 612
  Imprimir um array de trás para frente, 612
  Labirintos de qualquer tamanho, 613
  Máximo divisor comum, 611
  Método power recursivo, exercício, 611
  Oito rainhas, 612
  Palíndromos, 611
  Percorrer um labirinto utilizando retorno recursivo, 612
  pesquisa binária, 635
  pesquisa linear, 635
  Tempo para calcular números de Fibonacci, 613
  Visualizando uma recursão, 611
recursão indireta, **591**
recursão infinita, 296, **593**, 597, 598
recursos deste livro para o professor, xxvii
Recursos para o aluno incluídos neste livro, xxvi
recurso, vazamento, 258
rede, 552
rede de computadores, 4
rede local (*local area network* — LAN), **4**
rede peão, 4
rede social, 18
redimensionamento dinâmico, **190**
redirecionar um fluxo padrão, **555**
redireciona um fluxo, 555
ReentrantLock, classe, **835**, 836
refatoração, xxii
refatorando, **18**, **958**
referência, **67**
referência de serviço Web, 1025
referenciar um objeto, **67**
refinamento passo a passo de cima para baixo, 93, 95, 99
refresh, método da classe CachedRowSetDataProvider, **1004**
regexFilter, método da classe RowFilter, **924**
regionMatches, método da classe String, **519**
registrador do acumulador, 236, 238
registrar um ActionListener, 778
registrar uma porta, **866**
registrar um handler de evento, **431**
Registration Form Modification, 993
registro, **553**, 558
registro, chave, **553**, 586
registro de ativação, **162**, 598
registro de evento, 432
registro de transação, 586
registros de intercalação a partir de tabelas, 907
regra de aninhamento, **142**
Regra de Integridade de Entidade, **902**
Regra de Integridade Referencial, **901**
regra de negócio, **950**
regra geral (heurística), 136
regras de precedência de operador, **41**, 595
regras de promoção para tipos primitivos, **162**
regras para formar programas estruturados, 141
Regular Expressions Resource Center (www.deitel.com/regularexpressions/), 542
reinventando a roda, 7, 37, 219
relação áurea, **594**
relação de números de Fibonacci sucessivos, 594
relacionamento cliente-servidor, **859**
relacionamento de muitos para muitos, 903
relacionamento de muitos para um na UML, **370**
relacionamento de um para muitos na UML, **370**
relacionamento de um para um na UML, **370**
relacionamento entre uma classe interna e sua classe de primeiro nível, 440
relacionamento hierárquico de método trabalhador/método chefe, 156
relacionamento inteiro/parte, 369
relacionamento *tem um*, **253**, **279**, 369
relançando exceções, exercício, 359
relançar uma exceção, **348**, 359
RELATIVE, constante da classe GridBagConstraints, **796**
Reload, item do menu Applet de appletviewer, 725, 726
Relógio digital, exercício, 765
REMAINDER, constante da classe GridBagConstraints, **796**
remove, método da classe ArrayList<T>, 221, **222**
remove, método de interface Iterator, **641**
remover da pilha, operação, 701
remover duplicata String, 657
removeTableModelListener, método da interface TableModel, **915**
removeTrayIcon, método da classe SystemTray, **XXIV**
rendered, propriedade de um componente JSF, **965**
renderizando XHTML em um navegador da Web, **949**
reordenar a saída com um índice de argumento, 1102
repaint, método da classe Component, **456**
repaint, método da classe JComponent, **486**
repetição, 84, 144
  controlada por contador, 90, 97, 99
  controlada por sentinela, 93, 94, 95, 96, 97
  definida, **90**
repetição controlada por contador, 90, 97, 99, 120, 121, 237, 597
repetição controlada por sentinela, **93**, 94, 95, 96, 97, 152, 237
repetição definida, **90**
repetição indefinida, **93**
repetição, instrução de, **83**, **84**, **88**, 94, 597
  do...while, 84, 127, 128, 144
  for, 84, 124, 144
  while, 84, **89**, 91, 97, 120, 144
repetição termina, 89
repintado, 730
replaceAll, método
  da classe Matcher, **541**
  da classe String, **540**
replaceFirst, método
  da classe Matcher, **541**
  da classe String, **540**
repositórios de arquivos, **267**
representação de array bidimensional de um labirinto, 613
representando inteiros no formato hexadecimal, 1092
representar inteiros no formato octal, 1091
Representational State Transfer (REST), 1018, 1020
reproduzindo áudio, 756
RequestBean, **952**
RequestContext, interface, **1045**
  put, método, **1045**
required, propriedade de um componente JSF, **968**
requisitos, **16**, 364
reservas, sistema, 231
reset, propriedade de um JSF Button, **996**
resolução, 105, 484
resolver o problema das Torres de Hanói com um método recursivo, 600
@Resource, anotação, **1037**
resources, elemento de um documento JNLP, **736**
resposta de servidor, **949**
respostas para uma pesquisa, 198
restart, método da classe Timer, **751**, 763
RESTful, serviços Web, **1020**
resto, operador %, 117
resto, operador de atribuição composta, %=, 102
REST (Representational State Transfer), **1018**
resultado, 904
ResultSet, interface, **914**, 918, 919
  absolute, método, **919**
  close, método, 915
  CONCUR_READ_ONLY, constante, **919**
  CONCUR_UPDATABLE, constante, **919**
  constante de concorrência, 919
  getInt, método, **914**
  getObject, método, **914**, 926
  getRow, método, **919**
  last, método, **919**
  next, método, **914**
  nome de coluna, 914
  número de coluna, 914
  TYPE_FORWARD_ONLY, constante, **918**
  TYPE_SCROLL_INSENSITIVE, constante, **918**
  TYPE_SCROLL_SENSITIVE, constante, **918**
ResultSetMetaData, interface, **914**, 919
  getColumnClassName, método, **919**
  getColumnCount, método, **914**, 919
  getColumnName, método, **919**
  getColumnType, método, **914**
ResultSetTableModel, ativa JTable para exibir os conteúdos de ResultSet, 915
retângulo, 275, 484, 487, 495, 724, 728
retângulo arredondado, 496, 506
retângulo arredondado (para representar um estado em um diagrama de estado da UML), **374**
retângulo delimitador, **144**, **496**, 498, 770
retângulo em alto relevo, **498**
retângulo em baixo relevo, **498**
retângulo preenchido, 487
retângulo tridimensional, 496
retângulo tridimensional preenchido, 496
reticências (...) em uma lista de parâmetros de método, **217**
Retornando indicadores de erros a partir de métodos, exercício, 275
retorno de carro, 35
return, instrução, 591
return, palavra-chave, **65**, 156, 161, 1069

Return, tecla, 726
Returns:, nota, **XXXIX**
@return, tag javadoc, **XXXIX**
reutilização de software, 7, 264, 279, 671, XXVII
reutilizar, 16, 37
reversão recursiva, **608**
reverse, 650
reverse, método da classe Collections, 645, **650**
reverse, método da classe StringBuilder, **528**
reverseOrder, método da classe Collections, **646**
reverter uma transação, **939**
RGB, valor, 491
RGB, valores, **176**, 486
Richards, Martin, 6
RIGHT, constante da classe FlowLayout, **463**
Ritchie, Dennis, 6
.rmi, extensão de arquivo, **756**, **758**
robusto, 39
roda de mouse, 450
rolagem, 444, 447
rolagem automática, 447
rolagem vertical, 470
rolando dois dados, 170
rollback, método da interface Connection, **939**
rollover Icon, **437**
Rotação de imagens, exercício, 766
rotate, método da classe Graphics2D, **508**
rótulo, 299, **425**, **LIV**
rótulo em uma switch, **132**
rótulos para marcas de verificação, 769
RoundRectangle2D, classe, 484
RoundRectangle2D.Double, classe, **503**, 506
RowFilter, classe, **924**
RowSet conectado, **924**
RowSet desconectado, **924**
RowSet, documentação (java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html), 924
RowSet, guia de introdução
  java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/rowsetImpl.html, 924
RowSet, interface, **924**
RSS feed, XXVI
RSS feeds, **18**
RSS Resource Center, XXVIII
Ruby on Rails, **19**
Rumbaugh, James, 16
run, comando de depurador, **1079**
run, método da interface Runnable, **808**, 887
Runnable, interface, 329, **808**, 888
  run, método, **808**, 887
Run, opção no NetBeans, 1022
RuntimeException, classe, **343**, 349

