Universidade São Francisco

Prof. Peter Jandl Junior
Núcleo de Educação a Distância
Universidade São Francisco
1999

Jandl, Peter Jr.

Introdução ao Java, Apostila. p. cm.
Inclui índice e referências bibliográficas.

1. Java (linguagem de programação de computadores) 2. Programação para Internet.

1999.

Marcas Registradas

Índice Analítico

“Qual o absurdo de hoje que será
a verdade do amanhã?”
Alfred Whitehead

Prefácio

A linguagem Java e sua plataforma de programação constituem um fascinante objeto de estudo: ao mesmo tempo que seduzem pelas promessas de portabilidade, exibem um conjunto rico de bibliotecas que realmente facilitam o desenvolvimento de novas aplicações e utiliza como paradigma central a orientação à objetos.

Embora a mídia, grandes empresas e inúmeros especialistas vejam o Java como uma ferramenta indispensável para um futuro bastante próximo, tais previsões podem, obviamente, mudar em função de novas propostas e descobertas que hoje acontecem num ritmo quase inacreditável.

De qualquer forma o estudo Java é uma opção conveniente pois mesmo que o futuro desta linguagem não seja tão brilhante como parece, torna-se um meio eficiente para o aprendizado da orientação à objetos que é incontestavelmente importante.

Este material pretende apresentar de forma introdutória os conceitos essenciais da linguagem de programação Java e para tanto trata, sem profundidade, as questões fundamentais da orientação à objetos. Iniciando pelo ambiente de programação Java e a sua sintaxe, discute a construção de aplicações de console, aplicações gráficas e a construção de *applets*, procurando cobrir os elementos de maior uso da vasta biblioteca de programação oferecida por esta plataforma.

São quase uma centena de exemplos, vários diagramas de classe e ilustrações distribuídos ao longo deste texto que colocam de maneira prática como aplicar os conceitos discutidos.

Espera-se assim que o leitor encontre aqui um guia para um primeiro contato com a plataforma Java, percebendo as possibilidades de sua utilização e ganhando motivação para a continuidade de seus estudos.

O Autor.

"Mesmo um grande jornada se inicia com um primeiro passo."
Provérbio Chinês

1 A Linguagem Java

1.1 O que é o Java?

Java é uma nova linguagem de programação, introduzida no mercado em 1995 pela *Sun Microsystems*, que provocou e ainda provoca excitação e entusiasmo em programadores, analistas e projetistas de *software*.

Mas por que o Java produz esta reação? Simplesmente porque é o resultado de um trabalho consistente de pesquisa e desenvolvimento de mais do que uma simples linguagem de programação, mas de todo um ambiente de desenvolvimento e execução de programas que exibe as facilidades proporcionadas pela orientação à objetos, pela extrema portabilidade do código produzido, pelas características de segurança que esta plataforma oferece e finalmente pela facilidade de sua integração ao outros ambientes, destacando-se a Internet.

Figura 1 Java Logo

1.2 Pequeno Histórico

Tudo começou em 1991, com um pequeno grupo de projeto da *Sun Microsystems* denominado *Green* que pretendia criar uma nova geração de computadores portáteis inteligentes, capazes de se comunicar de muitas formas, ampliando suas potencialidades de uso. Para tanto decidiu-se criar também uma nova plataforma para o desenvolvimento destes equipamentos de forma que seu *software* pudesse ser portado para os mais diferentes tipos de equipamentos. A primeira escolha de um linguagem de programação para tal desenvolvimento foi C++, aproveitando suas características e a experiência do integrantes do grupo no desenvolvimento de produtos. Mas mesmo o C++ não permitia realizar com facilidade tudo aquilo que o grupo visionava.

James Gosling, coordenador do projeto, decidiu então pela criação de uma nova linguagem de programação que pudesse conter tudo aquilo que era considerado importante e que ainda assim fosse simples, portátil e fácil de programar. Surgiu assim a linguagem interpretada *Oak* (carvalho em inglês) batizada assim dada a existência de uma destas árvores em frente ao escritório de Gosling. Para dar suporte a linguagem também surgiu o *Green OS* e uma interface gráfica padronizada.

Após dois anos de trabalho o grupo finaliza o *Star7* (ou *7), um avançado PDA (*Personal Digital Assistant*) e em 1993 surge a primeira grande oportunidade de aplicação desta solução da *Sun* numa concorrência pública da *Time-Warner* para desenvolvimento de uma tecnologia para TV a cabo interativa, injustamente vencida pela SGI (*Silicon Graphics Inc.*).

O *Oak*, então rebatizado Java devido a problemas de *copyright*, continua sem uso definido até 1994, quando estimulados pelo grande crescimento da Internet, Jonathan Payne e Patrick Naughton desenvolveram o programa navegador *WebRunner*, capaz de efetuar o *download* e a execução de código Java via Internet. Apresentado formalmente pela *Sun* como o navegador *HotJava* e a linguagem Java no *SunWorld'95*, o interesse pela solução se mostrou explosivo. Poucos meses depois a *Netscape Corp.* lança uma nova versão de seu navegador *Navigator* também capaz de efetuar o *download* e a execução de pequenas aplicações Java então chamadas *applets*. Assim se inicia a história de sucesso do Java.

Numa iniciativa também inédita a *Sun* decide disponibilizar o Java gratuitamente para a comunidade de desenvolvimento de *software*, embora detenha todos os direitos relativos à linguagem e as ferramentas de sua autoria. Surge assim o *Java Developer's Kit* 1.0 (JDK 1.0). As plataformas inicialmente atendidas foram: *Sun Solaris* e *Microsoft Windows 95/NT*. Progressivamente foram disponibilizados *kits* para outras plataformas tais como *IBM OS/2, Linux* e *Applet Macintosh.*

Em 1997, surge o JDK 1.1 que incorpora grandes melhorias para o desenvolvimento de aplicações gráficas e distribuídas e no início de 1999 é lançado o JDK 1.2, contendo muitas outras melhorias, de forma que também seja conhecido como Java 2. Atualmente a *Sun* vem liberando novas versões ou correções a cada nove meses bem como novas API (*Application Program Interface*) para desenvolvimento de aplicações específicas. A última versão disponível é o JDK 1.3.

1.3 Características Importantes

A linguagem Java exibe importantes características que, em conjunto, diferenciam-na de outras linguagens de programação:

- Orientada à Objetos
  Java é uma linguagem puramente orientada à objetos pois, com exceção de seus tipos primitivos de dados, tudo em Java são classes ou instância de uma classe. Java atende todos os requisitos necessários para uma linguagem ser considerada orientada à objetos que resumidamente são oferecer mecanismos de abstração, encapsulamento e hereditariedade.
- Independente de Plataforma
  Java é uma linguagem independente de plataforma pois os programas Java são compilados para uma forma intermediária de código denominada *bytecodes* que utiliza instruções e tipos primitivos de tamanho fixo, ordenação *big-endian* e um biblioteca de classes padronizada. Os *bytecodes* são como uma linguagem de máquina destinada a uma única plataforma, a máquina virtual Java (JVM – *Java Virtual Machine*), um interpretador de *bytecodes*. Pode-se implementar uma JVM para qualquer plataforma assim temos que um mesmo programa Java pode ser executado em qualquer arquitetura que disponha de uma JVM.
- Sem Ponteiros
  Java não possui ponteiros, isto é, Java não permite a manipulação direta de endereços de memória nem exige que o objetos criados seja destruídos livrando os programadores de uma tarefa complexa. Além disso a JVM possui um mecanismo automático de gerenciamento de memória conhecido como *garbage collector*, que recupera a memória alocada para objetos não mais referenciados pelo programa.
- Performance
  Java foi projetada para ser compacta, independente de plataforma e para utilização em rede o que levou a decisão de ser interpretada através dos esquema de *bytecodes*. Como uma linguagem interpretada a performance é razoável, não podendo ser comparada a velocidade de execução de código nativo. Para superar esta limitação várias JVM dispõe de compiladores *just in time* (JIT) que compilam os *bytecodes* para código nativo durante a execução

otimizando a execução, que nestes casos melhora significativamente a performance de programas Java.

- Segurança
  Considerando a possibilidade de aplicações obtidas através de uma rede, a linguagem Java possui mecanismos de segurança que podem, no caso de *applets*, evitar qualquer operação no sistema de arquivos da máquina-alvo, minimizando problemas de segurança. Tal mecanismo é flexível o suficiente para determinar se uma *applet* é considerada segura especificando nesta situação diferentes níveis de acesso ao sistema-alvo.
- Permite Multithreading
  Java oferece recursos para o desenvolvimento de aplicações capazes de executar múltiplas rotinas concorrentemente bem dispõe de elementos para a sincronização destas várias rotinas. Cada um destes fluxos de execução é o que se denomina *thread*, um importante recurso de programação de aplicações mais sofisticadas.

Além disso, o Java é uma linguagem bastante robusta, oferece tipos inteiros e ponto flutuante compatíveis com as especificações IEEE, suporte para caracteres UNICODE, é extensível dinamicamente além de ser naturalmente voltada para o desenvolvimento de aplicações em rede ou aplicações distribuídas.

Tudo isto torna o Java uma linguagem de programação única.

1.4 Recursos Necessários

Para trabalharmos com o ambiente Java recomendamos o uso do *Java Developer's Kit* em versão superior à JDK 1.1.7 e um navegador compatível com o Java tais como o *Netscape Communicator* 4.5 ou o *Microsoft Internet Explorer* 4 ou versões superiores.

O JDK, em diversas versões e plataformas oficialmente suportadas pela *Sun*, pode ser obtido gratuitamente no *site*:

http://www.javasoft.com/products/jdk/

Versões de demonstração dos navegadores da *Microsoft*, *Netscape* e *Sun* (*HotJava*) podem ser obtidas nos seguintes *sites*:

http://www.microsoft.com/
http://www.netscape.com/
http://java.sun.com/products/hotjava/

Outras informações sobre a plataforma Java, artigos, dicas e exemplos podem ser obtidas nos *sites*:

http://www.javasoft.com/	Site oficial da *Sun* sobre o Java
http://www.javasoft.com/tutorial/	Tutoriais sobre o Java
http://java.sun.com/docs/books/tutorial/	Outros tutoriais sobre o Java
http://www.javaworld.com/	Revista *online*
http://www.javareport.com/	Revista *online*
http://www.jars.com/	*Applets*, exemplos e outros recursos
http://www.gamelan.com/	Diversos recursos e exemplos Java
http://www.internet.com/	Diversos recursos e exemplos Java
http://www.javalobby.org/	Revista *online*
http://www.sys-con.com/java	Revista *online*
http://sunsite.unc.edu/javafaq/javafaq.html	Respostas de dúvidas comuns sobre Java
http://www.acm.org/crossroads/	Revista *online* da ACM
http://www.december.com/works/java.html	Diversos recursos e exemplos Java

1.5 O Sun Java Developer's Kit

O JDK é composto basicamente por:

- Um compilador (javac)
- Uma máquina virtual Java (java)
- Um visualizador de *applets* (appletviewer)
- Bibliotecas de desenvolvimento (os *packages* java)
- Um programa para composição de documentação (javadoc)
- Um depurador básico de programas (jdb)
- Versão *run-time* do ambiente de execução (jre)

Deve ser observado que o JDK não é um ambiente visual de desenvolvimento, embora mesmo assim seja possível o desenvolvimento de aplicações gráficas complexas apenas com o uso do JDK que é, de fato, o padrão em termos da tecnologia Java.

Outros fabricantes de software tais como *Microsoft*, *Borland*, *Symantec* e IBM oferecem comercialmente ambientes visuais de desenvolvimento Java, respectivamente, *Visual J++*, *JBuilder*, *VisualCafe* e *VisualAge for Java*.

1.6 Um Primeiro Exemplo

Com o JDK adequadamente instalado num computador, apresentaremos um pequeno exemplo de aplicação para ilustrarmos a utilização básica das ferramentas do JDK. Observe o exemplo a seguir, sem necessidade de compreendermos em detalhe o que cada uma de suas partes representa:

```
//Eco.java

import java.io.*;

public class Eco {

  public static void main(String[] args) {
    for (int i=0; i < args.length; i++)
      System.out.print(args[i] + " ");
    System.out.println();
  }
}
```

Exemplo 1 Primeiro Exemplo de Aplicação Java

Utilize um editor de textos qualquer para digitar o exemplo, garantindo que o arquivo será salvo em formato de texto simples, sem qualquer formatação e que o nome do arquivo será Eco.java (utilize letras maiúsculas e minúsculas exatamente como indicado!).

Para podermos executar a aplicação exemplificada devemos primeiramente compilar o programa. Para tanto devemos acionar o compilador Java, ou seja, o programa javac, como indicado abaixo:

```
javac nome_do_arquivo.java
```

O compilador Java exige que os arquivos de programa tenham a extensão ".java" e que o arquivo esteja presente no diretório corrente, assim, numa linha de comando, dentro do diretório onde foi salvo o arquivo Eco.java digite:

```
javac Eco.java
```

Isto aciona o compilador Java transformando o código digitado no seu correspondente em *bytecodes*, produzindo um arquivo Eco.class. Não existindo erros o javac não exibe qualquer mensagem, caso contrário serão exibidas na tela uma mensagem para cada erro encontrado, indicando em que linha e que posição desta foi detectado o erro.

Tendo compilado o programa, ou seja, dispondo-se de um arquivo ".class" podemos invocar uma máquina virtual Java (JVM), isto é, o interpretador Java, da seguinte forma:

```
java nome_do_arquivo
```

Note que não é necessário especificarmos a extensão ".class" para acionarmos a JVM. Digitando-se o comando abaixo executamos o programa Eco:

```
java Eco
```

Aparentemente nada aconteceu, mas se a linha digitada fosse outra contendo argumentos diversos tal como:

```
java Eco Testando 1 2 3
```

Obteríamos o seguinte resultado:

```
Testando 1 2 3
```

Na verdade o programa exemplo apresentado apenas imprime todos os argumentos fornecidos como um eco. Embora de utilidade duvidosa, esta aplicação nos mostra como proceder a compilação e execução de programas escritos em Java utilizando o JDK.

1.7 O Ambiente Java

O exemplo dado na seção anterior, além de ilustrar aspectos relacionados a programação de aplicações Java, serve também para caracterizar uma parte do ambiente de desenvolvimento da linguagem Java: a criação de programas, sua compilação e execução, como mostra a Figura 2.

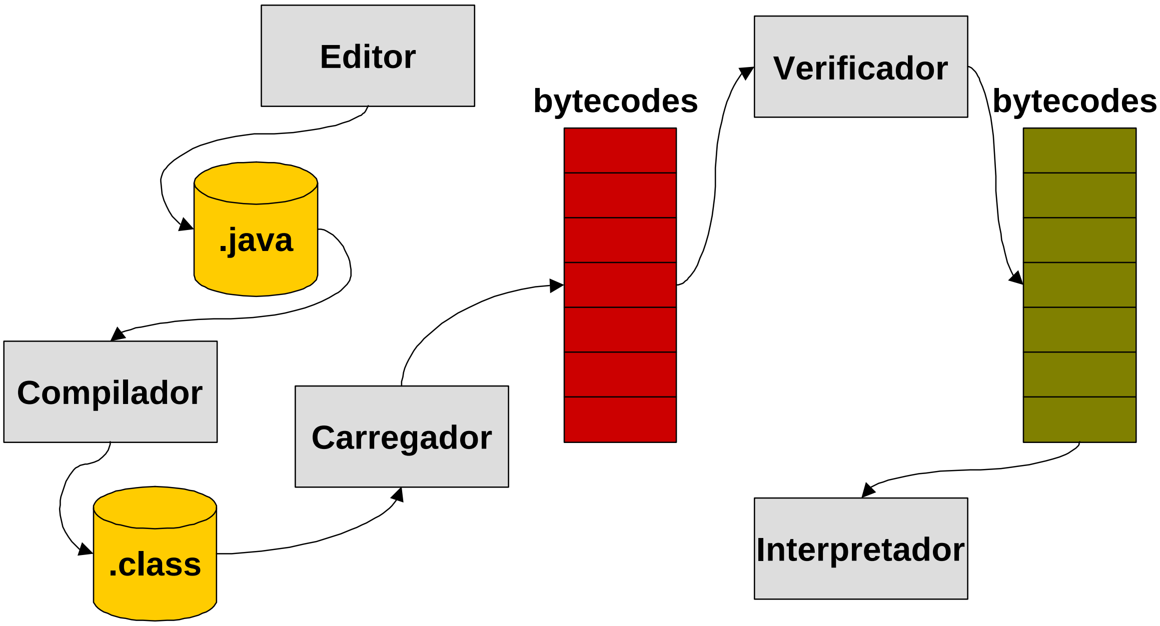


Figura 2 Ambiente de Desenvolvimento Java

Como visto, devemos utilizar um editor de textos ASCII simples para produzirmos o código fonte de um programa que, obrigatoriamente deve ser alvo como arquivos de extensão ".java". Utilizando o compilador javac o arquivo fonte é transformado em *bytecodes* Java que podem ser executados em qualquer plataforma computacional que disponha de uma máquina virtual Java como ilustrado na Figura 3.

Quando solicitamos a execução do programa, tanto através do interpretador java como de sua versão *run-time* jre, o arquivo ".class" é primeiramente carregado, verificado para garantir-se os requisitos de segurança do sistema e só então é propriamente interpretado. No caso específico das *applets* o navegador (*browser*) efetua o download de um arquivo ".class" interpretando-o ou acionando algum interpretador associado (*plug-in*). Em ambientes onde existam compiladores JIT (*just in time*) os *bytecodes* já verificados são convertidos em instruções nativas do ambiente durante a execução, aumentando a performance das aplicações ou *applets* Java.

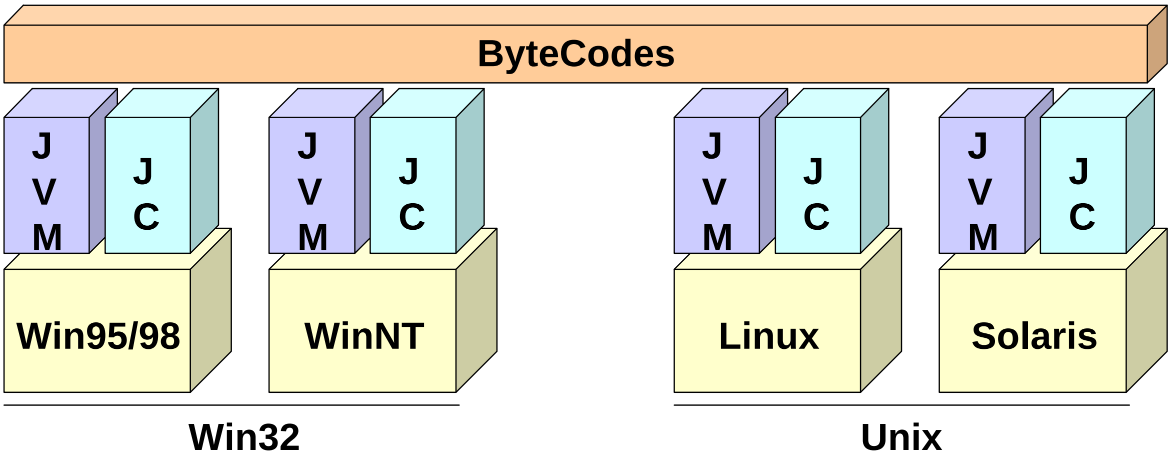


Figura 3 Ambiente Java e os *Bytecodes*

O resultado deste ambiente é que o Java, embora interpretado inicialmente, torna-se independente de plataforma, simplificando o projeto de aplicações de rede ou distribuídas que tenham que operar em ambientes heterogêneos, além de permitir a incorporação de vários mecanismos de segurança.

2 Java: Variáveis, Operadores e Estruturas de Controle

Apresentaremos agora uma pequena discussão sobre a sintaxe da linguagem Java, abordando os tipos de dados existentes, as regras para declaração de variáveis, as recomendações gerais para nomenclatura, os operadores, sua precedência e as estruturas de controle disponíveis. Como será notado, a sintaxe da linguagem Java é muito semelhante àquela usada pela linguagem C/C++.