S

SaaS (Software as a Service), xxii, 19
saída, 32, 35
saída alinhada à direita, **126**
saída formatada, 1097
  -, (sinal de subtração), flag de formatação, **126**
  , , (vírgula), flag de formatação, **127**
  %f, especificador de formato, **71**
  0, flag, **196**, 244
  alinhamento à direita, **126**, 1091
  alinhamento à esquerda, **126**, 1091
  alinhando pontos de fração decimal na saída, 1091
  arredondando, 1091
  boolean, valores, **138**
  caractere de conversão, 1091
  caracteres de sufixo de conversão de data e hora, **1094**
  composições de data e hora, 1094
  datas, 1091
  formato exponencial, 1091
  inserindo caracteres literais, 1091
  inteiros no formato hexadecimal, 1092
  inteiros no formato octal, 1091
  largura de campo, **126**, 1091
  números de ponto flutuante, **71**
  precisão, **71**, 1091
  separador de agrupamento, **127**
  sinal de subtração (-), flag de formatação, **126**
  vírgula (,), flag de formatação, **127**
saindo da instrução for, 135
SalariedEmployee, classe concreta que estende classe abstract Employee, 313
SalariedEmployee, classe que implementa o método getPaymentAmount da interface Payable, 328
salário bruto, 115
Salesforce, 18
SalesForce, XXVII
Salesforce.com, XXVII
SansSerif, fonte do Java, 492
saturação, **491**
SavingsAccount, Classe (exercício), 275
Scanner, classe
  documentação (java.sun.com/javase/6/docs/api/java/util/Scanner.html), 258
  hasNext, método, **131**
  java.sun.com/javase/6/docs/api/java/util/Scanner.html, 258
  next, método, **62**
  nextDouble, método, **72**
  nextLine, método, **62**
Scanner, classe, 37, **38**
Scanner de phishing, 588
Scanner de spam, 550
Scanner de spam, serviço Web, 1065
scheduler de thread, **807**
Screen, classe (estudo de caso ATM), 367, 369, 377, 382, 383, 384, 385, 386, 393
script de shell, **562**
script (Unicode), **XXXIII**
SCROLL_TAB_LAYOUT, constante da classe JTabbedPane, **792**
seção administrativa do computador, 4
seção de armazenamento do computador, 3
seção de detalhe de método na API, **1076**
seção de detalhes de construtor na API, **1076**
seção de detalhes de um campo na API, **1075**
seção de entrega do computador, 3
seção de recebimento do computador, 3
seção de resumo de classe aninhada na API, **1074**
seção de resumo de construtor na API, **1074**
seção de resumo de método na API, **1075**
seção de resumo de um campo na API, **1074**
Seção especial
  construindo seu próprio compilador, 695
  construindo seu próprio computador, 236
  exercícios de manipulação avançada de string, 546
  projetos desafiadores de manipulação de string, 548
Second Life, 18
SecurityException, classe, **561**
See Also:, nota, **XXXVII**
@see, tag javadoc, **XXXVII**
segundo refinamento, 100
segundo refinamento no refinamento passo a passo de cima para baixo, 94
segurança, 10
segurança de thread, **815**, 840
segurança de tipo em tempo de compilação, **671**
SEI (*service endpoint interface*), **1022**, 1024
seleção, 84, 144
seleção dupla, 144
seleção dupla, instrução, **84**
seleção, instrução, **83**, **84**
  if, 84, 85, 129, 144
  if...else, 84, **85**, 97, 129, 144
  switch, 84, **129**, 133, 144
seleção múltipla, instrução, **84**
  if, 85
seleção única, instrução de, 144
selecionando dados a partir de uma tabela, 900
Selecionando formas, exercício, 513
selecionar cada dígito, 54
selecionar um item de um menu, 428
SELECT, SQL, palavras-chave, **903**, 904, 905, 906
_self, frame-alvo, **863**
semear números aleatórios, **164**
send, método da classe DatagramSocket, **879**
senha, 429
seno, 157
seno trigonométrico, 157
separador de diretório, **267**
separador de grupos (saída formatada), **127**
SequenceInputStream, classe, 578
sequência, 84, 144, **639**, **709**
sequência de escape, **35**, 38, 558, 1103, 1108
  barra invertida, 35
  ", aspas duplas, 35
  \t, tabulação horizontal, 35
  nova linha, , 38
  nova linha, \l, 35
sequência de mensagens na UML, **384**
sequência, estrutura, **83**
Serializable, interface, 329, **570**
serialização de objeto, **570**, 872
serialized-form.html, gerado por javadoc, **XLII**
série infinita, 152
Serif, fonte do Java, 492
ServerSocket, classe, **866**, 871, 887
  accept, método, **866**, 871
service, método da interface Servlet, **951**
ServiceManager, classe, **746**
  lookup, método, **746**
Services, guia no NetBeans, 997
serviço de sistema, 866
serviço de uma classe, 245
serviço orientado para conexão, **859**
serviço sem conexão, **859**, 877
serviços Web, **18**
  adicionando uma referência de serviço Web a um aplicativo, **1024**
  artefatos do lado do cliente, **1025**
  artefatos do lado do servidor, **1021**
  Build, opção no NetBeans, 1022
  classe proxy, **1022**, 1025
  Clean and Build, opção no NetBeans, 1022
  Clean, opção no NetBeans, 1022
  consumir um serviço Web, **1019**, 1060
  Deploy, opção no NetBeans, 1022
  do servidor de aplicativo GlassFish Tester página Web, 1023
  @GET, anotação, **1029**
  host de serviço Web, **1019**
  implantando um serviço Web, 1022
  JAX-RS, **1018**
  JAX-WS, **1018**, 1025
  MessageContext, interface, **1037**
  name, atributo da anotação @WebParam, **1022**

name, atributo da anotação @WebService, **1022**
@Path, anotação, **1029**
@PathParam, anotação, **1029**
POJO (Plain Old Java Object), **1021**
processando tipos definidos pelo usuário, 1051
@Produces, anotação, **1029**
publicando um serviço Web, **1060**
Publicando um serviço Web com um aplicativo Java SE 6 padrão (today.java.net/pub/a/today /2007/07/03/jax-ws-web-services-without-ee-containers.html), 1022
publicar um serviço Web, **1019**
referência de serviço Web, 1025
RequestContext, interface, **1045**
@Resource, anotação, **1037**
REST, **1018**
Run, opção no NetBeans, 1022
serviceName, atributo da anotação @WebService, **1022**
SOAP, 1025
Testando um serviço Web de outro computador, 1024
testar um serviço Web, 1023
@WebMethod, anotação, **1022**
@WebParam, anotação, **1022**
@WebService, anotação, **1021**
WebServiceContext, interface, 1037
serviço Web, **1018**, XXVI
implementado como uma classe, 1019, 1060
publicando um serviço Web, 1022
Serviço Web de Lista telefônica com JSON, 1065
Serviço Web de Neutralidade Sexual, **1065**
servidor, **4**, **859**
servidor de aplicativo, 947
servidores de aplicativo
Apache Tomcat (tomcat.apache.org), 1021
GlassFish (glassfish.dev.java.net), 1021
JBoss Application Server (www.jboss.com/products/platforms/application), 1021
servidor espera conexões de clientes, 866
servidor multiencadeado, 896, 897
servidor Web, 866, **948**
servlet, **950**
Servlet, interface, **950**
destroy, método, 951
init, método, 950
service, método, **951**
sessão, 971
sessão, monitoramento, **971**
em serviços Web, 1035
SessionBean, **952**
seta, 83
SET, cláusula de SQL, **909**
set, comando de depurador, **1081**
seta de navegabilidade na UML, **391**
seta de rolagem, **444**
seta de transição, 86, 89
na UML, **83**
seta de transição (na UML), 89
Set, interface, 637, **639**, 656, 658
seta, teclas, 458
setAlignment, método da classe FlowLayout, **463**
set, método
da classe BitSet, **LI**
de interface ListIterator, **643**
setAttribute, método da interface HttpSession, **1037**
setAutoCommit, método da interface Connection, **939**
setBackground, método da classe Component, 223, **447**, 490
setBounds, método da classe Component, 460
setCharAt, método da classe StringBuilder, **528**
setColor, método da classe Color, **487**
setColor, método da classe Graphics, **177**, **487**, 506
setCommand, método da interface JdbcRowSet, **925**
setConstraints, método da classe GridBagLayout, **796**
setDefaultCloseOperation, método da classe JFrame, **107**, **428**, **772**
setDelay, método da classe Timer, **765**
setDisabledTextColor, método da classe JTextComponent, **460**
setEditable, método da classe JTextComponent, **431**
setErr, método da classe System, **555**
setFileSelectionMode, método da classe JFileChooser, **580**
setFixedCellHeight, método da classe JList, **449**
setFixedCellWidth, método da classe JList, **449**
setFont, método da classe Component, **440**
setFont, método da classe Graphics, **492**
setForeground, método da classe JComponent, **778**
setHint, método da classe Manager, **759**
setHorizontalAlignment, método da classe JLabel, **428**
setHorizontalScrollBarPolicy, método da classe JScrollPane, **470**
setHorizontalTextPosition, método da classe JLabel, **428**
setIcon, método da classe JLabel, **427**
set, método, **66**, 248
setIn, método da classe System, **555**
setInverted, método da classe JSlider, **770**
setJMenuBar, método da classe JFrame, **778**
setLayout, método da classe Container, **427**, 460, 465, 792
setLineWrap, método da classe JTextArea, **470**
setListData, método da classe JList, **449**
setLocation, método da classe Component, 460, **772**
setLookAndFeel, método da classe UIManager, **784**
setMajorTickSpacing, método da classe JSlider, **772**
setMaximumRowCount, método da classe JComboBox, **444**
setMnemonic, método da classe AbstractButton, **777**
setOpaque, método da classe JComponent, **455**, 457
setor, 499
setOut, método de System, **555**
setPage, método da classe JEditorPane, **865**
setPaint, método da classe Graphics2D, **505**
setPaintTicks, método da classe JSlider, **772**
setPassword, método da interface JdbcRowSet, **925**
setProperty, método da classe Properties, **661**
setRolloverIcon, método da classe AbstractButton, **438**
setRowFilter, método da classe JTable, **924**
setRowSorter, método da classe JTable, **923**
setSeed, método da classe Random, 168
setSelected, método da classe AbstractButton, **778**
setSelectionMode, método da classe JList, **447**
setSize, método da classe Component, 460, 772
setSize, método da classe JFrame, **107**, **428**
setString, método da interface PreparedStatement, **927**, 932
setStroke, método da classe Graphics2D, **505**
setText, método da classe JLabel, 300, **428**
setText, método da classe JTextComponent, 470
setToolTipText, método da classe JComponent, **427**
setUrl, método da interface JdbcRowSet, **925**
setUsername, método da interface JdbcRowSet, **925**
setVerticalAlignment, método da classe JLabel, **428**
setVerticalScrollBarPolicy, método da classe JScrollPane, **470**
setVerticalTextPosition, método da classe JLabel, **428**
setVisible, método da classe Component, **428**, 465, 772
setVisible, método da classe JFrame, **107**
setVisibleRowCount, método da classe JList, **446**
Shape, hierarquia de classes, 281, 303
Shape, objeto, **505**
shell, **32**
shell, prompt no Linux, **10**
Shift, 460
Short, classe, **638**
short, tipo primitivo, 129, 1069, 1070
promoções, 162
shortcut, elemento de um documento JNLP, **735**
show, método da classe JPopupMenu, **781**
showDialog, método da classe JColorChooser, **490**
showDocument, método da interface AppletContext, **860**, 863
showInputDialog, método da classe JOptionPane, **75**, **422**
showMessageDialog, método da classe JOptionPane, **74**, **422**
Show Line Numbers, 958
showOpenDialog, método da classe JFileChooser, **581**
showStatus, método da classe Applet, **753**
shuffle, 201
algoritmo, 648
shuffle, método da classe Collections, 645, **648**, 649
shutdown, método da classe ExecutorService, **811**
signalAll, método da interface Condition, **836**
signal, método da interface Condition, **836**, 838
Silicon Graphics, 758
Simbad Robotics Simulator Project, 767
símbolo de estado de ação, **83**
símbolo de infinidade, 903
símbolos especiais, **553**
SimpleGraph, applet, 724
Simple Object Access Protocol (SOAP), 1018, 1020
Simpletron Machine Language (SML), 236, 695
Simpletron, simulador, 237, 239, 695, 766
simulação, 164
moeda, lançamento, 187
Simulação
Lebre e a tartaruga, A, 513
Simulação: A lebre e a tartaruga, 234
simulação de software, 236
simulação de supermercado, 718
simulador, 236
simulador de computador, 237
simulador de robótica, 767
Simulador de Simpletron baseado em multimídia, exercício, 766
simular um clique de botão direito do mouse em um mouse de um botão, 454
simular um clique no botão do meio do mouse com um mouse de um ou dois botões, 454
sinais de cifrão ($), 30
sinal de adição (+) indicando visibilidade pública na UML, 391
sinal de subtração (-), flag de formatação, **126**
sinal de subtração (-) indicando visibilidade privada na UML, 391
sin, método da classe Math, 157
Since:, nota, **XL**
@since, tag javadoc, **XL**
sincronização, 826
sincronizar acesso a uma coleção, 639
sincronizar threads, **805**