2.1 Tipos de Dados Primitivos

A linguagem Java possui oito tipos básicos de dados, denominados tipos primitivos, que podem agrupados em quatro categorias:

Tipos Inteiros	Tipos Ponto Flutuante	Tipo Caractere
Byte	Ponto Flutuante Simples	Caractere
Inteiro Curto	Ponto Flutuante Duplo	
Inteiro		**Tipo Lógico**
Inteiro Longo		Boleano

Tabela 1 Tipos de Dados Primitivos

Como pode ser facilmente observado, os tipos primitivos do Java são os mesmos encontrados na maioria das linguagens de programação e permitem a representação adequada de valores numéricos. A representação de outros tipos de dados utiliza objetos específicos assim como existem classes denominadas *wrappers* que encapsulam os tipos primitivos como objetos da linguagem.

2.1.1 Tipos de Dados Inteiros

Existem quatro diferentes tipos de dados inteiros *byte* (8 bits), *short* (inteiro curto – 16 bits), *int* (inteiro – 32 bits) e *long* (inteiro longo – 64 bits) cuja representação interna é feita através de complemento de 2 e que podem armazenar valores dentro dos seguintes intervalos numéricos:

Tipo	Valor Mínimo	Valor Máximo
byte	-128	+127
short	-32.768	+32.767
int	-2.147.483.648	+2.147.483.647
long	-9.223.372.036.854.775.808	+9.223.372.036.854.775.807

Tabela 2 Tipos de Dados Inteiros

Por *default* os valores literais são tratados como inteiros simples (*int*) ou seja valores de 32 bits. Não existe em Java o modificador *unsigned* disponível em outras linguagens assim os tipos inteiros são sempre capazes de representar tanto valores positivos como negativos.

2.1.2 Tipo de Dados em Ponto Flutuante

No Java existem duas representações para números em ponto flutuante que se diferenciam pela precisão oferecida: o tipo *float* permite representar valores reais com precisão simples (representação interna de 32 bits) enquanto o tipo *double* oferece dupla precisão (representação interna de 64 bits). Os valores m ponto flutuante do Java estão em conformidade com o padrão IEEE 754:

Tipo	Valor Mínimo	Valor Máximo
float		
double		

Tabela 3 Tipos de Dados em Ponto Flutuante

Deve ser utilizado o ponto como separador de casas decimais. Quando necessário, expoentes podem ser escritos usando o caractere ‘e’ ou ‘E’, como nos seguintes valores: 1.44E6 (= 1.44 x 10^6 = 1,440,000) ou 3.4254e-2 (= 3.4254 x 10^{-2} = 0.034254).

2.1.3 Tipo de Dados Caractere

O tipo *char* permite a representação de caracteres individuais. Como o Java utiliza uma representação interna no padrão UNICODE, cada caractere ocupa 16 bits (2 bytes) sem sinal, o que permite representar até 32.768 caracteres diferentes, teoricamente facilitando o trabalho de internacionalização de aplicações Java. Na prática o suporte oferecido ao UNICODE ainda é bastante limitado embora permita a internacionalização do código Java.

Alguns caracteres são considerados especiais pois não possuem uma representação visual, sendo a maioria caracteres de controle e outros caracteres cujo uso é reservado pela linguagem. Tais caracteres podem ser especificados dentro dos programas como indicado na tabela abaixo, ou seja, precedidos por uma barra invertida (‘\’):

Representação	Significado
\n	Pula linha (*newline* ou *linefeed*)
\r	Retorno de carro (*carriage return*)
\b	Retrocesso (*backspace*)
\t	Tabulação (*horizontal tabulation*)
\f	Nova página (*formfeed*)
\’	Apóstrofe
\”	Aspas
\\	Barra invertida
\u223d	Caractere UNICODE 233d
\g37	Octal
\fca	Hexadecimal

Tabela 4 Representação de Caracteres Especiais

O valor literal de caracteres deve estar delimitado por aspas simples (‘ ’).

2.1.4 Tipo de Dados Lógico

Em Java dispõe-se do tipo lógico *boolean* capaz de assumir os valores *false* (falso) ou *true* (verdadeiro) que equivalem aos estados *off* (desligado) e *on* (ligado) ou *no* (não) e *yes* (sim).

Deve ser destacado que não existem equivalência entre os valores do tipo lógico e valores inteiros tal como usualmente definido na linguagem C/C++.

2.2 Declaração de Variáveis

Uma variável é um nome definido pelo programador ao qual pode ser associado um valor pertencente a um certo tipo de dados. Em outras palavras, uma variável é como uma memória, capaz de armazenar um valor de um certo tipo, para a qual se dá um nome que usualmente descreve seu significado ou propósito. Desta forma toda variável possui um nome, um tipo e um conteúdo.

O nome de uma variável em Java pode ser uma sequência de um ou mais caracteres alfabéticos e numéricos, iniciados por uma letra ou ainda pelos caracteres '_' (*underscore*) ou '$' (cifrão). Os nomes não podem conter outros símbolos gráficos, operadores ou espaços em branco, podendo ser arbitrariamente longos embora apenas os primeiros 32 caracteres serão utilizados para distinguir nomes de diferentes variáveis. É importante ressaltar que as letras minúsculas são consideradas diferentes das letras maiúsculas, ou seja, a linguagem Java é sensível ao caixa empregado, assim temos como exemplos válidos:

a
_especial
total
TOT
x2
Maximo
$mine
ExpData

Segundo as mesmas regras temos abaixo exemplos inválidos de nomes de variáveis:

1x
Total geral
numero-minimo
void

A razão destes nomes serem inválidos é simples: o primeiro começa com um algarismo numérico, o segundo possui um espaço em branco, o terceiro contêm o operador menos mas por que o quarto nome é inválido?

Porque além das regras de formação do nome em si, uma variável não pode utilizar como nome uma palavra reservada da linguagem. As palavras reservadas são os comandos, nomes dos tipos primitivos, especificadores e modificadores pertencentes a sintaxe de uma linguagem.

As palavras reservadas da linguagem Java, que portanto não podem ser utilizadas como nome de variáveis ou outros elementos, são:

abstract
boolean
break
byte
case
catch
char
class
continue
default
do
double
else
extends
false
final
finally
float
for
if
implements
import
instanceof
int
interface
long
native
new
null
package
private
protected
public
return
short
static
super
switch
synchronized
this
throw
throws
transient
true
try
void
while

Além destas existem outras que embora reservadas não são utilizadas pela linguagem:

const
outer
future
rest
generic
var
goto
volatile
inner
operator

Algumas destas, tal como o goto, faziam parte da especificação preliminar do *Oak*, antes de sua formalização como Java. Recomenda-se não utilizá-las qualquer que seja o propósito.

Desta forma para declararmos uma variável devemos seguir a seguinte sintaxe:

```
Tipo nome1 [, nome2 [, nome3 [..., nomeN]]];
```

Ou seja, primeiro indicamos um tipo, depois declaramos uma lista contendo um ou mais nomes de variáveis desejadas deste tipo, onde nesta lista os nomes são separados por vírgulas e a declaração terminada por ';' (ponto e vírgula). Exemplos:

```
int i;
float total, preco;
byte mascara;
double valorMedio;
```

As variáveis podem ser declaradas individualmente ou em conjunto:

```
char opcao1, opcao2;
```

A declaração anterior equivale as duas declarações abaixo:

```
char opcao1;
char opcao2;
```

Também é possível definirmos uma valor inicial para uma variável diretamente em sua declaração como indicado a seguir:

```
int quantidade = 0;
float angulo = 1.57;
boolean ok = false;
char letra = ‘c’;
```

Variáveis podem ser declaradas em qualquer ponto de um programa Java, sendo válidas em todos no escopo onde foram declaradas e nos escopos internos à estes. Por escopo entende-se o bloco (conjunto de comandos da linguagem) onde ocorreu a declaração da variável.

2.2.1 Regras para Denominação de Variáveis

Em Java recomenda-se que a declaração de variáveis utilize nomes iniciados com letras minúsculas. Caso o nome seja composto de mais de uma palavras, as demais deveriam ser iniciadas com letras maiúsculas tal como nos exemplos:

contador
posicaoAbsoluta
total
valorMinimoDesejado
sinal
mediaGrupoTarefa2

A utilização de caracteres numéricos no nome é livre enquanto o uso do traço de sublinhar (*underscore* ‘_’) não é recomendado.

2.3 Comentários

Comentários são trechos de texto, usualmente explicativos, inseridos dentro do programa de forma que não sejam considerados como parte do código, ou seja, são informações deixadas juntamente com o código para informação de quem programa.

O Java aceita três tipos de comentários: de uma linha, de múltiplas linhas e de documentação. O primeiro de uma linha utiliza duas barras (`//`) para marcar seu início:

```
// comentário de uma linha
// tudo após as duas barras é considerado comentário
```

O segundo usa a combinação `/*` e `*/` para delimitar uma ou mais linhas de comentários:

```
/* comentário
   de múltiplas linhas */
```

O último tipo é semelhante ao comentário de múltiplas linhas mas tem o propósito de documentar o programa:

```
/** comentário de documentação que também
  * podem ter múltiplas linhas
  */
```

Geralmente o comentário de documentação é posicionado imediatamente antes do elemento a ser documentado e tem seu conteúdo extraído automaticamente pelo utilitário javadoc fornecido juntamente com o JDK. Esta ferramenta gera páginas em formato html contendo os comentário organizados da mesma forma que a documentação fornecida juntamente com o JDK.

Aproveitando tal característica do javadoc, usualmente se adicionam *tags* html aos comentários de documentação para melhorar a forma final da documentação produzida com a inclusão de imagens, tabelas, textos explicativos, *links* e outros recursos. Além das *tags* html vários tipos de informação administradas pelo javadoc podem ser adicionadas a estes comentários especiais através de marcadores pré-definidos iniciados com “@” que permitem a criação automática de ligações hipertexto entre a documentação e a formatação padronizada de outros elementos, tais como nome do autor, parâmetros, tipo de retorno, etc. como abaixo:

```
/** Classe destinada ao armazenamento de dados relacionados a
  * arquivos ou diretórios.
  * <p> Pode ser usada para armazenar árvores de diretórios.
  * @author Peter Jandl Jr
  * @see java.io.File
  */
public class FileData extends File {
  /** Construtor
    * @param filename nome do arquivo
    */
  public FileData(String filename) {
  }
}
```

2.4 Operadores

A linguagem Java oferece um conjunto bastante amplo de operadores destinados a realização de operações aritméticas, lógicas, relacionais e de atribuição.

2.4.1 Operadores Aritméticos

Como na maioria das linguagens de programação, o Java possui vários operadores aritméticos:

Operador	Significado	Exemplo
+	Adição	a + b
-	Subtração	a – b
*	Multiplicação	a * b
/	Divisão	a / b
%	Resto da divisão inteira	a % b
-	Sinal negativo (- unário)	-a
+	Sinal positivo (+ unário)	+a
++	Incremento unitário	++a ou a++
--	Decremento unitário	--a ou a--

Tabela 5 Operadores Aritméticos

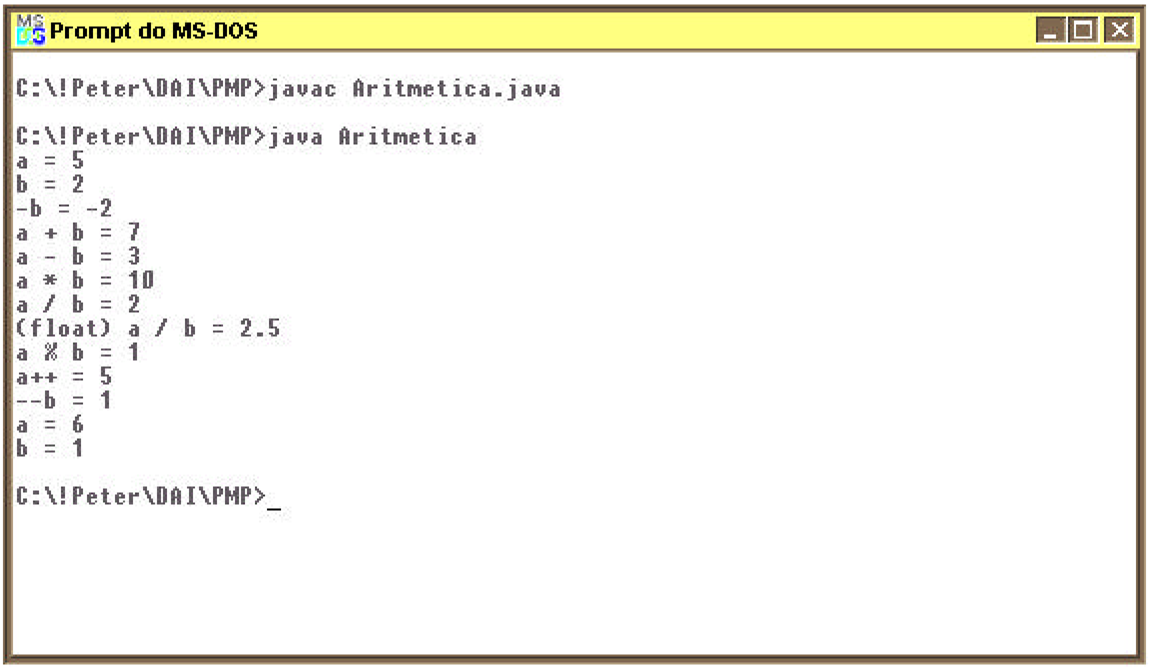
Estes operadores aritméticos podem ser combinados para formar expressões onde deve ser observada a precedência (ordem convencional) de avaliação dos operadores. Parêntesis podem ser utilizados para determinar uma forma específica de avaliação de uma expressão. A seguir um exemplo de aplicação que declara algumas variáveis, atribui valores iniciais e efetua algumas operações imprimindo os resultados obtidos.

```
// Aritmetica.java

public class Aritmetica {

  static public void main (String args[]) {
    // Declaracao e inicializacao de duas variaveis
    int a = 5;
    int b = 2;

    // Varios exemplos de operacoes sobre variaveis
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("-b = " + (-b));
    System.out.println("a + b = " + (a + b));
    System.out.println("a - b = " + (a - b));
    System.out.println("a * b = " + (a * b));
    System.out.println("a / b = " + (a / b));
    System.out.println("(float) a / b = " + ((float)a / b));
```

```
  System.out.println("a % b = " + (a % b));
  System.out.println("a++ = " + (a++));
  System.out.println("--b = " + (--b));
  System.out.println("a = " + a);
  System.out.println("b = " + b);
 }
}
```

Exemplo 2 Aplicação Aritmetica (operadores aritméticos)

Compilando e executando o código fornecido teríamos o resultado ilustrado a seguir:



```
C:\!Peter\DAI\PMP>javac Aritmetica.java

C:\!Peter\DAI\PMP>java Aritmetica
a = 5
b = 2
-b = -2
a + b = 7
a - b = 3
a * b = 10
a / b = 2
(float) a / b = 2.5
a % b = 1
a++ = 5
--b = 1
a = 6
b = 1

C:\!Peter\DAI\PMP>_
```

Figura 4 Resultado da Aplicação Aritmetica

Embora o exemplo só tenha utilizado variáveis e valores inteiros, o mesmo pode ser realizado com variáveis do tipo ponto flutuante (*float* ou *double*)

2.4.2 Operadores Relacionais

Além dos operadores aritméticos o Java possui operadores relacionais, isto é, operadores que permitem comparar valores literais, variáveis ou o resultado de expressões retornando um resultado do tipo lógico, isto é, um resultado falso ou verdadeiro. Os operadores relacionais disponíveis são:

Operador	Significado	Exemplo
==	Igual	a == b
!=	Diferente	a != b
>	Maior que	a > b
>=	Maior ou igual a	a >= b
<	Menor que	a < b
<=	Menor ou igual a	a >= b

Tabela 6 Operadores Relacionais

Note que o operador igualdade é definido como sendo um duplo sinal de igual (==) que não deve ser confundido com o operador de atribuição, um sinal simples de igual (=). Como é possível realizar-se uma atribuição como parte de uma expressão condicional, o compilador não faz nenhuma menção sobre seu uso, permitindo que uma atribuição seja escrita no lugar de uma comparação por igualdade, constituindo um erro comum mesmo para programadores mais experientes.

O operador de desigualdade é semelhante ao existente na linguagem C, ou seja, é representado por “!=”. Os demais são idênticos a grande maioria das linguagens de programação em uso.

É importante ressaltar também que os operadores relacionais duplos, isto é, aqueles definidos através de dois caracteres, não podem conter espaços em branco.

A seguir um outro exemplo simples de aplicação envolvendo os operadores relacionais. Como para os exemplos anteriores, sugere-se que esta aplicação seja testada como forma de se observar seu comportamento e os resultados obtidos.

```
// Relacional.java

import java.io.*;

public class Relacional {

  static public void main (String args[]) {
    int a = 15;
    int b = 12;

    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("a == b -> " + (a == b));
    System.out.println("a != b -> " + (a != b));
    System.out.println("a  < b -> " + (a < b));
    System.out.println("a  > b -> " + (a > b));
    System.out.println("a <= b -> " + (a <= b));
    System.out.println("a >= b -> " + (a >= b));
  }
}
```

Exemplo 3 Aplicação Relacional (operadores relacionais)

2.4.3 Operadores Lógicos

Como seria esperado o Java também possui operadores lógicos, isto é, operadores que permitem conectar logicamente o resultado de diferentes expressões aritméticas ou relacionais construindo assim uma expressão resultante composta de várias partes e portanto mais complexa.

Operador	Significado	Exemplo
&&	E lógico (*and*)	a && b
||	Ou Lógico (*or*)	a || b
!	Negação (*not*)	!a

Tabela 7 Operadores Lógicos

Os operadores lógicos duplos, isto é, definidos por dois caracteres, também não podem conter espaços em branco.

2.4.4 Operador de Atribuição

Atribuição é a operação que permite definir o valor de uma variável através de uma constante ou através do resultado de uma expressão envolvendo operações diversas.

```
boolean result = false;
i = 0;
y = a*x + b;
```

Em Java é válido o encadeamento de atribuições, tal como abaixo, onde todas as variáveis são inicializadas com o mesmo valor:

```
byte m, n, p, q;
⋮
m = n = p = q = 0;  // equivale a m = (n = (p = (q = 0)));
```

2.4.5 Precedência de Avaliação de Operadores

Numa expressão onde existam diversos operadores é necessário um critério para determinar-se qual destes operadores será primeiramente processado, tal como ocorre em expressões matemáticas comuns. A precedência é este critério que especifica a ordem de avaliação dos operadores de um expressão qualquer. Na Tabela 8 temos relacionados os níveis de precedência organizados do maior (Nível 1) para o menor (Nível 15). Alguns dos operadores colocados na tabela não estão descritos neste texto.

Nível	Operadores
1	**.** (seletor) [ ] ( )
2	++ -- ~ *instanceof* *new* *clone* - (unário)
3	* / %
4	+ -
5	<< >> >>>
6	< > <= >=
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= *op*=
15	,

Tabela 8 Precedência dos Operadores em Java

Note que algumas palavras reservadas (*instanceof*, *new* e *clone*) se comportam como operadores. Operadores de um mesmo nível de precedência são avaliados conforme a ordem em que são encontrados, usualmente da esquerda para direita. Para modificarmos a ordem natural de avaliação dos operadores é necessário utilizarmos os parêntesis, cujo nível de avaliação é o mais alto, para especificar-se a ordem de avaliação desejada.

2.5 Estruturas de Controle

Um programa de computador é uma sequência de instruções organizadas de forma tal a produzir a solução de um determinado problema. Naturalmente tais instruções são executadas em sequência, o que se denomina fluxo sequencial de execução. Em inúmeras circunstâncias é necessário executar as instruções de um programa em uma ordem diferente da estritamente sequencial. Tais situações são caracterizadas pela necessidade da repetição de instruções individuais ou de grupos de instruções e também pelo desvio do fluxo de execução.