SINGLE_INTERVAL_SELECTION, constante de interface ListSelectionModel, **447**
SINGLE_SELECTION, constante de interface ListSelectionModel, **447**
SingleSelectOptionsList, classe, 964
Single-Threaded Execution, padrão de design, LIX, LXVII
Singleton, padrão de design, LIX, **LX**
sintaxe, 30
sintaxe de chamada de construtor de superclasse, **291**
síntese de fala, 767
sistema, **365**
sistema de composição, 516
sistema de coordenadas, **105**, **484**, 485
sistema de gerenciamento de banco de dados relacional (RDBMS), **899**, 950
sistema de gerenciamento de bancos de dados (*DataBase Management System* — DBMS), **553**, **899**
sistema de nome de domínio (*domain name system* — DNS), servidor, **948**
sistema de números octal (base 8), 188, II
sistema de pesquisa, 949
Sistema de reservas de passagens aéreas, 231
sistema de software de comando e controle, 804
sistema de telefonia, 877
sistema, estrutura, **365**
sistema numérico hexadecimal (base 16), 188, 239, II
sistema operacional, **4**
sistema, requisitos, **364**
sistemas de número, 533
sistemas de pesquisa de código e sites de código, xxii, xxvii
size, método
  da classe ArrayBlockingQueue, **824**
  da classe ArrayList<T>, **222**
  da classe BitSet, **LI**
  da classe PriorityQueue, **656**
  de interface List, **641**, 643
  de interface Map, **661**
Skype, 18
sleep, método da classe Thread, **809**, 819, 820
Smashforce, XXVIII
SML, 695
SMS Language, 550
SMS, serviço Web, 1065
SOAP, formato wire, **1019**
SOAP (Simple Object Access Protocol), xxii, **1018**, 1019, 1020, 1025
  envelope, **1020**
  mensagem, **1019**
  wire, formato, **1019**
sobrecarga de método, **174**
sobrecarregar métodos genéricos, 677
sobrecarregar um método, **174**
social bookmarking, 17, XXVII
socket, **859**
Socket, classe, **866**, 876, 887, 888, LXVIII
  close, método, **867**
  getInetAddress, método, **871**
  getInputStream, método, **866**, 867
  getOutputStream, método, **866**
socket de datagrama, **859**, 877
SocketException, classe, **877**
SocketImpl, classe, LXVIII
software, 2, 3
Software as a Service (SAAS), **19**
software frágil, **293**
software, modelo, 237
software profissional (business software), 7
solicitação assíncrona, **1006**
solicitação, método de, **949**
solicitação síncrona, **1005**
solução fatorial iterativa, 598
som, 730, 743
somar elementos de um array, 195
sombrear um campo, **172**
som, sistema, **756**
sons (www.soundcentral.com), 762
sort, 219
sort, método
  da classe Arrays, **219**, **619**
  da classe Collections, **645**
SortDemo, applet, 724
SortedMap, interface, **658**
SortedSet, interface, **657**, 658
  first, método, **658**
  last, método, **658**
Source, visualização no Netbeans, XIII
SOUTH, constante da classe BorderLayout, **452**, 463
SOUTH, constante da classe GridBagConstraints, **793**
SOUTHEAST, constante da classe GridBagConstraints, **793**
SOUTHWEST, constante da classe GridBagConstraints, **793**
span, elemento, **954**
.spl, extensão de arquivo, **758**
-splash, opção, **XXI**
-splash, opção de linha de comando para o comando java, **XXI**
SplashScreen, classe, **XXI**
split, método da classe String, **535**, 540
spooler, **707**
spooling, 707
spooling de impressão, **707**, **818**
SpreadSheet, applet, 725
.sql, extensão de nome de arquivo, 911
SQL, 899, 900, 903, 908
  DELETE, instrução, 903, **909**
  FROM, cláusula, 903
  GROUP BY, 903
  IDENTITY, palavra-chave, **928**
  INNER JOIN, cláusula, 903, **907**
  INSERT, instrução, 903, **908**
  LIKE, cláusula, 905
  ON, cláusula, **907**
  ORDER BY, cláusula, 903, **905**, 906
  SELECT, consulta, **903**, 904, 905, 906
  SET, cláusula, **909**
  UPDATE, instrução, 903
  VALUES, cláusula, **908**
  WHERE, cláusula, **904**
SQLException, classe, **913**, 914, 927
SQLFeatureNotSupported-Exception, classe, 919
SQL, instrução, 938, 939
SQL, palavras-chave, 903
SQL, script, 911
SQL (Structured Query Language), 927
sqrt, método da classe Math, 157, 162
Stack, classe, **654**, 655, 704
  isEmpty, método, **655**
  peek, método, **655**
  pop, método, **655**
  push, método, **655**
Stack, classe genérica, 678
  Stack< Double >, 683
  Stack< Integer >, 683
Stack, declaração de classe genérica, 678
Stack, documentação de classe (java.sun.com/javase/6/docs/api/java/util/Stack.html), 654
StackTraceElement, classe, **351**
  getClassName, método, **351**
  getFileName, método, **351**
  getLineNumber, método, **351**
  getMethodName, método, **351**
start, método da classe JApplet, **728**, 730, 732
start, método da classe Timer, **751**
start, método da interface Player, **760**
startsWith, método da classe String, **521**
stateChanged, método da interface ChangeListener, **772**
Statement, interface, **914**, 927
  close, método, 915
  executeQuery, método, **914**
state, objeto, LXVI
State, padrão de design, LIX, LXII, LXVI
  classe state, **LXII**
  objeto de contexto, **LXII**
  state, objeto, **LXII**
  subclasse state, **LXII**
static
  campo (variável de classe), **258**
  importação, **261**
  importação simples, **261**
  importação sob demanda, **261**
  membro de classe, 258
  método, 60, **74**, 126
  palavra-chave, 156, 1069
  variável de classe, 259
Static Text, componente JSF, **953**, 961
step, comando de depurador, **1083**
step up comando de depurador, **1083**
stop, comando de depurador, **1080**
stop, método da classe JApplet, **728**
stop, método da classe Timer, **752**
stop, método da interface AudioClip, **756**
store, método da classe Properties, **663**
strategy, objeto, LXVI
Strategy, padrão de design, LIX, LXII, LXVI
strictfp, palavra-chave, 1069
string, **32**, 163
  literal, **32**
String, classe, 516
  charAt, método, **518**, 528
  compareTo, método, 519, 521
  concat, método, **524**
  endsWith, método, **521**
  equals, método, 519, 520
  equalsIgnoreCase, método, 519, 521
  format, método, **74**, **244**, 1104
  getChars, método, **518**
  imutável, **260**
  indexOf, método, **522**
  lastIndexOf, método, **522**
  length, método, **518**
  matches, método, **536**
  regionMatches, método, **519**
  replaceAll, método, **540**
  replaceFirst, método, **540**
  split, método, **535**, 540
  startsWith, método, **521**
  substring, método, 523
  toCharArray, método, **525**, 612
  toLowerCase, **643**
  toLowerCase, método, **525**
  toUpperCase, **643**
  toUpperCase, método, **525**
  trim, método, **525**
  valueOf, método, **525**
String, classe, métodos de pesquisa de classe, 523
String, objeto, imutável, **517**
StringBuffer, classe, **526**
StringBuilder, classe, 516, **526**

append, método, **529**
capacity, método, **527**
charAt, método, **528**
construtores, 527
delete, método, **530**
deleteCharAt, método, **530**
ensureCapacity, método, **527**
getChars, método, **528**
insert, método, **530**
length, método, **527**
reverse, método, **528**
setCharAt, método, **528**
string, concatenação, 260
string de caractere, **32**
string de consulta, **949**
string de formato, **36**
string delimitadora, 536
StringIndexOutOfBoundsException, classe, **523**, 528
StringReader, classe, **578**
string vazia, **433**, **517**
StringWriter, classe, **578**, **1030**
Strmz, XXVIII
Stroke, objeto, **505**, 506
Stroustrup, Bjarne, 6, 336
Structured Query Language (SQL), **899**, 900, 903
suavização (anti-aliasing), 726
subárvore direita, **709**, 713, 719
subárvore esquerda, **709**, 713, 719
subclasse, **107**, **279**, 395, LX
subclasse concreta, 313
subclasse personalizada de classe JPanel, 455
subdiretório, 725
sublinhado (_) caractere curinga SQL, 904, 905
sublista, **643**
subList, método de interface List, **643**
submenu, **773**
submenu expandido, 778
submissor em um formulário virtual, **1006**
submit, método da classe ExecutorService, **851**
subprotocolo para comunicação, **913**
subscript (índice), **190**
subsistema, LXVIII
substantivos compostas no documento de requisitos, 366
substantivos compostos no documento de requisitos, 371
substitutos, **XXX**
substring, método da classe String, 523
operador, -, 41
subtração, operador de atribuição composta, -=, 102
subtract, método da classe BigInteger, **595**
Sun Audio, formato de arquivo (extensão .au), **756**, 758
Sun Microsystems, 2, XXIX, LVIII
super.paintComponent(g), 107
super, palavra-chave, 281, 297, 1069
chamar construtor de superclasse, 291
superclasse, **107**, **279**, 395, LXIV
construtor, 284
construtor padrão, 284
direta, **279**, 280
indireta, **279**, 280
método sobrescrito em uma subclasse, 297
sintaxe da chamada de construtor, **291**
superclasse abstrata, **309**, 398
supercomputador, **3**
suspender execução de um applet, 730
.swf, extensão de arquivo, **758**
swing.properties, arquivo, xxxii, 420
SwingConstants, interface, 329, **428**, 772
Swing Event Package, 164
Swing GUI, APIs, 419
Swing GUI, pacote de componentes, 164
SwingSet3, demo (download.java.net/javadesktop/swingset3/SwingSet3.jnlp), 419
SwingUtilities, classe, **784**, **871**
invokeLater, método, **871**
updateComponentTreeUI, método, **784**
SwingWorker, classe, **841**
cancel, método, **850**
doInBackground, método, 841, 842
done, método, 841, 842
execute, método, 841
get, método, 841
isCancelled, método, **847**
process, método, 841, 848
publish, método, 841, 847
setProgress, método, 841, 847
switch, instrução de múltipla seleção, **129**, 133, 144, 167, 1069
case, rótulo, 132
default, caso, **132**, 133, 167
diagrama de atividades com instruções break, 134
expressão controladora, **132**
switch, instrução de seleção múltipla, 84
switch, lógica, 134
Sybase, 899
Sybase, Inc., XXIX
synchronized
instrução, **812**
método, **812**
palavra-chave, 664, **812**, 1069
System, classe
arraycopy, **219**, 221
currentTimeMillis, método, 613
exit, método, **345**, **561**
setErr, método, **555**
setIn, método, **555**
setOut, **555**
System.err, (fluxo de erro padrão), **341**, 554, 577, 1091
System.in, (fluxo de entrada padrão), 554
System.out
print, método, 32, **34**
printf, método, **35**
println, método, 32, 34
System.out, (fluxo de saída padrão), 554, 577
System.out, (objeto de saída padrão), **32**
System.out.print, método, 32, 34
System.out.printf, método, **35**
System.out.println, método, 32, 34
SystemColor, classe, **505**
SystemTray, classe, **XXIV**
addTrayIcon, método, **XXIV**
getDefaultSystemTray, método, **XXIV**
removeTrayIcon, método, **XXIV**