As linguagens de programação tipicamente possuem diversas estruturas de programação destinadas ao controle do fluxo de execução, isto é, estruturas que permitem a repetição e o desvio do fluxo de execução. Geralmente as estruturas de controle de execução são divididas em:

- Estruturas de repetição simples
  Destinadas a repetição de um ou mais comandos, criando o que se denomina laços. Geralmente o número de repetições é pré-definido ou pode ser

determinado pelo programa durante a execução. No Java dispõe-se da diretiva *for*.

- Estruturas de desvio de fluxo
  Destinadas a desviar a execução do programa para uma outra parte, quebrando o fluxo sequencial de execução. O desvio do fluxo pode ocorrer condicionalmente, quando associado a avaliação de uma expressão, ou incondicionalmente. No Java dispõe-se das diretivas *if* e *switch*.
- Estruturas de repetição condicionais
  Semelhantes as estruturas de repetição simples mas cuja repetição está associada a avaliação de uma condição sendo geralmente utilizadas quando não se conhece de antemão o número necessário de repetições. No Java dispõe-se das diretivas *while* e *do while*.

Além destas estruturas existem ainda:

- Mecanismos de modularização
- Estruturas de controle de erros

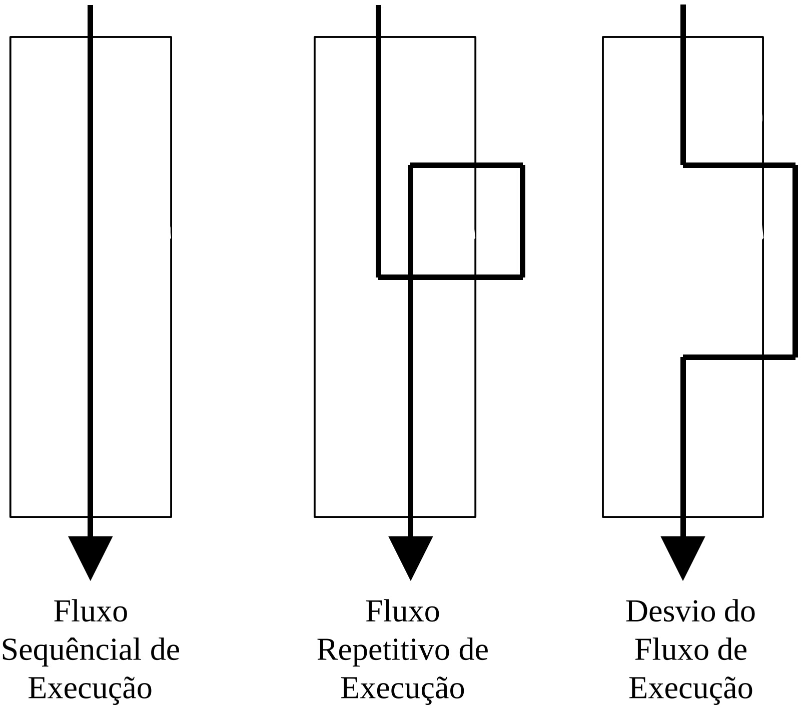


Figura 5 Variações do Fluxo de Execução de um Programa

Os mecanismos de modularização são aqueles que nos permitem a construção de funções e procedimentos (dentro do paradigma procedural) ou métodos (dentro da paradigma da orientação à objetos) e serão discutidos na próxima seção. Já as estruturas de controle de erros constituem uma importante contribuição a programação pois simplifica bastante a a inclusão e construção de rotinas de tratamento de erros dentro do código.

Antes de tratarmos especificamente das estruturas de controle da linguagem é necessário colocarmos algumas definições. Formalmente as instruções de um programa são chamadas diretivas (*statements*). Tradicionalmente as diretivas são escritas uma após o outra num programa e são separadas de alguma forma, por exemplo com uma quebra de linha ou um caractere de pontuação.

Em Java, tal como na linguagem C/C++, as diretivas são separadas uma das outras através do símbolo de pontuação ';' (ponto e vírgula), sendo possível existir várias diretivas numa mesma linha desde que separadas por um ponto e vírgula.

Abaixo temos um exemplo hipotético de várias diretivas colocadas sequenciamente:

```
diretiva1;
diretiva2;
diretiva3;
⋮
diretivaN;
```

Como nas outras linguagens de programação, as estruturas de controle podem operar sobre diretivas isoladas (individuais) ou sobre várias diretivas tratadas como um conjunto que é denominado bloco. Um bloco em Java é um grupo de diretivas delimitadas por chaves ({…}). Por sua vez um bloco de diretivas recebe um tratamento equivalente ao de uma única diretiva individual.

```
{
  diretiva1;
  diretiva2;
  diretiva3;
      ⋮
  diretivaN;
}
```

```
diretivaUnica;
```

2.5.1 Estruturas de repetição simples

Como repetição simples consideramos um trecho de código, isto é, um conjunto de diretivas que deve ser repetido um número conhecido e fixo de vezes. A repetição é uma das tarefas mais comuns da programação utilizada para efetuarmos contagens, para obtenção de dados, para impressão etc. Em Java dispomos da diretiva *for* cuja sintaxe é dada a seguir:

```
for (inicialização; condição de execução; incremento/decremento)
  diretiva;
```

O *for* possui três campos ou seções, todas opcionais, delimitados por um par de parêntesis que efetuam o controle de repetição de uma diretiva individual ou de um bloco de diretivas. Cada campo é separado do outro por um ponto e vírgula. O primeiro campo é usado para dar valor inicial a uma variável de controle (um contador). O segundo campo é uma expressão lógica que determina a execução da diretiva associada ao *for*, geralmente utilizando a variável de controle e outros valores.

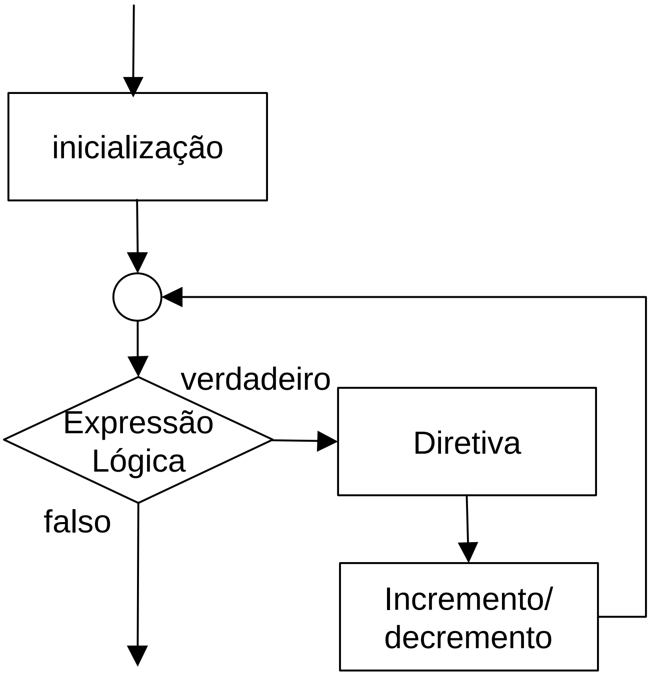


Figura 6 Comportamento da Diretiva *for*

Após a execução da seção de inicialização ocorre a avaliação da expressão lógica. Se a expressão é avaliada como verdadeira, a diretiva associada é executada, caso contrário o comando for é encerrado e a execução do programa prossegue com o próximo comando após o *for*. O terceiro campo determina como a variável de controle será modificada a cada iteração do *for*. Considera-se como iteração a execução completa da diretiva associada, fazendo que ocorra o incremento ou decremento da variável de controle. A seguir um exemplo de utilização da diretiva *for*:

```
import java.io.*;

public class exemploFor {
  public static void main (String args[]) {
    int j;
    for (j=0; j<10; j++) {
      System.out.println(“”+j);
    }
  }
}
```

Exemplo 4 Utilização da Diretiva *for*

Se executado, o exemplo acima deverá exibir uma contagem de 0 até 9 onde cada valor é exibido numa linha do console.

2.5.2 Estruturas de desvio de fluxo

Existem várias estruturas de desvio de fluxo que podem provocar a modificação da maneira com que as diretivas de um programa são executadas conforme a avaliação de uma condição. O Java dispõe de duas destas estruturas: *if* e *switch*.

O *if* é uma estrutura simples de desvio de fluxo de execução, isto é, é uma diretiva que permite a seleção entre dois caminhos distintos para execução dependendo do resultado falso ou verdadeiro resultante de uma expressão lógica.

```
if (expressão_lógica)
  diretiva1;
else
  diretiva2;
```

A diretiva *if* permite duas construções possíveis: a primeira, utilizando a parte obrigatória, condiciona a execução da diretiva1 a um resultado verdadeiro oriundo da avaliação da expressão lógica associada; a segunda, usando opcionalmente o *else*, permite que seja executada a diretiva1 caso o resultado da expressão seja verdadeiro ou que seja executada a diretiva2 caso tal resultado seja falso. Isto caracteriza o comportamento ilustrado pela Figura 7.

A seguir um exemplo de uso da diretiva *if*.

```
import java.io.*;

public class exemploIf {

  public static void main (String args[]) {
    if (args.length > 0) {
      for (int j=0; j<Integer.parseInt(args[0]); j++) {
        System.out.print("" + j + " ");
      }
      System.out.println("\nFim da Contagem");
    }
    System.out.println("Fim do Programa");
  }
}
```

Exemplo 5 Utilização da Diretiva *if*

Ao executar-se ente programa podem ocorrer duas situações distintas como resultado: se foi fornecido algum argumento na linha de comando (args.length > 0), o programa o converte para um número inteiro (Integer.parseInt(args[0])) e exibe uma contagem de 0 até o número fornecido e depois é exibida uma mensagem final, senão apenas a mensagem final é exibida. Caso se forneça um argumento que não seja um valor inteiro válido ocorre um erro que é sinalizado como uma exceção **java.lang.NumberFormatException** que interrompe a execução do programa.

Já o *switch* é uma diretiva de desvio múltiplo de fluxo, isto é, baseado na avaliação de uma expressão ordinal é escolhido um caminho de execução dentre vários possíveis. Um resultado ordinal é aquele pertencente a um conjunto onde se conhecem precisamente o elemento anterior e o posterior, por exemplo o conjunto dos números inteiros ou dos caracteres, ou seja, um valor dentro de um conjunto cujos valores podem ser claramente ordenados (0, 1, 2 .... no caso dos inteiros e ‘A’, ‘B’, ‘C’... no caso dos caracteres).

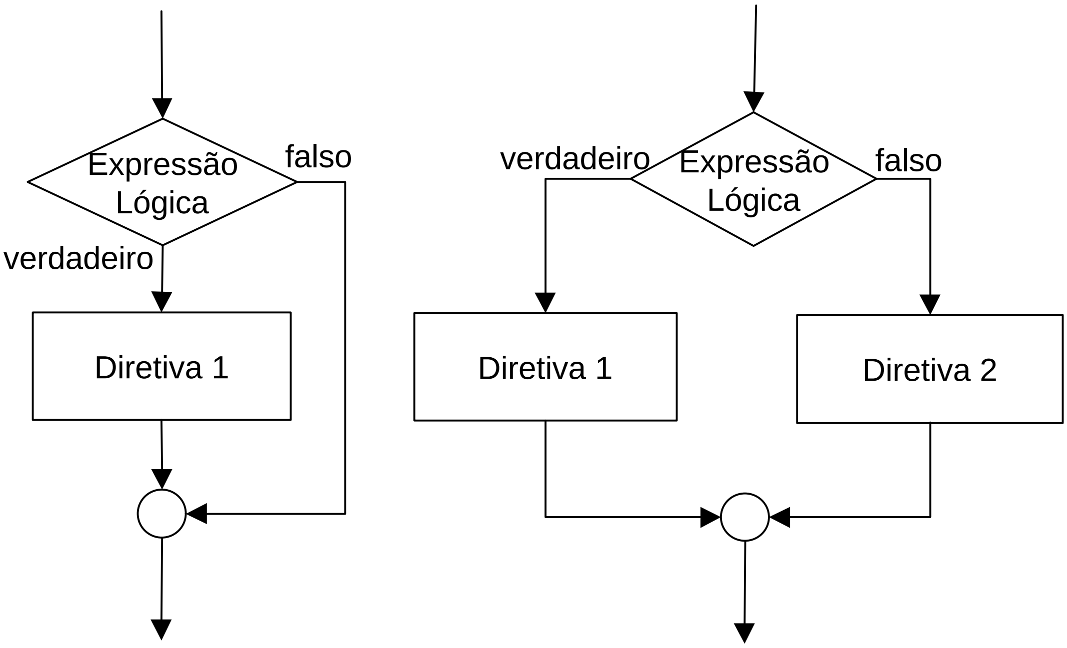


Figura 7 Comportamento da Diretiva *if*

O *switch* equivale logicamente a um conjunto de diretivas *if* encadeadas, embora seja usualmente mais eficiente durante a execução. Na Figura 8 temos ilustrado o comportamento da diretiva *switch*.

A sintaxe desta diretiva é a seguinte:

```
switch (expressão_ordinal) {
  case ordinal1: diretiva3;
                 break;
  case ordinal2: diretiva2;
                 break;
  default: diretiva_default;
}
```

A expressão utilizada pelo *switch* deve necessariamente retornar um resultado ordinal. Conforme o resultado é selecionado um dos casos indicados pela construção case ordinal. As diretivas encontradas a partir do caso escolhido são executadas até o final da diretiva *switch* ou até uma diretiva *break* que encerra o *switch*. Se o valor resultante não possuir um caso específico são executadas as diretivas *default* colocadas, opcionalmente, ao final da diretiva *switch*.

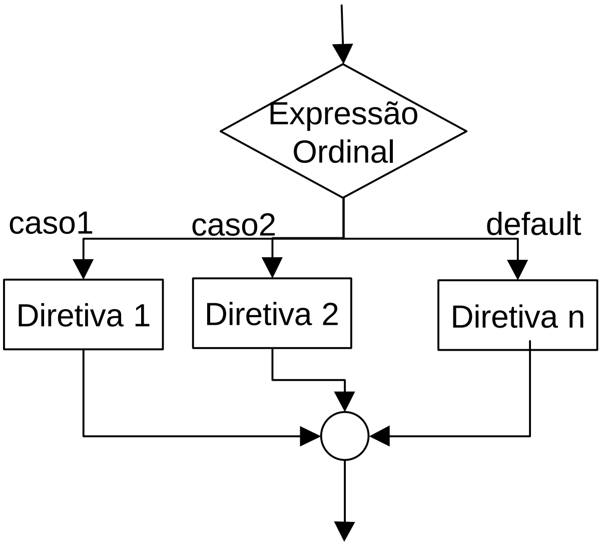


Figura 8 Comportamento da Diretiva *switch*

Note que o ponto de início de execução é um caso (*case*) cujo valor ordinal é aquele resultante da expressão avaliada. Após iniciada a execução do conjunto de diretivas identificadas por um certo caso, tais ações só são interrompidas com a execução de uma diretiva *break* ou com o final da diretiva *switch.*

A seguir temos uma aplicação simples que exemplifica a utilização das diretivas *switch* e *break*.

```
// exemploSwitch.java

import java.io.*;

public class exemploSwitch {

  public static void main (String args[]) {
    if (args.length > 0) {
      switch(args[0].charAt(0)) {
        case 'a':
        case 'A': System.out.println("Vogal A");
                  break;
        case 'e':
        case 'E': System.out.println("Vogal E");
                  break;
        case 'i':
        case 'I': System.out.println("Vogal I");
                  break;
        case 'o':
        case 'O': System.out.println("Vogal O");
                  break;
        case 'u':
        case 'U': System.out.println("Vogal U");
                  break;
        default:  System.out.println("Não é uma vogal");
      }
    } else {
      System.out.println("Não foi fornecido argumento");
    }
  }
}
```

Exemplo 6 Utilização da Diretiva *Switch*

No exemplo, quando fornecido algum argumento, a aplicação identifica se o primeiro caractere deste é uma vogal. Se o argumento não for iniciado por vogal ocorre a execução da seção *default* da diretiva *switch*. Caso não existam argumentos fornecidos o programa imprime uma mensagem correspondente.

2.5.3 Estruturas de repetição condicionais

As estruturas de repetição condicionais são estruturas de repetição cujo controle de execução é feito pela avaliação de expressões condicionais. Estas estruturas são adequadas para permitir a execução repetida de um conjunto de diretivas por um número indeterminado de vezes, isto é, um número que não é conhecido durante a fase de programação mas que pode ser determinado durante a execução do programa tal como um valor a ser fornecido pelo usuário, obtido de um arquivo ou ainda de cálculos realizados com dados alimentados pelo usuário ou lido de arquivos. Existem duas estruturas de repetição condicionais: *while* e *do while*.

O *while* é o que chamamos de laço condicional, isto é, um conjunto de instruções que é repetido enquanto o resultado de uma expressão lógica (uma condição) é avaliado como verdadeiro. Abaixo segue a sintaxe desta diretiva enquanto seu o comportamento é ilustrado a seguir.

```
while (expressão_lógica)
  diretiva;
```

Note que a diretiva *while* avalia o resultado da expressão antes de executar a diretiva associada, assim é possível que diretiva nunca seja executada caso a condição seja inicialmente falsa. Um problema típico relacionado a avaliação da condição da diretiva *while* é o seguinte: se a condição nunca se tornar falsa o laço será repetido indefinidamente.

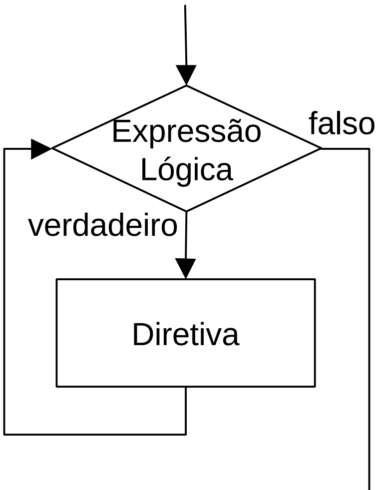


Figura 9 Comportamento da Diretiva *while*

O *do while* também é um laço condicional, isto é, tal como o *while* é um conjunto de instruções repetido enquanto o resultado da condição é avaliada como verdadeira mas, diferentemente do *while*, a diretiva associada é executada antes da avaliação da expressão lógica e assim temos que esta diretiva é executada pelo menos uma vez.

Seguem a sintaxe da diretiva *do while* e uma ilustração do seu comportamento.

```
do
  diretiva
while (expressão_lógica);
```

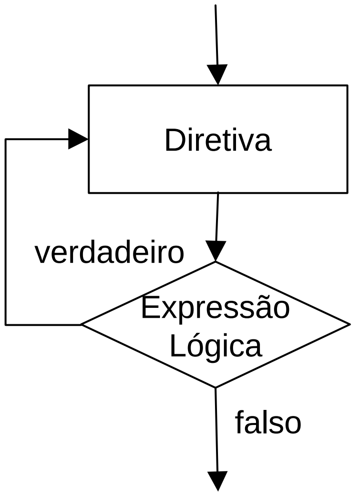


Figura 10 Comportamento da Diretiva *do while*

A seguir temos exemplos da aplicação destas duas diretivas de repetição.

```
// exemploWhile.java

import java.io.*;

public class exemploWhile {

  public static void main (String args[]) {
    int j = 10;
    while (j > Integer.parseInt(args[0])) {
      System.out.println(""+j);
      j--;
    }
  }
}
```

Exemplo 7 Utilização da Diretiva *while*

O exemplo acima ilustra o uso da diretiva *while* tal como um laço de repetição simples equivalente a uma diretiva *for* como abaixo:

```
for (int j=0; j>Integer.parseInt(args[0]); j--) {
  System.out.println(""+j);
}
```

A seguir um exemplo da diretiva *do while*.

```
// exemploDoWhile.java

import java.io.*;

public class exemploDoWhile {

  public static void main (String args[]) {
    int min = Integer.parseInt(args[0]);
    int max = Integer.parseInt(args[1]);
    do {
      System.out.println("" + min + " < " + max);
      min++; max--;
    } while (min < max);
    System.out.println("" + min + " < " + max +
                              " Condicao invalida.");
  }
}
```

Exemplo 8 Utilização da Diretiva *do while*

2.5.4 Estruturas de controle de erros

O Java oferece duas importantes estruturas para o controle de erros muito semelhantes as estruturas existentes na linguagem C++: *try catch* e *try finally*. Ambas tem o propósito de evitar que o programador tenha que realizar testes de verificação e avaliação antes da realização de certas operações, desviando automaticamente o fluxo de execução para rotinas de tratamento de erro. Através destas diretivas, delimita-se um trecho de código que será monitorado automaticamente pelo sistema. A ocorrência de erros no Java é sinalizada através de exceções, isto é, objetos especiais que carregam informação sobre o tipo de erro detectado. Existem várias classes de exceção adequadas para o tratamento do problemas mais comuns em Java, usualmente inclusas nos respectivos pacotes. Exceções especiais podem ser criadas em adição às existentes ampliando as possibilidades de tratamento de erro. Com o *try catch* a ocorrência de erros de um ou mais tipos dentro do trecho de código delimitado desvia a execução automaticamente para uma rotina designada para o tratamento específico deste erro.