T

tabela, 209
tabela de banco de dados relacional, 900
tabela de hash, colisões, 659
tabela de precedência de operadores, 1066
tabela de um banco de dados, **900**
tabela de valores, 209
tabela, elemento, 209
tabelas-verdade
para operador ^, 138
para operador !, 138
para operador &&, 136
para operador ||, 136
tabela-verdade, **136**
tabela-verdade, negação lógica ou operador NÃO lógico (!), 138
Tab, tecla, 31
TableModel, interface, **915**
addTableModelListener, **915**
getColumnClass, método, **915**, 919
getColumnCount, método, **915**, 919
getColumnName, método, **915**, 919
getRowCount, método, **915**
getValueAt, método, **915**
removeTableModelListener, **915**
TableModelEvent, classe, 923
Table componente (JSF), 997
internalVirtualForms, propriedade, **998**
paginationControls, propriedade, **998**
Table componente JSF, 995
TableRowSorter, classe, **923**
tabulação, 1103
tabulação, caractere, \t, 35
tabulação horizontal, 35
tabulação, paradas, 31, 35
tabulação (para recuo em código), 30
tabuleiro de damas, padrão, 53
exercício, 741
tag de abertura, **735**
tag de fechamento, **735**
tag (em um documento de XHTML), **729**
tags personalizados, **951**
tailSet, método da classe TreeSet, **658**
take, método da classe BlockingQueue, **823**, 824
tamanho da área de exibição do applet, 729
tamanho de fonte, **492**
tamanho de uma variável, **40**
tan, método da classe Math, 157
tangente, 157
tangente trigonométrica, 157
tarefa, **4**
taxa de juros, 125
Tax Plan Alternatives, 153
TCP (Transmission Control Protocol), **859**
Technorati, **18**
Technorati APIs, XXVIII
tecla constante, 460
tecla de ação, **458**
tecla de atalho, 773
tecla de função, 458
teclado, 3, 4, 419
teclado, evento, **435**, 458
tecla modificadora, 460
tela, 3, 4
tela de splash, **XXI**
Template Method, padrão de design, LIX, LXII, LXVI
tempo de execução constante, **618**
tempo de execução linear, **618**
tempo de execução logarítmico, **622**
tempo de execução quadrático, **618**
Tempo para calcular números de Fibonacci, exercício, 613
terminação, preparação, **258**, 298
terminação, teste de, 597
terminado, estado, 806
terminal, 4
terminar com sucesso, 561
terminar um aplicativo, 777
terminar um loop, 94
término anormal de um programa, 199
término, fase, 94
Testador de tempo de reação/precisão de reação, exercício, 766
Testando a classe de classificação por inserção, 627

Testando a classe de classificação por intercalação, 631
Testando a classe de classificação por seleção, 625
Testando um serviço Web de outro computador, 1024
testar um serviço Web, 1023
Text Area, componente JSF, **978**
texto de leitura, 425
texto de template fixa, **951**
Text Field componente
  autocompletar, 1006
  autoComplete, atributo, **1010**
  autoCompleteExpression, atributo, **1010**
Text Field, componente JSF, 961, **964**
texto fixo, 39, **1091**
  em uma string de formato, **1091**
texto ou ícones não editáveis, 423
texto para voz (text-to-speech — TTS), 767
texto que pula, 724
texto selecionado em um JTextArea, 468
Text Package, 163
textura de preenchimento, 506
TexturePaint, classe, **484**, 505, 506
The FairTax, 153
The Free Site (www.thefreesite.com), 762
The Java™ Language Specification (java.sun.com/docs/books/jls/), 41
this
  palavra-chave, 246, 259, 1069
  para chamar um outro construtor da mesma classe, 250
  referência, **246**
Thompson, Ken, 6
thread, **341**, **486**, 730, LIX
  agendamento, **806**, 820
  agendamento de prioridade, 807
  ciclo de vida, 805
  de execução, **804**
  estado, **805**
  sincronização, 663, **812**
Thread, classe, **807**, 809
  currentThread, método, 813
  interrupt, método, **809**
  sleep, método, **809**
thread adormecida, **806**
Thread.MAX_PRIORITY, 807
Thread.MIN_PRIORITY, 807
Thread.NORM_PRIORITY, 807
thread consumidor, **818**
thread de despacho de evento (*event-dispatch thread* — EDT), **840**
thread de despacho do evento (*event-dispatch thread* — EDT), **486**, 871
thread de espera, 827
thread, estados
  bloqueado, **806**, 812
  espera, **806**
  espera sincronizada, **806**
  executando, **806**
  executável, **805**, 806
  morto, 806
  novo, **805**
  pronto, **806**
  terminado, **806**
thread principal, **810**
thread produtor, **818**
threads concorrentes, 823
threads de execução, **804**
ThreeDimensionalShape, classe, 303
throughput, **4**
Throwable, classe, **343**, 349
  documentação (java.sun.com/javase/6/docs/api/java/lang/Throwable.html), 343
  getMessage, método, **349**
  getStackTrace, método, **349**
  hierarquia, 343
  printStackTrace, método, **349**
throw, instrução, **347**
throw, palavra-chave, **348**, 1069
throws, cláusula, **342**
throws, palavra-chave, **342**, 1069
@throws, tag javadoc, **XXXVIII**
Tic-Tac-Toe 3-D multithreaded, 897
TicTacToe, 277
  applet, 725
  exercício, 277
Tic-Tac-Toe (jogo da velha), 882
Tic-Tac-Toe multiencadeado, programa, 897
Timer, classe, 512, **751**, 752
  getDelay, método, **765**
  isRunning, método, **751**
  restart, método, **751**
  setDelay, método, **765**
  start, método, **751**
  stop, método, **752**
tipo, **38**
tipo bruto, **684**
tipo de conjunto de resultados, **918**
tipo de dados abstrato (ADT), 242
tipo de retorno, 65
  de um método, **59**, 65
  na UML, 377, 381
tipo de uma variável, **40**
tipo parametrizado, **678**
tipo, parâmetro, **674**, 678, 683
  escopo, 679
  seção, **674**, 678
tipo por referência, **67**, 268
tipo primitivo, **38**, 67, 105, 162
  boolean, 1082
  byte, 129
  char, 38, 129
  double, **38**, **70**, 95
  float, **38**, **70**
  int, 38, 95, 102, 129
  nomes são palavras-chave, 38
  passado por valor, 205
  promoções, 162
  short, 129
tipo, variável, **674**
title, elemento de um documento JNLP, **735**
titles, tabela do banco de dados books, 901, 902
toArray, método de interface List, **644**
toArray, método de List, 645
toBinaryString, método da classe Integer, **XLV**
toCharArray, método da classe String, **525**
toCharArray, método de String, 612
toJson, método da classe Gson, **1033**
token de uma String, **535**
tokenização, 535
tolerante a falhas, 39, **336**
toLowerCase, método da classe Character, **533**
toLowerCase, método da classe String, **525**, **643**
tom, **491**
TOP, constante da classe JTabbedPane, **792**
_top, frame-alvo, **863**
topo, **93**
Torres de Hanói, **598**
Torres de Hanói para o caso com quatro discos, 599
toString, método
  da classe ArrayList, 645, 688
  da classe Arrays, 541, 616
  da classe BitSet, **LI**
  da classe Formatter, 1104
  da classe Object, 284, 299
total, **90**, 94
toUpperCase, método da classe Character, **533**
toUpperCase, método da classe String, **525**, **643**
toURI, método da classe File, **761**
toURL, método da classe URI, **761**
trabalho, **4**
tradução, **5**
TRAILING, constante de alinhamento em GroupLayout, XIII
transação, **939**
Transaction, classe (estudo de caso ATM), 395, 396, 397, 398, 416
transferência de controle, **83**, 238, 239
Transição aleatória de imagens, exercício, 765
transição entre estados na UML, **374**, 375
transição (na UML), **83**
transient, palavra-chave, **572**, 1069
translate, método da classe Graphics2D, **508**
transmissão baseada em fluxo orientada para conexão, 877
transmissão sem conexão, **877**
transparência de JComponent, **455**
tratamento de evento, **428**, 431, 432
  fonte de evento, **434**
tratamento de evento de mouse, 449
tratamento de eventos de teclado, 458
tratamento de exceções, **336**
tratar uma exceção, 339
TrayIcon, classe, **XXIV**
TrayIcon, pacote de demonstração, XXIV
TreeMap, classe, **658**
TreeSet, classe, **656**, 657, 658
  headSet, método, **658**
  tailSet, método, **658**
Triângulos aleatórios, exercício, 512
triângulos gerados aleatoriamente, 512
trim, método da classe String, **525**
trimToSize, método da classe ArrayList<T>, 221
Triplos de Pitágoras, 152
trocando valores, 622, 624
true, **43**, 1069
true, palavra reservada, 85, **86**
truncado, 560
truncar, 41
truncar parte fracionária de um cálculo, **93**
try, bloco, **340**, 351
  termina, 341
try, bloco, envolvente, 349
try, instrução, **342**
try, palavra-chave, **340**, 1069
TwoDimensionalShape, classe, 303
Two-Phase Termination, padrão de design, LIX, LXVII
TYPE_FORWARD_ONLY, constante, **918**
TYPE_INT_RGB, constante da classe BufferedImage, **506**
TYPE_SCROLL_INSENSITIVE, constante, **918**
TYPE_SCROLL_SENSITIVE, constante, **918**
TypePad ATOM, XXVII
Types, classe, **914**

U

U+aaaa (convenção notacional Unicode), **XXX**
UDP (User Datagram Protocol), **859**, 877
UIManager, classe, **784**
  getInstalledLookAndFeels, método, **784**
  LookAndFeelInfo, classe aninhada, **784**
  setLookAndFeel, método, **784**

UIManager.LookAndFeelInfo, classe
  getClassName, método, **784**
UIViewRoot, classe, **953**
último a entrar, primeiro a sair (*last in*, *first out* — LIFO), **162**
último a entrar, primeiro a sair (*last in*, *first out* — LIFO),, 681
uma instrução por linha, 45
UML, diagrama de atividades
  círculo sólido incluído em um círculo aberto (para representar o fim de uma atividade) na UML, 375
  círculo sólido (para representar um estado inicial) na UML, 375
  pequeno símbolo de losango (para representar uma decisão) na UML, 375
UML, diagrama de casos de uso na
  ator, **364**
  caso de uso, 364
UML, diagrama de classes, **368**
  compartimento de atributos, 372
  compartimento de operações, 377
UML, diagrama de estado
  círculo sólido (para representar um estado inicial) na UML, **374**
  retângulo arredondado (para representar um estado) na UML, **374**
UML, diagrama de sequência
  ativação, **385**
  ponta de seta, 385
UML, parceiros, **17**
UML (Unified Modeling Language), **15**, 361, 365, 368, 372, 374, 395
  agregação, **369**
  aspas francesas (« e »), **69**
  associação, **368**
  atributo, **15**
  Centro de Recursos (www.deitel.com/UML/), 366
  círculo sólido, **84**
  círculo sólido cercado por um círculo vazio, 84
  compartimento em um diagrama de classes, **60**
  condição de guarda, **85**
  de um para um, relacionamento, **370**
  diagrama, 365
  diagrama de atividades, 83, 86, 89, 124, 128
  diagrama de classes, **60**
  diagrama elidido, **368**
  estado final, **84**
  linha pontilhada, **84**
  losango, 85
  losango sólido representando composição, **369**
  losango vazio representando agregação, **369**
  multiplicidade, **368**
  nome de papel, **369**
  nota, **84**
  quadro, 385
  relacionamento de muitos para um, **370**
  seta, 83
  símbolo de agregação, **89**
  Specification (www.omg.org/technology/documents/formal/uml.htm), 369
  um para muitos, relacionamento, **370**
UML (www.uml.org), 84
um para muitos, relacionamento, **903**
um para um, mapeamento, **658**
união de dois conjuntos, 276
união teórica, 276
Unicode, conjunto de caracteres, 54, 105, 134, **516**, 520, 533, **553**, 1070
Unicode Consortium, **XXIX**
Unicode, padrão, **XXIX**
único ponto de entrada, 140
único ponto de saída, 140
unidade central de processamento (*central processing unit* — CPU), **4**
unidade de aritmética e lógica (ALU), **3**
unidade de armazenamento secundária, **4**
unidade de disco, 724
unidade de entrada, **3**
unidade de memória, **3**
unidade de processamento, 4
unidade de resolução de tela, 105
unidade de saída, **3**
unidade lógica, **4**
unidades independentes, 4
Unified Modeling Language (UML), xxii, **15**, 16, 361, 365, 368, 372, 374, 395
uniforme, princípio do projeto Unicode, **XXIX**
Uniform Resource Identifier (URI), **555**, **860**, 948
Uniform Resource Locator (URL), **555**, 860, 948
universal, princípio do projeto Unicode, **XXIX**
UNIX, 32, 131, 562, 723, 781
UnknownHostException, classe, **867**
unlock, método da interface Lock, **835**, 838
unmarshal, método da classe JAXB, **1031**
UnsupportedOperationException, classe, **644**
unwatch, comando de depurador, **1086**
Upcoming.org, XXVII
UPDATE, instrução SQL, 903, **908**
updateComponentTreeUI, método da classe SwingUtilities, **784**
URI, classe
  toURL, método, **761**
URL, LXVIII
URL, classe, 755
  openStream, método, **1034**
URLStreamHandler, classe, LXVIII
  reescrevendo, 971
User Datagram Protocol (UDP), **859**, 877
uso de letras maiúsculas e minúsculas no estilo de frases, **422**
uso de letras maiúsculas e minúsculas no estilo título de livro, **422**, 435
UTF-8, **XXX**
UTF-16, **XXX**
UTF-32, **XXX**
Utilities Package, 163
Utiliza pesquisa binária para localizar um item em um array., 622

V

validação, **964**
validade, verificação, 253
validate, método da classe Container, **467**
ValidatorException, classe, 970
Validators
  Double Range, **964**
  Length, **964**, 968
  Long Range, **964**
valor absoluto, 157
valor-chave, 714
valor consistente, 243
valor correto, 243
valor de código, XXX
valor de deslocamento, **165**
valor de deslocamento (números aleatórios), 167
valor de flag, **93**
valor de semente (números aleatórios), 167
valor de sentinela, **93**, 94, 97
valor de sinal, **93**
valor de uma variável, **40**
valor de um parâmetro, 862
valores duplicados, 714
valor fictício, **93**
valor final, 121
valor inicial da variável de controle, **120**
valor inicial de um atributo, 372
valor inicial padrão, **65**
valor obsoleto, **815**
valor padrão, **65**, 105
valor para o inteiro mais próximo, 185
Valor unicode do caractere digitado, 460
valueChanged, método da interface ListSelectionListener, **447**
valueOf, método da classe String, **525**
VALUES, cláusula de SQL, **908**
values, método de um enum, **257**
v, opção do comando jar, **734**
variação na UML, 375
variável, **36**, 38
  nome, **38**, **40**
  tamanho, **40**
  tipo, **40**
  tipo por referência, **67**
  valor, **40**
variável constante, **134**, 195, 262
  deve ser inicializada, 195
variável de ambiente
  CLASSPATH, 34
  PATH, 33
variável de classe, **158**, **258**
variável de controle, **90**, **120**, 121, 122
variável de instância, **58**, 64, 70, 158
variável, escopo, 123
variável local, **63**, 91, 172, 173, 247
variável não é modificável, 262
varredura, **224**
varredura de um arco, **498**
varrer, **498**
varrer no sentido anti-horário, 498
vazamento de recurso, **345**
Vector, classe, **639**
velocidade de piscamento, 765
Vendas totais, 231
vendor, elemento de um documento JNLP, **735**
verificador de bytecode, **10**
Verificador de Ortografia, projeto, 548
Verificando com assert que um valor está dentro do intervalo, 354
Version nota, **XL**
@version, tag javadoc, **XL**
VERTICAL, constante da classe GridBagConstraints, **793**
VERTICAL_SCROLLBAR_ALWAYS, constante da classe JScrollPane, **470**
VERTICAL_SCROLLBAR_AS_NEEDED, constante da classe JScrollPane, **471**
VERTICAL_SCROLLBAR_NEVER, constante da classe JScrollPane, **471**
vi, 8
vídeo, 743, 761
videogame, 165
View, 419
vinculação dinâmica, **319**
vinculação estática, **321**
vinculação tardia, **319**
vinculando de atributo, 960, 971
vinculando um JSF Table a uma tabela de banco de dados, 997
vincular a outro nó (link), **696**
vírgula (,), 125
vírgula (,) flag de formatação, **127**
visibilidade na UML, **391**
Visual Basic, linguagem de programação, **8**