A sintaxe do *try catch* é a seguinte:

```
try {
  diretiva_normal;
} catch (exception1) {
  diretiva_de_tratamento_de erro1;
} catch (exception2) {
  diretiva_de_tratamento_de erro2;
}
```

No Exemplo 7, a aplicação exibe uma contagem regressiva baseada num valor inteiro fornecido pelo primeiro argumento (args[0]). Se tal argumento não é fornecido, ocorre uma exceção da classe **java.lang.ArrayIndexOutOfBoundsException** pois o código tenta inadvertidamente usar um argumento que não existe (a mensagem exibida segue abaixo).

```
java.lang.ArrayIndexOutOfBoundsException
        at exemploWhile.main(Compiled Code)
```

Para evitar esse erro o programador poderia testar de forma convencional se foram fornecidos argumentos à aplicação, como no Exemplo 6 (args.length > 0).

A seguir temos uma nova versão do Exemplo 7 que contêm uma estrutura convencional de teste do número de argumentos fornecido.

```
// exemploTeste.java

import java.io.*;

public class exemploTeste {

  public static void main (String args[]) {
    int j = 10;
    if (args.length > 0) {
      while (j > Integer.parseInt(args[0])) {
        System.out.println(""+j);
        j--;
      }
    } else {
      System.out.println("Não foi fornecido um argumento
inteiro");
    }
  }
}
```

Exemplo 9 Aplicação com Teste Convencional

Caso não se forneçam argumentos não ocorrerá erro mas se o primeiro argumento fornecido não for numérico acontecerá a sinalização de uma exceção devido a tentativa conversão efetuada para determinação do limite da contagem a se exibida.

Outra alternativa é a utilização de uma diretiva *try catch* como abaixo:

```
// exemploTryCatch1.java

import java.io.*;

public class exemploTryCatch1 {

  public static void main (String args[]) {
    int j = 10;
    try {
      while (j > Integer.parseInt(args[0])) {
        System.out.println(""+j);
        j--;
      }
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Não foi fornecido um argumento.");
    }
  }
}
```

Exemplo 10 Aplicação da Diretiva *try catch*

Desta forma o código e a rotina de tratamento de erros ficam mais claramente isolados, aumentando a legibilidade e facilidade de modificação do programa. Considerando que o argumento fornecido pelo usuário pode não ser um inteiro válido ocorre uma outra exceção: **java.lang.NumberFormatException** pois a conversão se torna impossível. Contornar esse erro testando a validade do argumento requereria uma rotina extra no modo convencional. Com o uso da diretiva *try catch* teríamos:

```
// exemploTryCatch2.java

import java.io.*;

public class exemploTryCatch2 {

  public static void main (String args[]) {
    int j = 10;
    try {
      while (j > Integer.parseInt(args[0])) {
        System.out.println(""+j);
        j--;
      }
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Não foi fornecido um argumento.");
    } catch (java.lang.NumberFormatException e) {
      System.out.println("Não foi fornecido um inteiro
válido.");
    }
  }
}
```

Exemplo 11 Utilização da Diretiva *try catch* com 2 Exceções

Com o *try finally* temos um comportamento bastante diferente: uma rotina de finalização é garantidamente executada, isto é, o trecho particular de código contido na cláusula *finally* é sempre executado ocorrendo ou não erros dentro do trecho delimitado pela cláusula *try*.

A sintaxe do *try finally* é colocada a seguir:

```
try {
  diretiva_normal;
} finally {
  diretiva_de_tratamento_de erro;
}
```

Isto é particularmente interessante quando certos recursos do sistema ou estruturas de dados devem ser liberadas, independentemente de sua utilização.

3 Java e a Orientação à Objetos

Pretendemos aqui caracterizar o que é a orientação à objetos, suas principais características e vantagens bem como algumas das particularidades do Java com relação à este paradigma de programação. Como apropriado para um texto de caráter introdutório, o tema é coberto de forma superficial de modo que seja possível avançarmos com nosso objetivo: a programação em Java. Maiores detalhes sobre a modelagem, análise e a programação orientada à objetos podem ser obtidos nos inúmeros livros dedicados ao assunto.

3.1 A Programação Orientada à Objetos

Boa parte de nosso entendimento e relacionamento com o mundo se dá através do conceito de objetos. Ao observarmos as coisas e seres que existem ao nosso redor, existe uma natural tendência a tentarmos identificar o que é cada uma destas diferentes entidades. Ao olharmos para uma bola de futebol reconhecemos sua forma esférica, seu tamanho usual, colorido típico e aplicação. Mas a bola que observamos não é a única que existe, inúmeras outras estão “por aí” e mesmo possuindo características idênticas são objetos distintos.

Como o próprio nome diz, a bola é de futebol, ou seja, para ser usada no esporte que denominamos futebol, é um tipo de bola específica. Assim bola de futebol é a mais do que identidade de um certo objeto que estamos observando mas um tipo de objeto que apresenta características próprias (pense na descrição genérica de qualquer bola de futebol). Existem outras bolas (outros tipos) que não são de futebol. Bolas especiais para os múltiplos esportes diferentes e bolas de recreação são exemplos de outros objetos que compartilham das mesmas características de uma bola, ou melhor, de uma esfera.

Pense agora num outro objeto, um liquidificador. Como utilizamos este eletrodoméstico? Através dos controles existentes e da colocação de alimentos líquidos e sólidos em seu copo. O liquidificador pode estar desligado ou ligado em diferentes velocidades. Isto significa que podemos operá-lo através de operações de desligar, ligar, carregar (colocar algo no seu copo), aumentar ou diminuir sua velocidade. Mesmo sem saber como funciona internamente este aparelho somos capazes de usá-lo corretamente. Detalhes de sua estrutura ficam ocultos dentro de sua carcaça pois temos acesso apenas aos controles que o projetista julgou necessário para seu uso doméstico.

Através destes dois exemplos cotidianos descrevemos informalmente as principais características da programação orientada à objetos:

- Identidade
- Classificação
- Polimorfismo
- Hereditariedade
- Encapsulamento

Para nos relacionarmos com os objetos do mundo nós os denominamos de alguma forma, ou seja, todos os objetos tem um nome que os representa. Ao mesmo tempo que cada objeto tem uma identidade e características próprias, somos capazes de reconhecer categorias de objetos, ou seja, grupos de objetos que compartilham características comuns embora distintas em cada objeto (os serem humanos tem características comuns: estrutura e funcionamento do corpo, embora possam ser distinguidos por sua altura, idade, peso, cor e aparência da pele, olhos, cabelos, pelos etc.). É natural para o nosso entendimento de mundo criarmos uma classificação para as coisas, ou seja, criarmos classes de objetos para facilitar nossa compreensão do mundo.

Polimorfismo se refere a nossa capacidade de nos reconhecer num objeto particular um outro mais geral, por exemplo: podemos falar de uma bola de futebol em termos de uma bola genérica, podemos tratar um liquidificador como um eletrodoméstico genérico, podemos tratar morcegos, cachorros e orcas como animais mamíferos.

A hereditariedade é um mecanismos de criarmos novos objetos a partir de outros que já existem, tomando como prontas e disponíveis as características do objeto origem. Isto cria a possibilidade de criarmos várias classes, hierarquicamente relacionadas, partindo de uma mais geral para diversas outras mais especializadas, tal como a classificação proposta pela biologia para classificação dos seres vivos.

Finalmente o encapsulamento indica que podemos utilizar um objeto conhecendo apenas sua interface, isto é, sua aparência exterior, tal como fazemos muitas vezes com computadores, automóveis e outras máquinas de nosso tempo.

A programação orientada à objetos é uma forma de programação que se baseia na construção de classes e na criação de objetos destas classes, fazendo que estes trabalhem em conjunto para que os propósitos da criação de um programa sejam atingidos.

Esta nova forma de programar, embora mais complexa, é muito mais apropriada para o desenvolvimento de sistemas pois permite tratar os problemas de forma semelhante ao nosso entendimento de mundo ao invés de dirigir a solução para as características de funcionamento dos computadores tal como faz o paradigma da programação procedural.

3.2 Classes

Uma classe (*class*) é um tipo de objeto que pode ser definido pelo programador para descrever uma entidade real ou abstrata. Podemos entender uma classe como um modelo ou como uma especificação para certos objetos, ou seja, a descrição genérica dos objetos individuais pertencentes a um dado conjunto. A bola de futebol pode ser uma classe que descreve os objetos bola de futebol. A definição de uma classe envolve a definição de variáveis as quais se associam funções que controlam o acesso a estes. Assim a criação de uma classe implica em definir um tipo de objeto em termos de seus atributos (variáveis que conterão os dados) e seus métodos (funções que manipulam tais dados).

De maneira mais rigorosa, podemos dizer que uma classe é uma abstração que descreve as propriedades relevantes de uma aplicação em termos de sua estrutura (dados) e de seu comportamento (operações) ignorando o restante.

Um programa que utiliza uma interface controladora de um motor elétrico provavelmente definiria a classe **motor**. Os atributos desta classe poderiam ser: número de pólos, velocidade e tensão aplicada. Estes provavelmente seriam representados na classe por tipos como *int* ou *float*. Os métodos desta classe seriam funções para alterar a velocidade, ler a temperatura, etc.

Para uma aplicação do tipo editor de textos talvez fosse conveniente a definição de uma classe **parágrafo** que teria como um de seus atributos uma *string* ou um vetor de *strings* para armazenar o texto do parágrafo e também métodos que operariam sobre este atributo. Cada vez que um novo parágrafo é incorporado ao texto, a aplicação editor cria a partir da classe **parágrafo** um novo objeto contendo as informações particulares do novo trecho de texto. A criação de novos objetos de uma classe se chama instanciação ou simplesmente criação do objeto.

O essencial desta abordagem é que podemos (e devemos) criar novos modelos de dados, as classes, para melhor representar os dados que serão utilizados pela aplicação, isto é, para melhor representarmos os objetos reais a serem tratados pela aplicação. Note que um mesmo objeto pode ser descrito de diferentes maneiras, sendo cada uma destas maneiras mais ou menos adequada para representar tal objeto dentro de um determinado contexto, ou seja, dentro um de um certo problema.

A definição de uma nova classe em Java é um processo muito simples desde que saibamos como descrever um objeto, sendo assim a definição de um objeto é realmente uma tarefa mais complexa do que sua programação como uma classe.

Uma classe em Java pode ser definido através da seguinte estrutura de código:

```
qualificador_de_acesso class Nome_Da_Classe {
  // atributos da classe
  ⋮
  // métodos da classe
  ⋮
}
```

Os qualificadores de acesso indicam como a classe pode ser utilizada por outras classes e, consequentemente, outras aplicações. Existem dois especificadores de acesso básicos:

Qualificador	Significado
public	Indica que o conteúdo público da classe pode ser utilizado livremente por outras classes do mesmo pacote ou de outro pacote.
package	Indica que o conteúdo público da classe pode ser utilizado livremente por outras classes pertencentes ao mesmo pacote.

Para compreendemos melhor o significado destes qualificadores devemos entender primeiramente o que é um pacote para o Java.

Quando declaramos uma classe como pública (*public*) estamos indicando ao compilador que todo o conteúdo da classe indicado como público pode ser irrestritamente utilizado por classes que pertençam ao mesmo pacote, isto é, que esteja localizadas no mesmo diretório, ou que pertençam a outros pacotes (outros diretórios).

O segundo caso é quando indicamos que a classe é do tipo pacote (*package*) o que corresponde a situação padrão, que não precisa ser informada. Uma classe do tipo *package* pode ser utilizada como sendo pública para classes contidas no mesmo diretório, o mesmo não acontecendo para classes de outros pacotes ou diretórios. Isto serve para restringir o uso de classes “de serviço” ao próprio pacote. Por definição todas as classes de um diretório são do tipo pacote, exceto as explicitamente informadas como públicas.

3.2.1 Pacotes (*Packages*)

Um pacote (*package*) em Java nada mais é do que um diretório onde residem uma ou mais classes ou seja, é um conjunto de classes. Usualmente colocam-se num *package* classes relacionadas construídas com um certo propósito. Sob certos aspectos os *packages* reproduzem a idéias das *libraries* (bibliotecas) de outras linguagens de programação. Para facilitar o uso de classes existentes em diferentes diretórios, ou seja, a construção e utilização de pacotes, utiliza-se uma denominação especial para os pacotes, cuja sintaxe é:

```
import nome_do_pacote.Nome_da_classe;
```

Seguem alguns exemplos do uso de pacotes:

```
import java.io.*;
import java.awt.Button;
import br.usf.toolbox.ImageButton;
```

No primeiro indicamos que queremos utilizar qualquer classe do pacote **java.io**, um pacote padrão que acompanha o JDK. No segundo indicamos uma classe específica

(**Button**) do pacote **java.awt** que também acompanha o JDK. Nestes casos, tanto o pacote **java.io** como o **java.awt** devem estar localizados diretamente num dos diretórios especificados como diretórios de classes do JDK (contida na variável de ambiente **classpath** cujo valor default é **\jdk_path\lib\classes.zip**). No terceiro indicamos o uso da classe **ImageButton** do pacote **toolbox** cuja origem é a empresa de domínio www.usf.br (o uso dos nomes de domínio na denominação de pacotes é uma recomendação da *Sun* para que sejam evitadas ambigüidades) ou seja, deve existir um diretório **\br\usf\toolbox** a partir de um dos diretórios de classes do JDK.

Para que uma aplicação ou *applet* Java utilize classes de diferentes pacotes é necessário portanto incluir uma ou mais diretivas *import* que especifiquem tais pacotes. Por *default* apenas as classes do pacote **java.lang**, que contêm as definições de tipos e outros elementos básicos, são importadas.

Para definirmos classes de um pacote devemos adicionar a diretiva *package* com o nome do pacote no início de cada um dos arquivos de suas classes:

```
package br.usf.toolbox;

public class ImageButton {
  ⋮
}
```

3.2.2 Estrutura das Classes

Um classe basicamente possui dois grupos de elementos: a declaração de seus atributos e a implementação de seus métodos. Se desejássemos construir uma nova classe pública denominada **TheFirstClass** estruturalmente deveríamos escrever o seguinte código:

```
// TheFirstClass.java

public class TheFirstClass {
  // declaração de atributos
  ⋮
  // implementação de métodos
  ⋮
}
```

Exemplo 12 Uma Primeira Classe

Ao salvarmos o código correspondente a uma classe em um arquivo devemos tomar os seguintes cuidados:

1. Em um único arquivo Java podem existir várias diferentes definições de classes mas apenas uma delas pode ser pública (se deseja-se várias classes públicas então cada uma delas deve ser salva em arquivos separados).
2. O nome do arquivo deve ser o nome da classe pública, observando-se cuidadosamente o uso do mesmo caixa tanto para o nome da classe como para o nome do arquivo (Java é uma linguagem sensível ao caixa, isto é, diferencia letras maiúsculas de minúsculas em tudo).

Portanto o nome do arquivo para conter a classe pública **TheFirstClass** deve ser TheFirstClass.java. O que inicialmente pode parecer uma restrição é na verdade um mecanismo inteligente e simples para manter nomes realmente representativos para os arquivos de um projeto.

Graficamente representaremos nossas classes como indicado na ilustração a seguir, isto é, utilizaremos a notação proposta pela UML (*Unified Modeling Language*) bastante semelhante a OMT (*Object Modeling Techinique*).

Na notação UML classes são representadas por retângulos contendo o nome da classe. O nome que figura entre parêntesis abaixo do nome da classe é uma extensão da notação para a linguagem Java destinada a indicar o nome do pacote ao qual pertence a classe. A expressão "*from default*" indica que a classe não pertence explicitamente a qualquer pacote, sendo assim considerada do pacote *default* subentendido pelas classes Java presentes no diretório onde se localiza.

Figura 11 Representação de Classe (Notação UML)

Apesar de estar correta a definição de **FirstClass**, nossa classe é inútil pois não contêm nenhum elemento interno. Para que esta classe possa representar algo é necessário que possua atributos ou métodos como veremos nas próximas seções.

3.2.3 Regras para Denominação de Classes

Em Java recomenda-se que a declaração de classes utilize nomes iniciados com letras maiúsculas, diferenciando-se do nomes de variáveis ou instâncias de objetos. Caso o nome seja composto de mais de uma palavras, as demais também deveriam ser iniciadas com letras maiúsculas tal como nos exemplos:

Bola Socket Filter
BolaFutebol ServerSocket DataChangeObserver

A utilização de caracteres numéricos no nome também é livre enquanto o uso do traço de sublinhar (*underscore* '_') não é recomendado.

3.3 Atributos

Um atributo (*attribute*) de uma classe é uma variável pertencente a esta classe que é destinada a armazenar alguma informação que está intrinsecamente associada a classe. Por exemplo, ao falarmos de uma bola qualquer, o tamanho de uma bola, isto é, seu raio, está associado a sua forma geométrica de forma natural pois toda bola deve possuir um tamanho qualquer que seja. Assim sendo é bastante natural definirmos uma classe **Bola** que possua como um atributo uma variável destinada a armazenar seu tamanho ou raio, como abaixo:

```
// Bola.java

public class Bola {
  // Atributos
  float raio;
}
```

Como mostra o trecho de código, a adição de um atributo a uma classe corresponde a simples declaração de uma variável de um certo tipo dentro de sua definição cujo nome deveria indicar seu propósito. Muitas vezes nos referimos aos atributos de uma classe como sendo seus campos (*fields*) ou também como suas variáveis-membro (*members*). Nossa classe **Bola** permite agora representar objetos do tipo bola através de um atributo ou campo raio. Além deste atributo, outros poderiam ser adicionados para melhor modelarmos um objeto desta categoria:

```
// Bola.java
public class Bola {
  // Atributos
  float raio;
  boolean oca;
  int material;
  int cor;
}
```

Exemplo 13 Classe Bola

Temos que a classe **Bola** descrita utiliza vários atributos para descrever diferentes aspectos de um objeto do tipo bola. O tamanho da bola é descrito através de seu raio e como tal tamanho pode ser um número inteiro ou real utilizamos uma variável do tipo *float*. Imaginando que a bola pode ou não ser oca, utilizamos uma variável lógica de nome oca que indica quando *true* tal situação e seu contrário (uma bola maciça) quando *false*. Valores inteiros, representando códigos de material e cor podem ser utilizados para acrescentar outros detalhes na descrição do objeto.

Notem que esta descrição não é a única possível para um objeto bola, podendo ser adequada para resolução de certos problemas e inadequada para outros. O que deve ser ressaltado é que esta classe propõe o que denominamos um modelo conceitual para os objetos bola. A descrição de objetos em termos de classes é o que denominamos modelagem orientada à objetos, uma das etapas mais importantes do projeto de software utilizando a orientação à objetos.