Visual C#, linguagem de programação, **8**
Visual C++, linguagem de programação, **8**
visualização em uma coleção, **643**
visualização (no MVC), LXIX
visualização tridimensional, 724
visualizando recursão, exercício, 611
visualizar uma forma de ângulos diferentes, 724
Vlissides, John, LVIII
void, palavra-chave, **31**, 59, 1069
volatile, palavra-chave, 1069
volume de uma esfera, 182, 184
volume, operação de, **639**

W

WADL (Web Application Description Language), **1030**
wait, método da classe Object, 299, **825**
watch, comando de depurador, **1085**
.wav, extensão de arquivo, **756**
Web 2.0, **17**, XXVI
Web 3.0, **18**
Web Application Description Language (WADL), 1030
@WebMethod, anotação, **1022**
  operationName, atributo, **1022**
Web Application, projeto, **1020**
@WebParam, anotação, **1022**
  name, atributo, **1022**
Web Semântica, **18**
@WebService, anotação, **1021**
  name, atributo, **1022**
  serviceName, atributo, **1022**
WebServiceContext, interface, **1037**
Web Service Description Language (WSDL), 1024
WebTime, modificação, 992
  exercício, 992
webuijsf:body, elemento JSF, **953**
webuijsf:form, elemento JSF, **953**
webuijsf:head, elemento JSF, **953**
webuijsf:html, elemento JSF, **953**
webuijsf:label, elemento JSF, **965**
webuijsf:link, elemento JSF, **953**
webuijsf:message, elemento JSF, **965**
webuijsf:meta, elemento JSF, **953**
webuijsf:page, elemento JSF, **953**
webuijsf:staticText, elemento JSF, **953**, 1001
webuijsf:table, elemento JSF, 1001
webuijsf:tableColumn, elemento JSF, 1001
webuijsf:tableRowGroup, elemento JSF, 1001
webuijsf:textField, elemento JSF, **965**
weightx, campo da classe GridBagConstraints, **793**
weighty, campo da classe GridBagConstraints, **793**
WEST, constante da classe BorderLayout, **452**, 463
WEST, constante da classe GridBagConstraints, **793**
WHERE, cláusula de SQL, 903, **904**, 905, 906, 909
while, instrução de repetição, 84, **88**, 89, 92, 97, 120, 144, 1069
  diagrama de atividades (na UML), 89
widgets, **419**
width, atributo do elemento applet-desc, 736
width de um applet em pixels, **729**
Wikipedia, 18
Window, classe, 772
  addWindowListener, método, **773**
  dispose, método, **772**
  pack, método, **787**
windowActivated, método da interface WindowListener, **773**
WindowAdapter, classe, 453, **924**
windowClosed, método da interface WindowListener, **773**, **924**
windowClosing, método da interface WindowListener, **773**
WindowConstants, interface, **772**
  DISPOSE_ON_CLOSE, constante, 772
  DO_NOTHING_ON_CLOSE, constante, 772
  HIDE_ON_CLOSE, constante, 772
windowDeactivated, método da interface WindowListener, **773**
windowDeiconified, método da interface WindowListener, **773**
window gadgets, 419
windowIconified, método da interface WindowListener, **773**
WindowListener, interface, 452, **773**, **924**
  windowActivated, método, **773**
  windowClosed, método, **773**, **924**
  windowClosing, método, **773**
  windowDeactivated, método, **773**
  windowDeiconified, método, **773**
  windowIconified, método, **773**
  windowOpened, método, **773**
windowOpened, método da interface WindowListener, **773**
Windows, 5, 8, 131, 562, 723
Windows, aparência e funcionamento, 769
Windows Performance Package, 758
Windows Vista, 1024
Windows Wave, formato de arquivo (extensão .wav), **756**
Windows XP
  acesso HTTP a partir de outros computadores da rede, 1024
  permitir solicitações Web, 1024
WireFrame, applet, 725
Withdrawal, classe (estudo de caso ATM), 367, 368, 370, 371, 375, 377, 383, 385, 386, 393, 394, 395, 396, 398
"World Wide Wait", 1005
World Wide Web (WWW), **5**, 862
  navegador, 74
writeBoolean, método da interface DataOutput, 577
writeByte, método da interface DataOutput, 577
writeBytes, método da interface DataOutput, 577
writeChar, método da interface DataOutput, 577
writeDouble, método de interface DataOutput, 577
writeFloat, método de interface DataOutput, 577
writeInt, método da interface DataOutput, 577
writeLong, método da interface DataOutput, 577
Writer, classe, **578**
writeShort, método da interface DataOutput, 577
writeUTF, método da interface DataOutput, 577
WSDL (Web Service Description Language), **1024**
WYSIWYG (What You See Is What You Get), editor, 959

X

X_AXIS, constante da classe Box, **792**
Xadrez, jogo, 896
X, eixo, **105**
XHTML (Extensible HyperText Markup Language), 947
  applet, elemento, **729**
  body, elemento, **729**
  span, elemento, **954**
  tag, **729**
  tutoriais (www.deitel.com/xhtml/), 723, 860
XHTML (Extensible HyperText Markup Language), documento, **723**, 729, 862
x, coordenada, **105**, **484**, 502, 728
XML (Extensible Markup Language), 735, 950, 1024
  declaração, 953
  elemento, **735**
  elemento-raiz, **735**
  tag de abertura, **735**
  tag de fechamento, **735**
  vocabulário, 735
XMLHttpRequest, objeto, **1005**
xmlns, atributos, **953**
xor, método da classe BitSet, **LI**

Y

Y_AXIS, constante da classe Box, **792**
Yahoo! Maps, XXVIII
Yahoo! Search, XXVIII
Y, eixo, **105**
y, coordenada, **105**, **484**, 502
YouTube, 17, XXVII

Z

zero, contagem baseada em, 122
zero-ésimo elemento, **190**
ZERO, constante da classe BigInteger, **595**

LEIA COM ATENÇÃO

Ao abrir o pacote do SOFTWARE, você concorda em vincular-se aos termos do contrato abaixo expresso. Portanto, não tome nenhuma ação antes da completa leitura e compreensão dos termos abaixo propostos.

Termos de licença de uso do Software

Pelo presente instrumento particular, de um lado a PEARSON EDUCATION DO BRASIL LTDA., doravante denominada CONTRATADA, e de outro lado o comprador/consumidor, doravante denominado CONTRATANTE, independentemente da assinatura escrita, mas condicionada à instalação do SOFTWARE, que valerá por aquela, anuem ao que segue:

1. Do objeto

1.1. Este termo tem por objeto o presente SOFTWARE, que inclui o programa de computador e poderá incluir meios físicos, materiais impressos e documentação on-line ou eletrônica, sem, no entanto, incluir seu conteúdo literário, que é de propriedade da CONTRATADA.

2. Dos direitos de uso

2.1. Ao instalar o SOFTWARE o CONTRATANTE está concordado em vincular-se aos termos e condições aqui expressos. Caso não esteja de acordo, o contratante deve interromper imediatamente a sequência de instalação, sendo-lhe facultada a devolução do produto à sede da CONTRATADA para obter reembolso do valor pago.