Utilizando a notação UML temos a seguinte representação de uma classe e seus atributos:

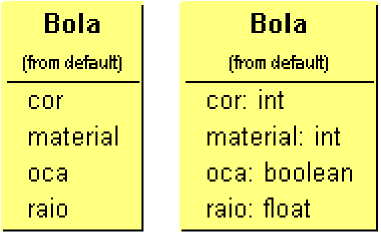


Figura 12 Representação de Classe e Atributos (Notação UML)

Observe que a lista de atributos ocupa uma divisão específica do retângulo definido para a classe. A notação UML admite que sejam especificados apenas os nomes dos atributos, ou seus nomes e tipos ou ainda os nomes, tipos e valores iniciais dos atributos conforme a conveniência desejada.

A classe **TheFirstClass**, do Exemplo 12 poderia ser ampliada com a adição de atributos tal como o código e o diagrama UML ilustrados a seguir.

```
// TheFirstClass.java

public class TheFirstClass {
  // Atributos
  public int campoInt;
  public double campoDouble;
  private boolean campoBoolean;
}
```

Exemplo 14 Classe TheFirstClass Ampliada

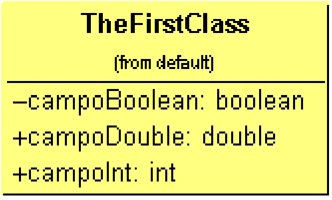


Figura 13 A Classe TheFirstClass (Notação UML)

Note que os campos indicados no diagrama são precedidos por um caractere cujo significado é dado na tabela a seguir:

Símbolo UML	Significado
-	O campo indicado é privado (*private*).
+	O campo indicado é público (*public*).
#	O campo indicado é protegido (*protected*).
(em branco)	O campo indicado é pacote (*package*).

Tabela 9 Notação UML para Atributos e Operações de Classe

Agora como utilizar os atributos de uma classe? Devemos lembrar que uma classe é um modelo de objeto, ou seja, a classe em si não representa nenhum objeto particular sendo necessário criarmos um objeto para efetivamente utilizarmos a classe, isto é, os atributos e métodos ali definidos. Os atributos pertencem à classe mas os valores associados aos atributos que descrevem um certo objeto não pertencem à classe mas ao objeto em si.

3.4 Instanciação

A criação de um objeto é o que chamamos instanciação. Instanciar significa criar uma instância da classe (*class instance*), isto é, um novo objeto que pode ser descrito através desta classe. Enquanto uma classe é um modelo abstrato de um objeto, uma instância representa um objeto concreto desta classe. A classe **Bola** do Exemplo 13 representa um modelo abstrato de bola enquanto cada bola que existe fisicamente é uma instância desta classe, ou seja, um objeto concreto deste tipo. Isto significa que utilizamos as mesmas características para descrever objetos diferentes, ou seja, uma bola de raio 4 cm, maciça, de madeira e amarela é uma bola diferente de outra de raio 1 m, oca, de plástico, azul, embora descritas através das mesmas características (raio, oca ou maciça, material e cor). Assim os atributos de uma classe são variáveis-membro desta classe que contêm informação representativa sobre um certo objeto deste tipo.

Do ponto de vista computacional, a instanciação corresponde a alocação de memória para armazenar informações sobre um certo objeto, ou seja, a reserva de uma porção de memória para guardar os valores associados aos atributos que descrevem um objeto de uma certa classe. A classe, através de sua definição, contêm informações suficientes para que seja determinada a quantidade de memória necessária ao armazenamento de um objeto do seu tipo. Enquanto este processo de ocupação de memória e sua organização interna são absolutamente transparentes para o programador, é este quem determina quando e como instanciar novos objetos, ou seja, é o programador aquele quem cria novos objetos para serem utilizados em um programa.

Para instanciarmos um novo objeto devemos utilizar o operador *new*, destinado a criação de novos objetos como segue:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

Esta construção possui duas partes: a declaração de um objeto e a instanciação propriamente dita. A parte esquerda é semelhante a uma declaração de variáveis onde indicamos um tipo e um nome para a variável. No nosso caso dizemos ser um objeto por corresponde a uma instância de uma classe. A parte direita é a instanciação do objeto, onde usamos o operador *new* para indicar que desejamos a criação um novo objeto de uma certa classe, o que é feito através de um método especial denominado construtor (métodos e construtores serão tratados a seguir, nas seções 3.5 e 3.7). Com isto efetuamos a criação de um novo objeto e guardamos uma referência para sua utilização na variável objeto. Para a criação de um objeto do tipo **Bola** poderíamos escrever:

```
Bola objeto = new Bola();
```

Desta forma objetos diferentes podem ser criados e referenciados através de variáveis objeto com nomes distintos:

```
Bola bolaDoJoao = new Bola();
Bola minhaBola = new Bola();
Bola outraBola = new Bola();
Bola bola1 = new Bola();
```

Portanto um mesmo programa pode utilizar vários diferentes objetos de uma mesma classe, tal qual podemos manusear diferentes objetos do mesmo tipo, mantendo-se apenas uma referência distinta para cada objeto (uma variável objeto diferente para cada um deles).

A classe **Bola** define atributos raio, oca, material e cor para seus objetos. Para indicarmos qual destes atributos desejamos utilizar utilizamos um outro operador denominado seletor (*selector*) indicado por um caractere ponto (“.”) como segue:

```
nomeDoObjeto.nomeDoAtributo
```

Esta construção pode ser lida como: selecione o atributo **nomeDoAtributo** do objeto **nomeDoObjeto**. Com o nome da variável objeto selecionamos o objeto em si e através do seletor indicamos qual atributo deste objeto que desejamos utilizar. Usar um atributo pode significar uma de duas operações: (i) consultar seu conteúdo, ou seja, ler ou obter seu valor e (ii) determinar seu conteúdo, isto é, escrever ou armazenar um certo valor neste atributo. A primeira situação envolve o atributo numa expressão qualquer enquanto na segunda situação temos o atributo recebendo o resultado de uma expressão qualquer através de uma operação de atribuição.

Observe o exemplo a seguir onde utilizamos a classe Bola num pequeno programa que instancia dois objetos diferentes e utiliza os atributos destes objetos:

```
// DuasBolas.java

public class DuasBolas {

  public static void main(String args[]) {
    // Instanciando um objeto
    Bola bola1 = new Bola();
    // Armazenando valores em alguns dos atributos deste objeto
    bola1.raio = 0.34;
    bola1.oca = false;
    bola1.cor = 10;
    // Instanciando um outro objeto
    Bola bola2 = new Bola();
    // Armazenando valores em alguns atributos do outro objeto
    bola2.oca = true;
    bola2.material = 1324;
    // Usando valores armazenados
    bola2.raio = 5 * bola1.raio;
    bola2.cor = bola1.cor;
    System.out.println(“Bola1:”);
    System.out.println(“  raio = “ + bola1.raio);
    System.out.println(“  oca  = “ + bola1.oca);
    System.out.println(“  cor  = “ + bola1.cor);
    System.out.println(“Bola2:”);
    System.out.println(“  raio = “ + bola2.raio);
    System.out.println(“  oca  = “ + bola2.oca);
    System.out.println(“  cor  = “ + bola2.cor);
  }
}
```

Exemplo 15 Instanciação de Objetos

Compilando e executando este programa temos que são exibidos os diferentes valores armazenados nos atributos dos dois objetos **Bola** criados.

A combinação **objeto.atributo** pode ser utilizada como uma variável comum do mesmo tipo definido para o atributo na classe objeto, permitindo a fácil manipulação do dados armazenados internamente no objeto. Como visto, valores dos atributos de um objeto podem ser utilizados diretamente em qualquer expressão e operação válida para seu tipo. Assim variáveis comuns podem receber valores de atributos de objetos e vice-versa, desde que de tipos compatíveis, como exemplificado:

```
int a = 7, b;
bola1.cor = a;
b = bola2.material;
bola1.material = b – 2;
bola2.cor = bola1.cor + 3;
if (bola2.cor == 0)
  System.out.println(“Bola branca”);
```

3.5 Métodos

Enquanto os atributos permitem armazenar dados associados aos objetos, ou seja, valores que descrevem a aparência ou estado de um certo objeto, os métodos (*methods*) são capazes de realizar operações sobre os atributos de uma classe ou capazes de especificar ações ou transformações possíveis para um objeto. Isto significa que os métodos conferem um caráter dinâmico aos objetos pois permitem que os objetos exibam um comportamento que, em muitos casos, pode mimetizar (imitar) o comportamento de um objeto real.

Outra forma de entender o que são os métodos é imaginar que os objetos são capazes de enviar e receber mensagens, de forma tal que possamos construir programas onde os objetos trocam mensagens proporcionando o comportamento desejado. A idéia central contida na mensagem que pode ser enviada ao objeto é mesma contida quando ligamos um liquidificar: acionar o botão que liga este aparelho numa de suas velocidades corresponde a enviar uma mensagem ao liquidificar “ligue na velocidade x”. Quando fazemos isto não é necessário compreender em detalhe como funciona o liquidificador mas apenas entender superficialmente como operá-lo.

Imaginemos uma lâmpada incandescente comum. A despeito de detalhes técnicos (tensão de operação, potência, frequência etc.) podemos assumir que uma lâmpada pode estar acesa ou apagada, o que caracteriza um estado lógico acesa (acesa = *true* ou acesa = *false*). Nós não fazemos uma lâmpada acender diretamente mas acionamos algo que faz com que ela funcione como desejado, assim, usualmente enviamos mensagens do tipo ligar e desligar para a lâmpada. Para construirmos uma classe para modelar uma lâmpada e o comportamento descrito poderíamos ter:

```
// Classe Lampada

public class Lampada {
  // atributos
  boolean acesa;

  // metodos
  public void ligar() {
    acesa = true;
  }
  public void desligar() {
    acesa = false;
  }
}
```

Exemplo 16 Classe Lampada

A classe **Lampada** modelada possui um atributo lógico **acesa** que identifica o estado atual da lâmpada. Além deste atributo podemos identificar dois métodos **ligar** e

desligar que fazem com que o estado da lâmpada seja adequadamente modificado para verdadeiro e falso conforme a mensagem enviada a lâmpada.

Notamos que um método nada mais é do que um bloco de código para o qual se atribui um nome como o método **ligar** destacado abaixo:

```
public void ligar() {
    acesa = true;
  }
```

O método é declarado como público pois desejamos que seja utilizado externamente a classe **Lampada** pelas classes que criem instâncias do objeto Lâmpada. A palavra reservada *void* indica que não existe um valor de retorno, isto é, o método **ligar** não devolve uma mensagem informando o resultado da operação (neste caso não era necessário).

Os parêntesis após o nome **ligar** tem duplo propósito: diferenciam a construção de um método da declaração de uma variável e permitem que sejam especificados valores auxiliares que podem ser enviados em anexo a mensagem para informar mais precisamente a forma com que a ação deve ser realizada (neste caso nada é especificado indicando que nada mais é necessário além da própria mensagem “ligar”). A especificação dos valores que um método recebe é denominada lista de parâmetros formais ou simplesmente lista de parâmetros. Os valores que são passados para o método quando de seu uso são chamados argumentos.

A representação da classe **Lampada** através da notação UML é dada a seguir. Note que as operações definidas para a classe são colocadas numa outra divisão.

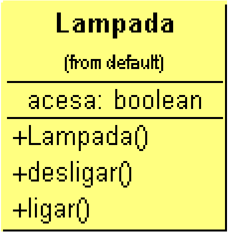


Figura 14 Classe Lâmpada (Notação UML)

Ao declararmos uma classe é comum a colocação da declaração de todos os seus métodos após a declaração dos seus atributos, porém podemos intercalar a declaração de métodos e atributos ou usar qualquer outra ordem que nos convenha. Existem autores e programadores que preferem concentrar a declaração dos atributos no final da classe, mas é apenas uma questão de estilo de escrita de código.

Recomenda-se de qualquer forma que sejam utilizadas linhas de comentários para delimitar as áreas código das classes, ou seja, comentários separando a seção dos atributos da seção dos métodos ou mesmo pequenos comentários para cada declaração contida na classe.

Para utilizarmos um método de uma classe procedemos de forma análoga ao uso de atributos, isto é, escolhemos o objeto através de seu nome e com o operador de seleção (*selector*) indicamos qual o método deve ser utilizado. Se o método recebe argumentos, os mesmo devem colocados no interior dos parêntesis separados por vírgulas, caso contrário os parêntesis permanecem vazios.

```
nomeDoObjeto.nomeDoMetodo(argumentos)
```

Assim para invocar o método **ligar** ou **desligar** devemos usar um dos trechos de código exemplificados abaixo, onde objeto é o nome da variável objeto que se refere a uma instância da classe **Lampada**:

```
objeto.ligar();
objeto.desligar();
```

A seguir um programa exemplo onde instanciamos um objeto **Lampada** e utilizamos seus métodos:

```
// LampTest.java

public class LampTest {

  static public void main (String args[]) {
    Lampada lamp = new Lampada();
    System.out.println("Lampada Criada (Acesa=" +lamp.acesa
+")");
    lamp.ligar();
    System.out.println("Lampada Ligada (Acesa=" +lamp.acesa
+")");
    lamp.desligar();
    System.out.println("Lampada Deslig (Acesa=" +lamp.acesa
+")");
  }
}
```

Exemplo 17 Teste da Classe Lampada

No exemplo temos inicialmente a instanciação de um novo objeto do tipo **Lampada** seguido da exibição do estado inicial do objeto através do uso direto de seu atributo **acesa**. Após isto é utilizado o método **ligar** que modifica o estado da lâmpada como indicado por outro comando de saída. Finalmente é usado o método **desligar** que faz com que o estado da lâmpada retorne ao estado inicial. Compile e execute este programa para visualizar e comprovar seus resultados.

Na classe **Lampada** todos os métodos definidos não recebiam argumentos tão pouco retornavam informações sobre o resultado de sua operação. Retomemos a idéia do liquidificador para exemplificarmos métodos que recebem argumentos e que retornam valores.

Imaginemos um liquidificador comum, de copo fixo, que pode operar em duas velocidades diferentes e que possui um controle analógico deslizante de velocidades. Se a primeira velocidade é a baixa, temos que ao ligar passamos primeiramente por esta velocidade e nesta, basta retornar a posição desligado. Para chegar na velocidade alta é necessário passar pela velocidade baixa, sendo que para desligar o liquidificador estando na velocidade alta também é necessário passar pela velocidade baixa. Esquematicamente poderíamos representar a troca de velocidades do liquidificador da maneira a seguir:

Desligado → Velocidade Baixa
Desligado ← Velocidade Baixa
Desligado → Velocidade Baixa → Velocidade Alta
Desligado ← Velocidade Baixa ← Velocidade Alta

Com isto temos definidos os estados de operação do liquidificador, que poderiam ser representado por um atributo **velocidade**. Este atributo poderia ser do tipo inteiro ficando convencionado que 0 equivale a desligado, 1 a velocidade baixa e 2 a velocidade alta. Por hora desconsideraremos quaisquer outros aspectos de funcionamento e uso do liquidificador, simplificando o modelo utilizado para definirmos uma classe **Liquidificador**.

Observando a transição entre os estados podemos imaginar uma mensagem que solicite o aumento de velocidade do liquidificador e outra que solicite sua diminuição, portanto métodos **aumentarVelocidade** e **diminuirVelocidade**. Estes dois métodos não necessitam parâmetros pois suas mensagens são inequívocas, tão pouco é necessário que retornem valores. Poderíamos ter também um outro método que retornasse qual a atual velocidade do liquidificador, por exemplo **obtemVelocidade**, que igualmente aos anteriores também não necessita receber parâmetros.

Agora a classe **Liquidificador**, conforme definido, possui um atributo e três métodos distintos, sendo que um código Java possível para a implementação desta classe é dado a seguir:

```
// Liquidificador.java

public class Liquidificador {
  // atributos
  private int velocidade;

  // metodos
  public void aumentarVelocidade() {
    switch (velocidade) {
      case 0:
      // se desligado passa p/ vel. baixa
        velocidade++;
        break;
      case 1:
      // se vel. baixa passa p/ vel. alta
        velocidade++;
        break;
      // se vel. alta nao faz nada (mantem vel.)
    }
  }

  public void diminuirVelocidade() {
    switch (velocidade) {
      case 1:
      // se vel. baixa passa p/ desligado
        velocidade--;
        break;
      // se vel. alta passa p/ vel. baixa
      case 2:
        velocidade--;
        break;
      // se desligado nao faz nada
    }
  }

  public int obterVelocidade() {
    return velocidade;
  }
}
```

Exemplo 18 Classe Liquidificador

Note que o atributo **velocidade** foi declarado como privado (*private*). Como será visto na seção 3.6 Acessibilidade, este qualificador de acesso indica que a variável-membro **velocidade** não pode ser acessada externamente a classe, ou seja, não pode ser lida ou escrita exceto pelos métodos da própria classe, o que é feito pelos métodos **aumentarVelocidade**, **diminuirVelocidade** e **obterVelocidade**.

Os métodos **aumentarVelocidade** e **diminuirVelocidade** são estruturalmente semelhantes: não recebem argumentos (a lista de parâmetros formais está vazia) nem devolvem valores (o retorno é declarado como *void*) e internamente utilizam uma estrutura de desvio múltiplo de fluxo (*switch*) que permite selecionar a ação adequada para cada situação, no caso, a velocidade do liquidificador. Estes métodos apenas modificam o estado do liquidificador quando é possível, não realizando ação alguma em outros casos. Como externamente o atributo velocidade não é acessível, torna-se impossível fazer que este atributo assuma valores inconsistentes, ou seja, velocidades não especificadas para o liquidificador. Da mesma forma é igualmente simples adicionar-se novas velocidades ao código.

O diagrama UML da classe Liquidificador é ilustrado a seguir.

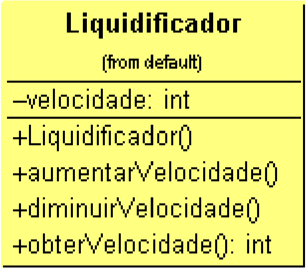


Figura 15 Diagrama UML da Classe Liquidificador

Finalmente o método **obterVelocidade**, que também não recebe argumentos, retorna o valor do atributo **velocidade**, pois este não pode ser diretamente lido por qualquer outra classe. Como este método devolve um valor seu uso somente é efetivo quando o resultado for empregado num expressão qualquer. A devolução de valores emprega a diretiva *return*. No diagrama UML da Figura 15 é indicado o tipo de retorno do método **obterVelocidade**. O exemplo a seguir demonstra a funcionalidade da classe liquidificador.

```
// LiquiTest.java

public class LiquiTest {

  public static void main(String args[]) {
    Liquidificador l = new Liquidificador();
    System.out.println("Liquidificador");
    System.out.println("> Vinicial = " + l.obterVelocidade());
    for(int v=0; v<5; v++) {
      System.out.print("> aumentando velocidade...");
      l.aumentarVelocidade();
      System.out.println("Vatual = " + l.obterVelocidade());
    }
    for(int v=0; v<5; v++) {
      System.out.print("> diminuindo velocidade...");
      l.diminuirVelocidade();
      System.out.println("Vatual = " + l.obterVelocidade());
    }
  }
}
```

Exemplo 19 Teste da Classe Liquidificador

O programa tenta aumentar e diminuir a velocidade do liquidificador instanciado mais do que realmente é possível, mas como antecipado, a classe não admite valores inconsistentes mantendo a velocidade dentro dos limites estabelecidos.

Imaginando agora um liquidificador de funcionalidade idêntica mas com controle digital de velocidades, ou seja, botões para velocidade 0 (desligado), velocidade baixa e velocidade alta. Agora é possível passar de qualquer velocidade para qualquer outra, não existindo uma sequência de estados como no liquidificador analógico, assim não temos um comportamento distinto para o aumento ou diminuição de velocidades apenas uma troca de velocidades.