2.2. O CONTRATANTE possui somente a mídia magnética ou física (a mídia incluída) em que o SOFTWARE foi registrado ou corrigido, mas a CONTRATADA e os desenvolvedores de software retêm todos os direitos, títulos e posse do SOFTWARE registrado na(s) cópia(s) original(is) da mídia(s) e todas as cópias subsequentes do SOFTWARE, independentemente da forma ou mídia em que o original ou outras cópias possam existir. Esta licença não é uma venda do SOFTWARE original ou de quaisquer cópias do conteúdo do presente software, de propriedade da CONTRATADA. O conteúdo do cd está protegido pela Lei de Direitos Autorais, sendo proibida a reprodução total ou parcial, em qualquer forma ou por qualquer meio, estando o CONTRATANTE sujeito, no caso de infração, às penalidades previstas em lei. Assim, o contratante deverá tratar o SOFTWARE como qualquer outro material protegido por direito autoral.

2.3. É permitido ao CONTRATANTE:

2.3.1. Utilizar e exibir não exclusivamente a cópia do programa de software incluído em um único programa de computador (isto é, em uma única CPU) e em um único local, desde que o CONTRATANTE se submeta aos termos deste Acordo.

2.3.2. Fazer uma única cópia do SOFTWARE somente para propósitos de backup ou de arquivamento.

2.3.3. Transferir o SOFTWARE fisicamente de um computador para outro desde que o SOFTWARE seja utilizado somente em um computador por vez.

2.4. Não é permitido ao CONTRATANTE:

2.4.1. Copiar a Documentação ou o SOFTWARE. O CONTRATANTE pode ser legalmente responsável por cópia ou infração de direitos autorais causadas ou encorajadas por sua falha em obedecer aos termos dessa restrição.

2.4.2. Compartilhar o SOFTWARE em uma rede ou por outros meios ou utilizá-lo em mais de um computador ou terminal de computador ao mesmo tempo. Não pode distribuir cópias do SOFTWARE ou da Documentação a outros.

2.4.3. Fazer engenharia reversa, desassemblar, descompilar, modificar, adaptar, traduzir ou criar trabalhos derivados baseados no SOFTWARE ou na Documentação sem a permissão prévia por escrito da EMPRESA.

2.4.4. Transferir o SOFTWARE para outra pessoa sem a permissão prévia por escrito da EMPRESA.

2.4.5. Alugar ou arrendar o SOFTWARE, nem mesmo revendê-lo por qualquer valor.

2.4.6. O SOFTWARE é licenciado como um produto único. Seus componentes não poderão ser separados para utilização em mais de um computador.

3. Disposições gerais

3.1. A CONTRATADA poderá rescindir o presente termo caso o contratante não cumpra com seus termos e com suas condições, sem prejuízo das penalidades da lei.

3.2. O contratante, não aceitando as condições ora ofertadas e objetivando o pacto contratual, poderá não dar sequência à instalação do produto, interrompendo-a imediatamente, sendo-lhe facultado o pleito de devolução do valor pago mediante sua solicitação expressa, condicionada à devolução do produto na sede da contratada.

Requisitos mínimos de sistema

- Processador Pentium de 800 MHz (mínimo) III ou mais rápido.
- Windows Vista ou Windows XP (com Service Pack 2).
- 512 MB de RAM no mínimo; recomenda-se 1 GB.
- 1,5 GB de espaço de disco rígido no mínimo.
- Unidade de CD-ROM.
- Internet connection.
- Navegador da Web, Adobe® Reader® e um utilitário de descompactação de arquivos zip.

Java™ COMO PROGRAMAR

8ª edição

PAUL DEITEL HARVEY DEITEL

Você tem em mãos a mais consagrada obra de introdução à programação orientada a objetos que se baseia no DEITEL® live-code para ensinar tudo o que você precisa saber sobre Java™ Standard Edition 6, JDBC™ 4, desenvolvimento de aplicativos Web e Web Services!

O Java™ é atualmente a linguagem de programação mais popular do mundo. Esta nova edição, completamente atualizada, utiliza uma abordagem agradável e cuidadosamente compassada para que o estudante desenvolva aplicativos baseados em Web e desktop. Os novos exercícios "Fazendo a diferença" envolvem questões relacionadas à consciência social que são importantes tanto para estudantes como para profissionais.

"Um ótimo livro-texto com uma enorme variedade de exemplos de vários domínios de aplicações — excelente para um curso de ciência da computação." — **William E. Duncan, Louisiana State University**

"Esta nova edição atualizada reflete o estado da arte em tecnologias Java. Como sempre, suas explicações profundas e claras tornam o livro indispensável." — **José Antonio González Seco, Parlamento da Andaluzia**

"*Java — como programar* introduz noções de boas práticas de projetos e metodologias desde o começo. É um excelente ponto de partida para o desenvolvimento de aplicações Java robustas e de alta qualidade." — **Simon Ritter, Sun Microsystems**

"Tem um estilo fácil de ler, como se fosse um bate-papo. Traz exemplos de códigos claros que permitem ao leitor tornar-se proficiente em Java." — **Patty Kraft, San Diego State University**

"O melhor livro-texto introdutório que já vi. Gostaria que este livro tivesse sido publicado quando eu estava aprendendo a programar!" — **Lance Andersen, Sun Microsystems**

"As técnicas de projetos orientados a objetos estão presentes em todo o livro. Os exemplos são de fácil compreensão e trazem ótimas oportunidades de aprendizado! A explicação sobre herança é muito boa — o conceito é construído com base no exemplo, podendo ser facilmente compreendido. Fiquei muito contente de ver o desenho de elementos gráficos opcionais no início do livro. Os exercícios são desafiadores e divertirão os estudantes." — **Sue McFarland Metzger, Villanova University**

"Traz aos novos programadores o conhecimento proveniente de muitos anos de experiência no desenvolvimento de softwares." — **Edward F. Gehringer, North Carolina State University**

"Parabenizo os autores pela dedicação a pesquisas detalhadas e pelo desenvolvimento de exemplos tão ilustrativos. A visão geral passo a passo de projetos orientados a objetos é excelente." — **Ric Heishman, George Mason University**

"Acho os novos exercícios 'Fazendo a diferença' muito interessantes. Abordando temas contemporâneos, eles encorajam o estudante a pesquisar dados na Internet e aplicá-los no exercício em questão." — **Vince O'Brien, Pearson Education**

"Este livro oferece uma base sólida para programação em Java. A maioria dos principais conceitos é ilustrada por programas completos. A abordagem é compreensível e detalhada. Há uma grande variedade de exercícios para testar sua compreensão do assunto." — **Shyamal Mitra, University of Texas at Austin**

Esta oitava edição de *Java — como programar* apresenta uma abordagem abrangente sobre a programação orientada a objetos e traz vários estudos de caso que tratam de: classes (GradeBook, Time, Employee), sistema OOD/UML™ ATM opcional, GUI e desenho de elementos gráficos opcionais, aplicativo Web baseado em banco de dados operando em múltiplas camadas (livro de endereços) e serviços Web (Blackjack, sistema de reservas de passagens aéreas, gerador de equações etc.)

Paul J. Deitel e o dr. **Harvey M. Deitel** são os fundadores da **Deitel & Associates, Inc.**, uma organização internacionalmente reconhecida de treinamento corporativo, criação e desenvolvimento de negócios na Internet, especializada em Java™, C, C++, C#, Visual Basic®, Visual C++®, Python®, XML, Internet, Web e tecnologias de objeto. Os Deitel são autores de muitos livros-texto, livros profissionais e cursos em vídeo líderes de venda internacionalmente.

Visite os sites do livro (**www.pearson.com.br/deitel** e **www.deitel.com/books/jhtp8**). Contate os autores pelo e-mail deitel@deitel.com. Para receber o newsletter *Deitel® Buzz Online* e para informações sobre a série de treinamentos corporativos *Deitel® Dive Into® Series*, oferecida em todo o mundo, acesse o site **www.deitel.com/newsletter/subscribe.html**.

www.pearson.com.br/deitel

No site, professores têm acesso ao manual de soluções (em inglês) e a apresentações em PowerPoint, enquanto estudantes contam com exercícios de múltipla escolha e com os códigos (em inglês) dos exemplos apresentados no livro.

ISBN 978-85-7605-563-1

www.pearson.com.br