Poderíamos definir esta classe como segue:

```
// Liquidificador2.java

public class Liquidificador2 {
  // atributos
```

```
  protected int velocidade;

  // metodos
  public void trocarVelocidade(int v) {
    // verifica se velocidade informada valida
    if (v>=0 && v<=2) {
      velocidade = v;
    }
  }

  public int obterVelocidade() {
    return velocidade;
  }
}
```

Exemplo 20 Classe Liquidificador2 (controle digital)

Nesta nova classe o atributo **velocidade** é definido como protegido (*protected*) de forma que possa ser eventualmente utilizado por classes baseadas na classe **Liquidificador2** (os conceitos de herança serão vistos na seção 5.3). O novo diagrama UML para esta classe seria como abaixo. Note a semelhança com o mostrado para o exemplo anterior e a indicação de tipo para argumento de métodos e valor de retorno.

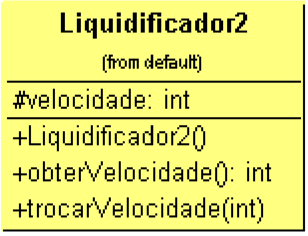


Figura 16 Diagrama UML da Classe Liquidificador2

O método **trocarVelocidade** é agora o único método responsável pela troca de velocidades do liquidificador. Tal como o uso dos botões, onde cada um corresponde ao acionamento do liquidificador numa dada velocidade, seu uso depende de um argumento que indicará qual a velocidade com que o liquidificador deve funcionar. Obviamente, se for especificada uma velocidade inconsistente não ocorre mudança alguma no estado do liquidificador (você não pode acionar um botão que não existe e portanto tal atitude não tem efeito prático).

Para acionarmos este método com seu argumento devemos escrever:

```
objeto.trocarVelocidade(expressão)
```

Onde como expressão entende-se qualquer expressão que produza um resultado inteiro, ou seja, um resultado do mesmo tipo esperado como argumento do método em questão. Desta forma consegue-se associar a troca de velocidades do liquidificador com o resultado numérico de uma expressão aritmética.

A seguir um exemplo de aplicação que pode testar a classe **Liquidificador2** como anteriormente realizado:

```
// LiquiTest2.java

public class LiquiTest2 {

  public static void main(String args[]) {
    Liquidificador2 l = new Liquidificador2();
    System.out.println("Liquidificador2");
    System.out.println("> Vinicial = " + l.obterVelocidade());
    for(int v=0; v<5; v++) {
      System.out.print("> aumentando velocidade...");
      l.trocarVelocidade(v);
```

```
      System.out.println("Vatual = " + l.obterVelocidade());
    }
    for(int v=4; v>=0; v--) {
      System.out.print("> diminuindo velocidade...");
      l.trocarVelocidade(v);
      System.out.println("Vatual = " + l.obterVelocidade());
    }
  }
}
```

Exemplo 21 Teste da Classe Liquidificador2

Embora o resultado da execução desta aplicação seja o mesmo da classe anterior, ficam caracterizados métodos capazes de:

(i) apenas efetuar operações,
(ii) efetuar operações retornando um resultado e
(iii) receber valores efetuando operações.

Combinando-se o recebimento de argumentos com o retorno de valores teríamos o último tipo de método, que será exemplificado mais a frente.

3.6 Acessibilidade

A acessibilidade de uma classe, método ou atributo de uma classe é a forma com que tal elemento pode ser visto e utilizado por outras classes. Este conceito é mais conhecido como encapsulamento ou *data hiding* (ocultamento de dados) sendo muito importante dentro da orientação à objetos. A determinação da acessibilidade de uma classe ou membro de classe é feita pelos qualificadores de acesso (*access qualifiers*) ou seja, por palavras reservadas da linguagem que definem o grau de encapsulamento exibido por uma classe e seus elementos, isto é, são especificações para restringir-se o acesso as declarações pertencentes a uma classe ou a própria classe como um todo. Como visto em parte, isto pode ser feito através do uso das palavras reservadas *public* (público), *private* (privado), *package* (pacote) e *protected* (protegido).

Restringir o acesso a certas partes de uma classe equivale a controlar seu uso, o que oferece várias vantagens dentre elas:

(i) o usuário da classe passa a conhecer apenas aquilo que é necessário para o uso da classe,
(ii) detalhes da forma de implementação são ocultos dos usuários da classe,
(iii) pode-se assegurar que certos atributos tenham valores restritos a um conjunto mantendo-se a consistência e a integridade dos dados armazenados na classe e
(iv) pode-se garantir que determinadas operações sejam realizadas numa certa sequência ou tenham seu funcionamento condicionado a estados da classe.

Isto é possível porque através dos qualificadores de acesso existentes podemos ocultar partes da classe, fazendo apenas que outras partes selecionadas sejam visíveis aos usuários desta classe. Como no caso do Exemplo 18, onde o atributo **velocidade** da classe **Liquidificador** não era visível externamente, impedindo que valores inconsistentes fossem armazenados nele, pois seu uso só podia ser indiretamente realizado através de métodos pertencentes a classe programados de forma a garantir a consistência deste atributo.

Se instanciássemos um objeto Liquidificador e tentássemos usar diretamente o atributo velocidade:

```
Liquidificador l = new Liquidificador();
l.velocidade = 3;
```

Obteríamos o seguinte erro durante a compilação que indica que o atributo existe mas não pode ser utilizado:

```
T.java:4: Variable velocidade in class Liquidificador not
accessible from class T.
```

Este erro é diferente daquele obtido na tentativa de usar-se um atributo inexistente dentro da classe:

```
l.capacidade = 7;
```

Cujo erro indicado seria:

```
T.java:5: No variable capacidade defined in class
Liquidificador.
```

A restrição do acesso a certas partes de uma classe representa permite a construção do que se chama interface do usuário da classe e interface do programador da classe. A interface do usuário é composta pelos atributos e métodos públicos de uma classe. Sendo públicos tais atributos e métodos podem ser livremente acessados por instâncias desta classe existentes em outras classes. A interface do programador é composta pela interface do usuário e por todos os atributos e métodos protegidos de uma classe. Atributos e métodos protegidos só podem ser acessados por novas classes que sejam criadas através do mecanismo de herança, a ser discutido na seção 5.3. Os outros elementos, declarados como privados são sempre exclusivos da classe, relacionando-se com a implementação da classe em si. A definição destas interfaces e o ocultamento de detalhes de implementação são os meios utilizados para se obter o adequado encapsulamento de atributos e operações da classe.

Quando não se especifica se um atributo ou método tem acesso público, protegido ou privado o Java assume um quarto tipo especial de acesso denominado de pacote (*package*). O tipo pacote é equivalente ao acesso público para outras classes pertencentes ao mesmo pacote, protegido para subclasses e privado para instâncias das subclasses. Abaixo uma pequena tabela esquematizando a acessibilidade de elementos declarados em uma classe quando consideradas as situações de implementação (codificação) e instanciação (criação de objetos dentro de programas).

Métodos e Atributos da Classe	**Implementação da Classe**	**Instância da Classe**
privados **(***private***)**	sim	não
protegidos **(***protected***)**	sim	não
pacote **(***package***)**	sim	sim
públicos **(***public***)**	sim	sim

Tabela 10 Acessibilidade de Membros de uma Classe

Num primeiro momento os especificadores de acesso *private* e *protected* parecem se equivaler tal como *package* e *public* mas caracterizam níveis de acesso bastante distintos quando da criação de novas classes a partir de outras. As diferenças entre eles serão melhor evidenciadas quando tratarmos do mecanismo de herança.

Através do encapsulamento obtêm-se várias vantagens:

(i) o código se torna mais claro pois ficam evidentes os membros que oferecem funcionalidades reais daqueles que são auxiliares,

(ii) minimizam-se os erros de programação pois as interfaces tornam-se mais simples,

(iii) classes semelhantes podem exibir interfaces semelhantes pois detalhes de implementação permanecem ocultos, facilitando sua utilização,

(iv) proporciona facilidades para extensão (criação de novas classes a partir de classes existentes) e

(v) oferece facilidade de modificação (como a classe é conhecida pela sua interface, modificações que não afetem a interface são transparentes para os usuários da classe).

Talvez a vantagem mais importante seja a facilidade de extensão pois através destes mecanismos é que podemos realmente obter a reusabilidade e a padronização do código que são os maiores benefícios da programação orientada à objetos.

Hoje tais benefícios são nitidamente explorados na criação de famílias de componentes de software reutilizáveis, que são robustos, fáceis de usar e ainda permitem a criação de extensões que os modifiquem conforme desejado conferindo grande versatilidade à sua utilização. O pacote **awt** (*Abstract Window Toolkit*) assim como o JFC (*Java Foundation Classes*) são alguns exemplos de conjuntos reutilizáveis de componentes Java que exploram tais conceitos e benefícios.

Resumidamente podemos afirmar que o encapsulamento permite separar o projeto (*design*) da implementação (*coding*), separação esta uma das grandes preocupações da engenharia de software. Há quem diga que o Java se parece muito com a linguagem C quando estamos preocupados com codificação (aspectos de implementação pura), mas quando estamos preocupados com o projeto, o Java se assemelha mais ao *Smalltalk*, uma linguagem puramente orientada à objetos extremamente voltada para os aspectos de projeto e clareza de definição. É necessário enfatizar que um bom projeto orientado à objetos pode ser facilmente implementado na maioria das linguagens de programação orientadas à objeto, o que não é verdade quando nos referimos apenas a uma boa implementação.

3.7 Construtores

A criação de novos objetos, como discutido na seção 3.4, é o que chamamos de instanciação. Nos vários exemplos vistos, a criação de um novo objeto sempre teve a forma abaixo:

```
Liquidificador meuLiquidificador = new Liquidificador();
```

Foi colocado que o operador *new* é o responsável pela criação de um novo objeto, mas isto é apenas uma meia verdade. O operador *new* sempre precede uma construção semelhante a uma chamada comum de um método qualquer mas cujo nome é o mesmo que o da classe cujo objeto estamos instanciando. Este método é o que denominamos construtor do objeto ou apenas construtor (*constructor*).

Os construtores são métodos especiais chamados pelo sistema no momento da criação de um objeto, assim sendo o operador *new* apenas indica que o método especial construtor de uma certa classe será utilizado. O resultado da invocação de um construtor é uma referência para a área de memória onde foi criado o novo objeto, ou seja, o construtor é responsável pela alocação de memória para o novo objeto, realizar operações preparatórias para o funcionamento do objeto (tais como abertura de arquivos ou estabelecimento de comunicação ou inicialização de dispositivos periféricos ou *drivers* de *software*) e também uma oportunidade de inicializar seus atributos com valores consistentes e adequados ao objeto que está sendo criado. Em um construtor podemos efetuar a chamada de outros métodos da classe ou mesmo criarmos novos objetos, isto é, utilizarmos construtores de outras classes, possibilitando a construção de objetos bastante sofisticados.

Através dos construtores podemos criar objetos concretos, isto é, estruturas de dados e operações que representam um objeto verdadeiro do mundo real, a partir de uma classe (um modelo abstrato de objetos de um certo conjunto).

Um construtor obrigatoriamente tem sempre o mesmo nome da classe a qual pertence. Outro ponto importante é que os construtores sempre devem ser especificados como públicos (*public*) e nunca podem ser declarados como abstratos (*abstract*). Tomando como exemplo a classe **String**, que está definida dentro do *package* **java.lang**, temos que existe um construtor **String()**, que não recebe argumentos, tal como nos exemplos vistos anteriormente (classe **Lampada** e seu construtor **Lampada()**, classe **Liquidificador** e seu construtor **Liquidificador()**). Inspecionando as classes construídas notamos que não existe menção a um construtor, assim como estamos usando um método que não foi declarado?

Para classes onde não exista um construtor explicitamente definido, o Java assume a existência de um construtor denominado *default*. O construtor *default* é um método de mesmo nome que a classe, sem argumentos, que inicializa as variáveis membro existentes na classe da seguinte forma: variáveis inteiras e reais recebem zero, variáveis lógicas recebem false e variáveis objeto recebem o valor *null* (sem referência). Isto é uma das facilidades oferecidas pelo Java: construtores *default* e inicialização automática de variáveis. Em todos os exemplos anteriores usamos os construtores default das classes construídas.

Para a classe **Lampada** (Exemplo 16) poderíamos ter definido explicitamente o seguinte construtor, que substitui o construtor *default*, possuindo funcionalidade equivalente:

```
// Cria um objeto Lampada no estado apagada
public Lampada() {
  acesa = false;
}
```

Exemplo 22 Um Construtor para Classe Lampada

Da mesma forma a classe **Liquidificador** (Exemplo 18) poderia ter sido definida contendo o seguinte construtor:

```
// Cria um objeto Liquidificador no estado desligado
public Liquidificador() {
  velocidade = 0;
}
```

Exemplo 23 Um Construtor para Classe Liquidificador

Um construtor semelhante também poderia ser adicionado a classe **Liquidificador2** (Exemplo 20). A adição destes construtores ao código não implica na necessidade de qualquer modificação nos programas de teste já elaborados para estas classes dado que tais construtores apenas substituem o construtor *default* utilizado. Nos diagramas UML da Figura 15 e da Figura 16 temos a indicação dos construtores destas classes.

Tal como para métodos comuns, podemos criar construtores que recebem argumentos, de forma que possamos inicializar o objeto de uma maneira particular. Estes construtores também são chamados de construtores parametrizados. Retomando a classe **java.lang.String**, outro construtor existente é um que aceita uma *string* literal (uma cadeia de caracteres literal) como argumento. Veja o exemplo a seguir:

```
// UseStrings.java

public class UseStrings {

  public static void main(String args[]) {
    String s0 = null;
    String s1 = new String();
    String s2 = new String("Alo Pessoal!");
    System.out.println("Testando construtores de Strings:");
    System.out.println("s0 = " + s0);
    System.out.println("s1 = " + s1);
```

```
    System.out.println("s2 = " + s2);
  }
}
```

Exemplo 24 Teste do Construtores da Classe String

Executando o programa obtemos como resultado a impressão dos valores *null*, branco e “Alo Pessoal!” correspondentes as respectivas inicializações destas variáveis.

Uma classe pode possuir vários construtores, todos com o mesmo nome obrigatório, diferenciando-se apenas pelas suas listas de argumentos (isto é o que chamamos: sobrecarga de construtor ou *constructor overload*, assunto que será visto mais a frente na seção 5.2). A possibilidade de uma classe possuir diversos construtores facilita seu uso em diferentes circunstâncias. Assim poderíamos adicionar o seguinte construtor a classe **Lampada** do Exemplo 16 que faz com que os novos objetos sejam criados num dado estado inicial.

```
// Cria um objeto Lampada no estado dado
public Lampada(boolean estado) {
  acesa = estado;
}
```

Exemplo 25 Um Construtor Adicional para Classe Lampada

3.8 Destrutores e a Coleta de Lixo

Os destrutores (*destructors*) também são métodos especiais que liberam as porções de memória utilizada por um objeto, ou seja, enquanto os construtores são responsáveis pela alocação de memória inicial e também preparo do objeto, os destrutores encerram as operações em andamento, fechando arquivos abertos, encerrando comunicação e liberando todos os recursos alocados do sistema, “apagando” todos os vestígios da existência prévia do objeto.

Na linguagem C++ os destrutores são acionados através de operações de eliminação de objetos acionadas pelo comando *delete*. Assim como os construtores, os destrutores do C++ possuem o mesmo nome da classe embora precedidos pelo caractere til (‘~’).

A importância dos destrutores reside no fato de que devemos devolver ao sistema os recursos utilizados pois caso contrário tais recursos (memória, *handlers* para arquivos etc.) podem se esgotar impedindo que este programa e os demais existentes completem suas tarefas.

Um dos maiores problemas do desenvolvimento de aplicações é justamente garantir a correta devolução dos recursos alocados do sistema, concentrando-se este problema na devolução da memória ocupada. Quando um programa não devolve ao sistema a quantidade integral de memória alocada e como se o sistema estivesse “perdendo” sua memória, termo conhecido como *memory leakage*. Um exemplo disto em C++ seria:

```
Lampada l;          // variável objeto
l = new Lampada();  // l recebe a referencia de um novo objeto
l = new Lampada();  // l recebe uma outra referencia de um 
objeto
                    // a referência anterior é perdida, ou seja,
                    // o primeiro objeto alocado fica perdido na
                    // memória
```

Exemplo 26 Simulação de *Memory Leakage*

Ciente destes problemas, os desenvolvedores do Java incorporaram ao projeto da máquina virtual um gerenciador automático de memória que é executado paralelamente ao programa Java. Cada vez que um objeto deixa de ser referenciado, isto é, sai do escopo do programa ficando perdido na memória, ele é marcado para eliminação futura. Quando o programa fica ocioso, esperando por entrada de dados por exemplo, o gerenciador automático de memória destroi os objetos perdidos, recuperando a memória perdida. Se a quantidade de memória também se esgota, este mecanismo procura por objetos perdidos para recuperar memória de imediato. Este inteligente mecanismo de eliminação de objetos perdidos e consequente recuperação de memória do sistema é conhecido como coletor automático de lixo (*automatic garbage collector* ou *garbage collector*).

Desta forma não é necessário liberar memória explicitamente nos programas em Java, libertando os programadores desta tarefa usualmente complexa e problemática. Para eliminar um objeto em Java basta perder sua referência o que pode ser feito através de uma nova inicialização da variável objeto que mantêm a referência ou atribuindo-se o valor *null* para tal variável.

O mesmo código C++ exemplificado anteriormente funciona perfeitamente em Java pois ao perder-se a referência do primeiro objeto este se torna alvo do coletor de lixo (alvo do **gc**) tendo a memória associada marcada e recuperada assim que possível, de forma automática e transparente pela máquina virtual Java.

```
Lampada l;         // variável objeto
l = new Lampada(); // l recebe a referencia de um novo objeto
l = new Lampada(); // l recebe uma outra referencia de um objeto
:                  // a referência anterior será eliminada pelo
gc
l = null;          // a referência anterior será eliminada pelo
gc
```

Exemplo 27 Exemplos de Destruição de um Objeto em Java

A presença da coleta automática de lixo no Java torna o conceito de destrutores um pouco diferente de seus equivalentes em outras linguagens orientadas à objetos. Para todos os objetos existem destrutores *default* tais como os construtores *default*. Como o conceito e propósito destes destrutores são ligeiramente diferentes em Java, tais destrutores são denominados finalizadores (*finalizers*). Estes métodos especiais são acionados pelo sistema quando um objeto perdido é efetivamente selecionado para destruição.

Assim como para os construtores, é possível explicitamente criar-se os finalizadores para realização de tarefas mais sofisticadas numa estrutura como a abaixo:

```
protected void finalize() {
  // codigo para “preparar a casa” antes da efetiva destruição
do objeto
}
```

Exemplo 28 Estrutura de um Finalizador (*finalizer*)

Os finalizadores não possuem argumento e não tem valor de retorno (*void*) podendo ser explicitamente acionados, embora isto não garanta que o objeto será imediatamente eliminado, mas apenas marcado para eliminação futura.

A coleta de lixo pode ser forçada programaticamente, isto é, pode ser acionada por meio de programas, através de uma chamada explicita a um método estático existente na classe **java.lang.System**:

```
System.gc(); //ativa a coleta automática de lixo
```

É importante ressaltar que um objeto não é imediatamente destruído ao sair do escopo do programa. O coletor de lixo apenas efetua uma marcação para eliminação futura, isto é, marca o objeto fora de escopo para ser eliminado quando se torna necessário recuperar memória para o sistema, evidenciando o caráter assíncrono da coleta de lixo.

Outra forma de forçar a destruição efetiva dos objetos é desativar o funcionamento assíncrono da coleta de lixo, ou seja, indicar para a máquina virtual que o objeto deve ser destruído assim que sair do escopo. Isto pode ser feito invocando-se a JVM como abaixo:

```
java –noasyncgc NomeDaClasse
```

Caso seja desejado, podemos avisar a JVM que uma mensagem deve ser impressa cada vez que a coleta de lixo ocorrer:

```
java –verbosegc NomeDaClasse
```

As opções acima podem ser combinadas juntamente com outras possíveis para controle da máquina virtual. Devido a pequenas diferenças de implementação das máquinas virtuais de plataformas distintas, sugere-se um teste cuidadoso de aplicações que pretendam utilizar no limite a memória disponível no sistema.

O programa abaixo permite a criação de uma quantidade arbitrária de objetos do tipo lâmpada, onde cada nova instanciação faz sair do escopo a instância anterior. O primeiro argumento do programa obrigatoriamente deve especificar a quantidade de objetos a serem instanciados, um segundo argumento indica que uma chamada ao coletor de lixo deve ser realizada após cada nova instanciação:

```
// GCTest.java

public class GCTest {

  public static void main(String args[]) {
    Lampada l;
    System.out.println("Iniciando Laço");
    for (int i=0; i<Integer.parseInt(args[0]); i++) {
      System.out.print(".");
      l = new Lampada();
      if (args.length > 1)
        System.gc();
    }
    System.out.println("Fim do Laço");
  }
}
```

Exemplo 29 Teste da Coleta de Lixo

Teste o programa para a criação de 12000 sem acionamento do coletor de lixo:

```
java –verbosegc GCTest 12000
```

Teste agora para a criação de 3 objetos forçando o acionamento do coletor de lixo:

```
java –verbosegc GCTest 3 x
```

Com isto aprendemos como funciona a coleta de lixa e como ativá-la quando necessário.

3.9 Dicas para o Projeto de Classes

Embora nosso objetivo seja cobrir o mais extensivamente possível os aspectos de programação através da linguagem Java, é importante destacarmos alguns pontos que devem ser observados na construção de novas classes, isto é, na modelagem e projeto das classes que comporão um projeto.

1. Mantenha os dados da classe como privados

   Dados privados seguem a risca o conceito de encapsulamento pois são inacessíveis aos usuários da classe que não podem desta forma utilizar valores inconsistentes. Assim são necessários métodos de acesso, isto é, métodos que permitam ler o conteúdo destes atributos privados (*acessor methods*) e métodos que permitam modificar o conteúdo destes atributos (*mutator methods*) que podem garantir a existência de apenas operações de leitura ou escrita ou ambas, sempre de forma consistente.

   ```
   // atributo
   private double frequencia;
   ```

```
// método de acesso (acessor method)
public double getFrequencia() {
  return frequencia;
}

// método de alteração (mutator method)
public void setFrequencia(double f) {
  if (f>=0) {
    frequencia = f;
  }
}
```

É prática comum em Java denominar o *acessor method* de um certo atributo **NomeDoAtributo** de **getNomeDoAtributo** e seu *mutator method* correspondente de **setNomeDoAtributo**. Note que o uso dos termos **set** e **get** é apenas uma conveniência em termos de padronização do código.

2. Se os dados não podem ser privados faça-os protegidos

   Se os métodos de acesso a certos atributos necessitarem de modificações em classes derivadas ou se for necessário utilizar diretamente tais dados em classes derivadas, faça os dados e seus métodos de acesso protegidos (*protected*).

```
// atributo
protected int valores[];

// metodo de acesso (acessor method)
protected int getValor(int index) {
  return valores[index];
}

// metodo de alteração (mutator method)
protected void setValor(int index, int value) {
  valores[index]= value;
}
```

3. Sempre inicialize atributos e variáveis

   Embora o Java inicialize as variáveis membro de uma classe, o mesmo não acontece com variáveis locais (declaradas dentro um método ou bloco de código). Assim recomenda-se a inicialização explícita de todas as variáveis nos seus respectivos blocos, métodos ou construtores. Este procedimento torna o código mais claro e não conta com facilidades especiais da máquina virtual Java.

```
public class Exemplo {
  // atributos
  private int a;
  public double b;

  // construtor
  public Exemplo() {
    a = 0;
    b = 0;
  }
}
```

4. Evite o uso de muitos tipos básicos numa mesma classe

   Quando ocorre a declaração de vários atributos de tipos básicos, se relacionados deve ser analisada a possibilidade de transformar tais atributos numa classe em separado. Campos como os que seguem:

```
float latitude;
float longitude;
```

São em verdade dados associados e que podem ser substituídos por uma classe, por exemplo, **PontoGeografico**:

```
public class PontoGeografico {
  private float latitude;
  private float longitude;
  public float getLatitude() {
    return latitude;
  }
  public void setLatitude(float l) {
    latitude = l;
  }
  public float getLongitude() {
    return longitude;
  }
  public void setLongitude (float l) {
    longitude = l;
  }
}
```

Assim a declaração inicial dos campos latitude e longitude pode ser trocada por:

```
PontoGeografico pg;
```

Desta forma as idéias contidas nos atributos individualmente declarados são preservadas e podem ser utilizadas por outras classes. Além disso torna-se mais fácil modificar-se esta estrutura ou adicionar-se novas regras de tratamento.

5. Nem todos os atributos necessitam de métodos de acesso

   Mesmo sendo privados, nem todos os atributos necessitam de métodos de acesso para leitura e escrita de dados. Avalie caso a caso e implemente apenas os métodos de acesso necessário. Se os dados só são modificados na criação do objeto, crie construtores parametrizados para realizar tal tarefa. Isto reduz a quantidade de código associada a uma classe.
6. Padronize a estrutura de suas classes

   Recomenda-se que as classes sejam escritas da mesma forma, ou seja, que seja utilizado um estilo consistente de escrita de programas para todas as classes, incluindo-se nisso o formato dos comentários e a distribuição destes dentro do código, bem como o posicionamento da declaração de atributos e de métodos.

   A declaração dos atributos pode ser colocada no início ou fim da classe. Os métodos podem ser ordenados alfabeticamente ou, como é mais útil, divididos em categorias: primeiro os públicos, depois os protegidos e finalmente os privados.
7. Divida classes com muitas responsabilidades

   Se uma classe possui muitos atributos ou métodos, estude sempre a possibilidade de dividir esta classe em duas ou três, tornando cada uma destas mais simples e ao mesmo tempo mais especializada.
8. Utilize nomes significativos na denominação de classes, métodos e atributos

   Ao escolher um nome para uma classe, método ou atributo, procure um que represente o mais precisamente possível o propósito deste elemento. Evite nomes sem significado ou abreviaturas não usuais. As classes e atributos devem utilizar substantivos como nomes simples ou substantivos e adjetivos como nomes duplos. Métodos devem dar preferência para verbos no infinitivo ou substantivos e verbos no infinitivo.

Estas dicas não são as únicas que poderiam ser dadas nem tão pouco definitivas, mas podem auxiliá-lo num bom projeto e redação de suas classes. Utilizando os comentários padronizados propostos pela ferramenta javadoc, contida no **JDK**, pode-se produzir com relativa facilidade uma documentação simples mas bastante consistente e de fácil manutenção. Quando possível faça diagramas OMT (*Object Modeling Technique*) ou UML (*Unified Modeling Language*) para suas classes mantendo-os junto com a documentação das mesmas.

4 Aplicações de Console

Através da linguagem Java é possível construirmos aplicações de console, isto é, aplicações que utilizam os consoles de operação dos sistemas operacionais para a interação com o usuário, ou seja, para a realização das operações de entrada de dados fornecidos pelo usuário e a exibição de mensagens do programa.

Como console de operação ou console entendemos as seções ou janelas em modo texto (que capazes de exibir apenas texto) de sistemas *Unix*, as janelas "*Prompt* do MS-DOS" existentes no *Windows* e outras semelhantes existentes nos demais sistemas operacionais.

As aplicações de console, embora em desuso para a construção de sistemas mais sofisticados, podem ser utilizadas para a criação de programas utilitários do sistema, tal como certos comandos do sistema operacional que na verdade são programas específicos disponíveis apenas para utilização nos consoles.

4.1 Estrutura das Aplicações de Console

Para construir-se aplicações de console basta uma classe onde seja definido o método estático *main*. Este método é o primeiro a ser executado numa aplicação, daí ser estático, ou seja, não pode depender de ninguém mais para ser instanciado, existindo por completo dentro da classe, como uma espécie de objeto permanente.

Sendo um método estático, *main* não pode acessar variáveis-membro que não sejam estáticas, embora possa internamente instanciar objetos de classes acessíveis. Assim, usualmente no método *main* ocorre a declaração e a instanciação dos objetos que serão utilizados pela aplicação de console. Abaixo temos a estrutura básica de uma aplicação de console:

```
// Classe Pública contendo método main
public class Aplicacao {

  // declaração do método main
  public static void main (String args[]) {
  // código correspondente ao main
  }
}

// Outras classes no mesmo arquivo
```

Exemplo 30 Estrutura de Uma Aplicação de Console

O método *main* também deve ser obrigatoriamente público, não retornando valores e recebendo um vetor de **java.lang.String** como argumento. Em algumas circunstâncias, o método *main* instancia objetos de sua própria classe como veremos em exemplos mais a frente.

Criemos então uma aplicação simples capaz de gerar uma certa quantidade de números aleatórios sendo esta quantidade especificada pelo usuário através de um argumento passado ao programa. Para isto devemos usar o método gerador de números aleatórios (*random*) disponível na classe **java.Math** sob o nome **random**.

```
// RandomNumbers.java

public class RandomNumbers {

  public static void main (String args[]) {
    for (int i=0; i<Integer.parseInt(args[0]); i++) {
      System.out.println("" + Math.random());
    }
  }
}
```

Exemplo 31 Aplicação RandomNumbers

Este programa, quando executado, exibe tantos números aleatórios no intervalo [0, 1] quantos solicitados através do primeiro argumento passado ao programa. Uma pequena modificação adicional pode fazer com que os número produzidos estejam dentro do intervalo [0, n], onde n também pode ser fornecido como um segundo argumento.

```
// RandomInts.java

public class RandomInts {

  public static void main (String args[]) {
    int multiplicador = 1;
    if (args.length == 2) {
      multiplicador = Integer.parseInt(args[1]);
      for (int i=0; i<Integer.parseInt(args[0]); i++) {
        System.out.println("" + Math.round(
                                Math.random()*multiplicador));
      }
    } else {
      for (int i=0; i<Integer.parseInt(args[0]); i++) {
        System.out.println("" + Math.random() );
      }
    }
  }
}
```

Exemplo 32 Aplicação RandomInts

Outras modificações desejáveis para esta aplicação seriam: (i) o controle de erros provocados por argumentos inválidos ou pelo fornecimento de um número insuficiente de argumentos com a consequente exibição de mensagens de erro.; (ii) a geração de números aleatórios dentro de um intervalo [m, n]; (iii) melhor apresentação dos resultados de saída pois apenas os 24 últimos números solicitados podem ser visualizados; etc.

4.2 Saída de Dados

As aplicações de console utilizam como padrão a *stream* de dados **out**, disponível estaticamente através da classe **java.lang.System**. Uma *stream* pode ser entendida como um duto capaz de transportar dados de um lugar (um arquivo ou dispositivo) para um outro lugar diferente. O conceito de *stream* é extremamente importante pois é utilizado tanto para manipulação de dados existentes em arquivos como também para comunicação em rede e outros dispositivos.

A *stream* de saída padrão é aberta automaticamente pela máquina virtual Java ao iniciarmos uma aplicação Java e permanece pronta para enviar dados. No caso a saída padrão está tipicamente associada ao dispositivo de saída (*display*) ou seja, a janela de console utilizada pela aplicação conforme designado pelo sistema operacional.

Enquanto a *stream* de saída **out** é um objeto da classe **java.io.PrintStream** a *stream* de entrada **in** é um objeto da classe **java.io.InputStream**.

A *stream* de saída **out** é um objeto da classe **java.io.PrintStream**, que pode ser utilizada através dos seguintes métodos:

Método	Descrição
print(char)	Imprime um caractere.
print(boolean)	Imprime um valor lógico.
print(int)	Imprime um inteiro.
print(long)	Imprime um inteiro longo.
print(float)	Imprime um valor em ponto flutuante simples.
print(double)	Imprime um valor *double*.
print(String)	Imprime uma *string*.
println()	Imprime o finalizador de linha.
println(char)	Imprime um caractere e o finalizador de linha.
println(boolean)	Imprime um valor lógico e o finalizador de linha.
println(int)	Imprime um inteiro e o finalizador de linha.
println(long)	Imprime um inteiro longo e o finalizador de linha.
println(float)	Imprime um valor em ponto flutuante simples e o finalizador de linha.
println(double)	Imprime um valor *double* e o finalizador de linha.
println(String)	Imprime uma *string* e o finalizador de linha.

Tabela 11 Métodos do Objeto java.io.PrintStream

Como foi feito em vários dos exemplos anteriores para enviarmos dados para a saída padrão devemos acionar utilizar a construção:

```
System.out.print(valor);
System.out.println(valor);
```

A saída padrão pode ser redirecionada para outro dispositivo, ou seja, podemos especificar um arquivo ou um canal de comunicação em rede para receber os dados que seriam enviados ao console. Para isto pode-se utilizar o método estático **setOut** da classe **java.lang.System**:

```
System.setOut(novaStreamDeSaida);
```

O redirecionamento da saída padrão pode ser útil quando queremos que um programa produza um arquivo de saída ou envie sua saída para um outro dispositivo.

4.3 Entrada de Dados

Da mesma forma que toda aplicação de console possui uma *stream* associada para ser utilizada como saída padrão, existe uma outra *stream* denominada de entrada padrão, usualmente associada ao teclado do sistema. Esta *stream*, de nome **in**, disponível estaticamente através da classe **java.lang.System** e pertencente a classe **java.io.InputStream** é também aberta automaticamente quando a aplicação é iniciada pela máquina virtual Java permanecendo pronta para fornecer os dados digitados.

Devemos observar que embora pronta para receber dados dos usuários, os métodos disponíveis para a entrada destes dados é bastante precária. Veja na Tabela 12 os principais métodos disponíveis na classe **java.io.InputStream**.

Método	Descrição
available()	Retorna a quantidade de bytes que podem ser lidos sem bloqueio.
read()	Lê um byte.
read(byte[])	Preenche o vetor de bytes fornecido.
skip(long)	Descarta a quantidade de bytes especificada da entrada.

Tabela 12 Métodos da Classe java.io.InputStream

Como podemos observar a entrada de dados através de objetos **InputStream** é orientada a byte, isto é, as operações de leitura realizam a entrada de bytes, o que equivale a leitura de caracteres simples. Isto é pouco confortável quando se pretende ler valores numéricos (inteiros ou em ponto flutuante), *strings* ou qualquer outra informação diferente de caracteres pois exige que os vários caracteres fornecidos pelo usuário sejam concatenados e testados para a formação de um valor ou *string*.

A seguir um exemplo de aplicação capaz de efetuar a leitura de caracteres isolados entendidos como conceitos de A até E, equivalentes a notas de 5 a 1 respectivamente:

```
// Media.java

public class Media {

  public static void main(String args[])
                        throws java.io.IOException {
    float notaMedia = 0;
    int contagem = 0;
    int valor;
    System.out.println("Forneca como notas conceitos A - E.");
    System.out.println("Um traco '-' encerra a entrada de
notas.");
    System.out.println("Digite um valor: ");
    while ('-' != (valor = System.in.read())) {
      notaMedia += 'F' - valor;
      contagem++;
      System.in.skip(2);
      System.out.println("Digite um valor: ");
    }
    notaMedia /= contagem;
    System.out.println("Foram fornecidos " +
                    contagem + " valores.");
    System.out.println("Nota media     =  " + notaMedia);
    System.out.println("Conceito medio =  " +
                    (char) (70 - Math.round(notaMedia)));
  }
}
```

Exemplo 33 Aplicação Média

Nesta aplicação existem vários detalhes que devem ser comentados:

- O método **read**, da *stream* de entrada padrão **in**, quando não consegue tratar a entrada fornecida (a digitação de dois caracteres por exemplo), lança a exceção **java.io.Exception** mas como optamos por não tratá-la via *try catch*, o método **main** também lança esta exceção.
- Cada caractere lido é na verdade um byte, que pode ser tratado numericamente. Os caracteres maiúsculos A, B, C ... são representados pelos códigos ASCII 65, 66, 67, etc. Alguma manipulação algébrica transforma tais códigos em valores de 5 a 1:
  ```
  notaMedia += 'F' - valor;
  ```
  O inverso também é possível:
  ```
  (char) (70 - Math.round(notaMedia))
  ```
- O “enter” pressionado após a digitação de cada caractere deve ser extraído da *stream* de entrada. Como equivale a dois caracteres (uma sequência de “\r\n”) usamos o método **skip**.
- Em dois pontos do código utilizamos o operador composto “+=” e “/=” que significa:
  ```
  variável op= expressão
  variável = variável op expressão
  ```

A aplicação não trata quaisquer tipos de erros, nem sequer o uso de caracteres minúsculos ao invés de maiúsculos (uma boa sugestão para um exercício adicional).

A dificuldade de tratar a entrada de dados em aplicações de console é uma clara indicação de que os projetistas do Java concentraram seus esforços para tornar esta plataforma mais adaptada para o desenvolvimento de aplicações gráficas destinadas as GUIs (*Graphical User Interfaces*) dos diversos sistemas operacionais existentes.

Para contornar os problemas de leitura de valores inteiros, em ponto flutuante e também *strings*, apresentamos a classe **Console** que contém três métodos para leitura de valores destes tipos.

Método	Descrição
readDouble()	Lê um valor *double* da entrada padrão.
readInteger()	Lê um valor inteiro da entrada padrão.
readString()	Lê uma *string* da entrada padrão.

Tabela 13 Métodos da Classe Console

Abaixo o código da classe Console onde foi utilizado a classe **java.io.BufferedReader** para prover a facilidade da leitura de uma linha da entrada de cada vez. Todos os métodos oferecidos fazem um tratamento mínimo de erros de forma que valores inválidos são retornados como valores nulos.

```
// Console.java

import java.io.*;

public final class Console {

  public static double readDouble() {
    try {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      String s = br.readLine();
      Double d = new Double(s);
      return d.doubleValue();
    } catch (IOException e) {
      return 0;
    } catch (NumberFormatException e) {
      return 0;
    }
  }

  public static int readInteger() {
    try {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      String s = br.readLine();
      return Integer.parseInt(s);
    } catch (IOException e) {
      return 0;
    } catch (NumberFormatException e) {
      return 0;
    }
  }

  public static String readString() {
    try {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      String s = br.readLine();
      return s;
```

```
    } catch (IOException e) {
      return "";
    }
  }
}
```

Exemplo 34 Classe Console

Como todos os métodos da classe Console são estáticos, não é necessário instanciar-se um objeto para fazer uso destes métodos, como é ilustrado no exemplo a seguir, que lê um número real, um inteiro e uma *string* mas de forma bastante robusta.

```
// ConsoleTeste.java

public class ConsoleTest {

  public static void main(String args[]){
    System.out.print("Forneca um numero real: ");
    double x = Console.readDouble();
    System.out.println("Numero Fornecido: " + x);

    System.out.print("Forneca um numero inteiro: ");
    int y = Console.readInteger();
    System.out.println("Numero Fornecido: " + y);

    System.out.print("Forneca uma string: ");
    String s = Console.readString();
    System.out.println("String Fornecida: " + s);
  }
}
```

Exemplo 35 Aplicação da Classe Console

4.4 Uma Aplicação Mais Interessante

Apesar das aplicações de console não terem o apelo atrativo das interfaces gráficas, ainda assim é possível construir-se aplicações interessantes. Propõe-se como um exemplo mais longo um jogo de cartas popular: o “Vinte e Um”.

O projeto do jogo segue a lógica comum: para jogar-se “Vinte e Um” é necessário conhecer-se as regras do jogo que são simples e dispor-se de um baralho comum. As regras do jogo são as seguintes:

(i) Utilizam-se as 52 cartas do baralho comum (de ás ao rei de cada um dos quatro naipes embaralhadas.

(ii) Jogam dois adversários de cada vez: o jogador (usuário) e o *croupier* (computador).

(iii) O objetivo é fazer 21 pontos ou pelo menos mais pontos que o adversário embora que obter mais de 21 pontos (estourar) perde imediatamente. Em caso de empate o *croupier* vence.

(iv) O ás vale 1 ponto, dois vale 2 pontos e assim por diante até o dez. Valete, dama e rei valem apenas 1 ponto.

(v) O jogador sempre começa recebendo uma carta que é exibida para todos. O jogador pode solicitar tantas cartas quanto desejar.

(vi) Se o jogador parar com 21 pontos ou menos joga o *croupier* até que obtenha pontuação igual ou superior ao jogador, perdendo apenas se estourar.

Para implementar-se o jogo podemos utilizar a seguinte estratégia: criamos um objeto **Carta** para representar individualmente as cartas de um baralho; criamos um objeto **Baralho** que contêm 52 cartas e depois uma classe **VinteUm** que implemente as regras do jogo “Vinte e Um” utilizando os objetos pré-definidos.

A classe **Carta** deve definir um objeto capaz de armazenar duas informações: o número da carta e seu naipe, ambas podendo ser representadas por valores inteiros. Como não existe carta sem naipe e vice-versa, o construtor deve receber dois valores inteiros um para cada informação. A classe poderia ainda oferecer métodos para obter-se o número (valor) da carta e seu naipe, bem como um método para descrever a carta (ás de paus, quatro de ouros, etc.). A classe também poderia conter a definição de algumas constantes para os naipes e também valores das cartas.

Esta classe poderia ser utilizada para a criação de uma classe **Baralho** ou outra que necessitasse manusear cartas. Segue no Exemplo 36 o código sugerido para implementação da classe **Carta**.

```
// Carta.java

public class Carta {
  public static final int PAUS = 1;
  public static final int OUROS = 2;
  public static final int ESPADAS = 3;
  public static final int COPAS = 4;

  private int naipe;
  private int valor;

  public Carta(int n, int v) {
    if (n >= PAUS && n <= COPAS)
      naipe = n;
    else
      naipe = 1;
    if (v >= 1 && n <= 13)
      valor = v;
    else
      valor = 1;
  }

  public int getNaipe() {
    return naipe;
  }

  public int getValor() {
    return valor;
  }

  public String toString() {
    String tmp = new String("");
    switch(valor) {
      case 1:  tmp = "as";     break;
      case 2:  tmp = "dois";   break;
      case 3:  tmp = "tres";   break;
      case 4:  tmp = "quatro"; break;
      case 5:  tmp = "cinco";  break;
      case 6:  tmp = "seis";   break;
      case 7:  tmp = "sete";   break;
      case 8:  tmp = "oito";   break;
      case 9:  tmp = "nove";   break;
      case 10: tmp = "dez";    break;
      case 11: tmp = "valete"; break;
      case 12: tmp = "dama";   break;
      case 13: tmp = "rei";    break;
    }
    switch(naipe) {
      case PAUS:    tmp = tmp + " de paus";    break;
```

```
      case OUROS:   tmp = tmp + " de ouros";   break;
      case ESPADAS: tmp = tmp + " de espadas"; break;
      case COPAS:   tmp = tmp + " de copas";   break;
    }
    return tmp;
  }
}
```

Exemplo 36 Classe Carta

Como um baralho comum possui 52 cartas, uma classe **Baralho** deveria armazenar 52 cartas. O construtor desta classe deve garantir que existam as 13 cartas de cada naipe, oferecendo um baralho ordenado. Um método **embaralhar** poderia misturar as cartas e outro **darCarta** poderia retirar uma carta do monte. Finalmente um outro método poderia verificar se ainda existem cartas no monte, tal como **temCarta**. Estas operações básicas servem para a maioria dos jogos de cartas, ficando a cargo do usuário destas classes implementar outras classes especificas ou o jogo diretamente. Segue o código proposto para a classe **Baralho**.

```
// Baralho.java

public class Baralho {
  private Carta monte[];
  private int topo;

  public Baralho() {
    monte = new Carta[52];
    topo = 0;
    for (int n=1; n<5; n++)
      for (int v=1; v<14; v++) {
        monte[topo++] = new Carta(n, v);
      }
  }

  public boolean temCarta() {
    return topo > 0;
  }

  public Carta darCarta() {
    Carta tmp = null;
    if (topo > 0)
      tmp = monte[--topo];
    return tmp;
  }

  public void embaralhar() {
    for(int c=0; c<52; c++) {
      int i = (int) Math.round(Math.random()*51);
      Carta tmp = monte[i];
      monte[i] = monte[c];
      monte[c] = tmp;
    }
  }
}
```

Exemplo 37 Classe Baralho

Finalmente podemos construir o jogo “Vinte e Um”. Para tanto necessitamos de uma outra classe denominada **VinteUm** que definirá um baralho e variáveis para armazenar as cartas e a pontuação de cada um dos jogadores. Praticamente todo código fica colocado no método **main**, com exceção de um chamado **mostraCartas** que imprime na tela as cartas que o jogador possui e soma os pontos obtidos com estas. Observe atentamente a implementação sugerida.

```
// VinteUm.java

public class VinteUm {

  public static void main(String args[]) {
    Baralho b = new Baralho();
    b.embaralhar();
    int jogador = 0;
    int pontosJogador = 0;
    int comput = 0;
    int pontosComput = 0;
    Carta cartasJogador[] = new Carta[20];
    Carta cartasComput[]  = new Carta[20];
    String resp = "N";
    do {
      cartasJogador[jogador++] = b.darCarta();
      System.out.println("\nSuas cartas:");
      pontosJogador = mostraCartas(cartasJogador, jogador);
      System.out.println("Seus pontos = " + pontosJogador);
      if (pontosJogador < 21) {
        System.out.println("Quer carta? (S|N)");
        resp = Console.readString();
      }
    } while (resp.equals("S") && pontosJogador < 21);
    if (pontosJogador > 21){
      System.out.println("Voce perdeu!");
      return;
    }
    while (pontosComput < pontosJogador && pontosComput != 21) {
      cartasComput[comput++] = b.darCarta();
      System.out.println("\nMinhas cartas:");
      pontosComput = mostraCartas(cartasComput, comput);
      System.out.println("Meus pontos = " + pontosComput);
    }
    if (pontosComput >= pontosJogador && pontosComput <= 21){
      System.out.println("\nVoce perdeu!");
    } else {
      System.out.println("\nVoce ganhou!");
    }
  }

  private static int mostraCartas(Carta mao[], int quant) {
    int pontos = 0;
    for (int i=0; i<quant; i++) {
      System.out.print("  " + mao[i].toString());
      if (mao[i].getValor() > 10)
        pontos++;
      else
        pontos += mao[i].getValor();
    }
    System.out.println();
    return pontos;
  }
}
```

Exemplo 38 Classe VinteUm

A seguir uma ilustração do diagrama de classe obtido com o código proposto para a implementação das classes do jogo “Vinte e Um”.

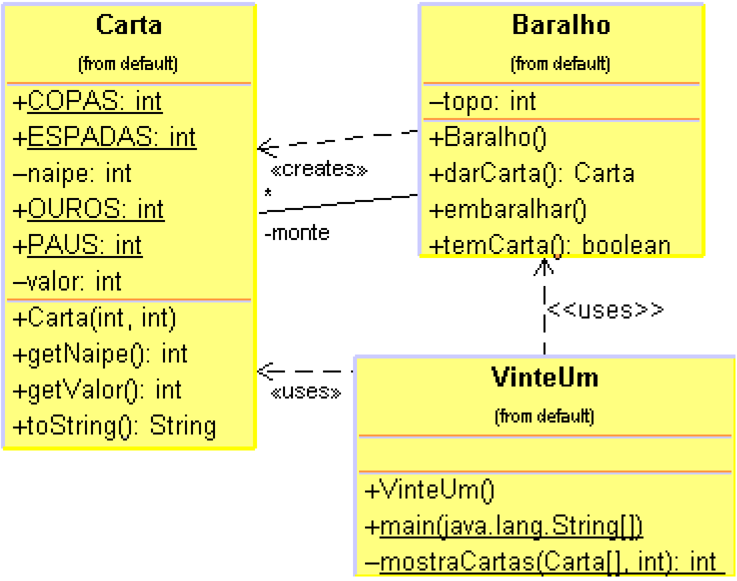


Figura 17 Diagrama de Classe do VinteUm

Experimente agora executar a aplicação e jogar uma partida do jogo.

5 Sobrecarga, Herança e Polimorfismo

A construção de classes, embora de fundamental importância, não representa o mecanismo mais importante da orientação à objetos. A grande contribuição da orientação à objetos para o projeto e desenvolvimento de sistemas é o polimorfismo. A palavra polimorfismo vem do grego *poli morfos* e significa muitas formas. Na orientação à objetos representa uma característica onde se admite tratamento idêntico para formas diferentes baseado em relações de semelhança, isto é entidades diferentes podem ser tratadas de forma semelhante conferindo grande versatilidade aos programas e classes que se beneficiam destas características.

5.1 Sobrecarga de Métodos

A forma mais simples de polimorfismo oferecido pela linguagem Java é a sobrecarga de métodos (*method overload*) ou seja é a possibilidade de existirem numa mesma classe vários métodos com o mesmo nome. Para que estes métodos de mesmo nome possam ser distinguidos eles devem possuir uma assinatura diferente. A assinatura (*signature*) de um método é uma lista que indica os tipos de todos os seus argumentos, sendo assim métodos com mesmo nome são considerados diferentes se recebem um diferente número ou tipo de argumentos e tem, portanto, uma assinatura diferente. Um método que não recebe argumentos tem como assinatura o tipo *void* enquanto um outro método que recebe dois inteiros como argumentos tem como assinatura os tipos *int*, *int* como no exemplo abaixo:

```
// Sobrecarga.java

public class Sobrecarga {

  public long twice (int x) {
    return 2*(long)x;
  }

  public long twice (long x) {
    return 2*x;
  }

  public long twice (String x) {
    return 2*(long)Integer.parseInt(x);
  }
}
```

Exemplo 39 Sobrecarga de Métodos

No exemplo temos na classe **Sobrecarga** a implementação de três métodos denominados **twice** que tomam um único argumento retornando um valor que é o dobro do valor do argumento recebido. Através da sobrecarga foram implementadas três versões do método **twice**, cada uma capaz de receber um argumento de tipo diferente. Na prática é como se o método **twice** fosse capaz de processar argumentos de tipos diferentes, o que para os usuários da classe **Sobrecarga**.

Outra situação possível é a implementação de métodos com mesmo nome e diferentes listas de argumentos, possibilitando uma utilização mais flexível das idéias centrais contidas nestes métodos como a seguir:

```
// Sobrecarga2.java

public class Sobrecarga2 {
  public long somatorio (int max) {
    int total=0;
    for (int i=1; i<=max; i++)
      total += i;
    return total;
  }
  public long somatorio (int max, int incr) {
    int total=0;
    for (int i=1; i<=x; i += incr)
      total += i;
    return total;
  }
}
```

Exemplo 40 Sobrecarga de Métodos

No Exemplo 40 temos o método **somatorio** efetua a soma dos primeiros **max** números naturais ou efetua a soma dos primeiro **max** números naturais espaçados de **incr** e que a seleção de um método ou outro se dá pelo diferente número de argumentos utilizados pelo método **somatorio**.

A API do Java utiliza intensivamente o mecanismo de sobrecarga, por exemplo a classe **java.lang.String** onde o método **indexOf** possui várias implementações. O método **indexOf** se destina a localizar algo dentro de uma *string*, isto é, a determinar a posição (índice) de um caractere ou *substring*, o que caracteriza diferentes possibilidades para seu uso, conforme a tabela a seguir:

Método	Descrição
indexOf(int)	Retorna a posição, dentro da *string*, da primeira ocorrência do caractere especificado.
indexOf(int, int)	Retorna a posição, dentro da *string*, da primeira ocorrência do caractere especificado a partir da posição dada.
indexOf(String)	Retorna a posição, dentro da *string*, da primeira ocorrência da *substring* especificada.
indexOf(String, int)	Retorna a posição, dentro da *string*, da primeira ocorrência da *substring* especificada a partir da posição dada.

Figura 18 Sobrecarga do Método indexOf da Classe java.lang.String

Desta forma, fica transparente para o usuário da classe a existência destes quatro métodos ao mesmo tempo que disponíveis tais alternativas para localização de caracteres simples ou *substrings* dentro de *strings*.

Numa classe onde são implementados vários métodos com o mesmo nome fica a cargo do compilador determinar qual o método apropriado em função da lista de argumento indicada pelo método efetivamente utilizado (*dinamic binding*). O valor de retorno não faz parte da assinatura pois admite-se a conversão automática de tipos impedindo o compilador de identificar o método adequando.

Um outro ponto importante é que a linguagem Java não fornece recursos para sobrecarga de operadores tal como existe na linguagem C++. Apesar disto ser um aspecto restritivo da linguagem, é perfeitamente condizente com a filosofia que orientou o desenvolvimento desta linguagem que foi criada para ser pequena, simples, segura de se programar e de se usar. A ausência de mecanismos para definir a sobrecarga de operadores pode ser contornada através da definição e implementação apropriada de classes e métodos.

5.2 Sobrecarga de Construtores

Da mesma forma que podemos sobrecarregar métodos de uma classe, o mesmo pode ser feito com seus construtores, ou seja, uma classe pode possuir mais de um construtor onde cada um deles é diferenciado por sua assinatura. O compilador determina qual dos construtores utilizar através dos argumentos utilizados na chamada do construtor. Obviamente deve existir um construtor cuja lista de argumentos seja a utilizada na criação de um novo objeto, caso contrário será indicado o erro:

```
C:\exemplos\Capítulo4>javac SobrecargaTest.java
SobrecargaTest.java:4: Wrong number of arguments in constructor.
    Sobrecarga s = new Sobrecarga(false);
                   ^
```

Java oferece um construtor *default* para cada classe o qual não precisa ser declarado e implementado. Este construtor, que não recebe parâmetros, mesmo quando implementado é denominado construtor *default*. Quando sobrecarregamos o construtor de uma classe, tal como faríamos com qualquer outro método, criamos construtores denominados construtores cópia (*copy constructors*). A seguir um exemplo de uma classe contendo três diferentes construtores (o construtor *default* e dois construtores cópia):

```
// Sobrecarga3.java

public class Sobrecarga3 {
  // Atributos
  public int prim;
  public int sec;

  // Construtores
  public Sobrecarga3() {
    prim = sec = 0;
  }
  public Sobrecarga3(int p) {
    prim = p;
    sec = 0;
  }
  public Sobrecarga3(int p, int s) {
    prim = p;
    sec = s;
  }
}
```

Exemplo 41 Sobrecarga de Construtores

A classe **Sobrecarga3** possui dois atributos inteiros **prim** e **sec** que são inicializados com valor zero através do construtor default. Para que o atributo **prim** seja inicializado com outro valor podemos utilizar o construtor cópia que recebe um parâmetro inteiro. Para que tanto o atributo **prim** com **sec** seja inicializados diferentemente dispõe-se ainda de um segundo construtor cópia.

O uso de construtores sobrecarregados é bastante conveniente pois possibilita reduzir o código que deve ser escrito para utilização de um objeto como nos trechos de código abaixo:

```
Sobrecarga3 s = new Sobrecarga3();
s.prim = 10;
```

Pode ser substituído por apenas:

```
Sobrecarga3 s = new Sobrecarga3(10);
```

Outro caso possível:

```
Sobrecarga3 s = new Sobrecarga3();
s.prim = 5;
s.sec = -2;
```

Pode ser substituído por:

```
Sobrecarga3 s = new Sobrecarga3(5, -2);
```

A API do Java utiliza extensivamente o mecanismo de sobrecarga de construtores para oferecer aos programadores a maior flexibilidade possível. Um exemplo disto é a classe **java.lang.String** que possui 11 diferentes construtores dos quais destacamos alguns:

Construtor	Descrição
String()	Cria uma nova *string* sem conteúdo.
String(byte[])	Cria uma nova *string* convertendo o conteúdo do vetor de *bytes* especificado conforme a codificação *default* de caracteres.
String(char[])	Cria uma nova *string* convertendo o conteúdo do vetor de *char* especificado.
String(String)	Cria uma nova *string* contendo a mesma sequência de caracteres da *string* especificada.
String(StringBuffer)	Cria uma nova *string* contendo a mesma sequência de caracteres do objeto **StringBuffer** especificado.

Tabela 14 Construtores da Classe java.lang.String

Ao mesmo tempo que a disponibilidade de vários construtores cópia é bastante conveniente, sua implementação em uma certa classe deve ser objeto de análise cuidadosa para determinar-se se esta adição será realmente útil aos usuários da classe evitando-se assim onerar-se desnecessariamente o código produzido.

5.3 Herança

A herança (*inheritance*) é o segundo e mais importante mecanismo do polimorfismo e pode ser entendido de diversas formas das quais a mais simples é: uma técnica onde uma classe passa a utilizar atributos e operações definidas em uma outra classe especificada como seu ancestral. Rigorosamente falando, a herança é o compartilhamento de atributos e operações entre classes baseado num relacionamento hierárquico do tipo pai e filho, ou seja, a classe pai contêm definições que podem ser utilizadas nas classes definidas como filho. A classe pai é o que se denomina classe base (*base class*) ou superclasse (*superclass*) e as classes filho são chamadas de classes derivadas (*derived classes*) ou subclasses (*subclasses*). Este mecanismos sugere que uma classe poderia ser definida em termos mais genéricos ou amplos e depois refinada sucessivamente em uma ou mais subclasses específicas. Daí o origem do termo técnico que descreve a herança: especialização (*specialization*).

Em Java indicamos que uma classe é derivada de uma outra classe utilizando a palavra reservada *extends* conforme o trecho simplificado de código dado a seguir: A superclasse não recebe qualquer indicação especial.

```
// SuperClass.java
public class SuperClass {
  ⋮
}
// SubClass.java
public class SubClass extends SuperClass {
  ⋮
}
```

Exemplo 42 Exemplo Simplificado de Herança

Num diagrama UML teríamos a seguinte representação para classes hierarquicamente relacionadas:

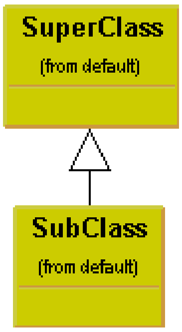


Figura 19 Representação de Herança (Notação UML)

Em princípio, todos atributos e operações definidos para uma certa classe base são aplicáveis para seus descendentes que, por sua vez, não podem omitir ou suprimir tais caraterísticas pois não seriam verdadeiros descendentes se fizessem isto. Por outro lado uma subclasse (um descendente de uma classe base) pode modificar a implementação de alguma operação (reimplementar) por questões de eficiência sem modificar a interface externa da classe. Além disso as subclasses podem adicionar novos métodos e atributos não existentes na classe base, criando uma versão mais específica da classe base, isto é, especializando-a. Na Figura 20 temos um exemplo de hierarquia de classes onde são indicados os atributos e operações adicionados em cada classe. Note que os atributos e operações adicionados em uma dada classe base não são indicados explicitamente na classe derivada mas implicitamente pela relação de herança.

A herança é portanto um mecanismo de especialização pois uma dada subclasse possui tudo aquilo definido pela sua superclasse além de atributos e operações localmente adicionados. Por outro lado a herança oferece um mecanismo para a generalização pois uma instância de uma classe é uma instância de todos os ancestrais desta classe, isto é, um objeto de uma classe derivada pode ser tratado polimorficamente como um objeto da superclasse, como veremos na seção 5.3.

No mundo real alguns objetos e classes podem ser descritos como casos especiais, especializações ou generalizações de outros objetos e classes. Por exemplo, a bola de futebol de salão é uma especialização da classe bola de futebol que por sua vez é uma especialização da classe bola. Aquilo que foi descrito para uma certa classe não precisa ser repetido para uma classe mais especializada originada na primeira.

Atributos e operações definidos na classe base não precisam ser repetidos numa classe derivada, desta forma a orientação à objetos auxilia a reduzir a repetição de código dentro de um programa ao mesmo tempo que possibilita que classes mais genéricas sejam reaproveitadas em outros projetos (reusabilidade de código). Outro aspecto bastante importante é que se o projeto de uma hierarquia de classe é suficientemente genérico e amplo temos o desenvolvimento de uma solução generalizada e com isto torna-se possível mais do que a reutilização de código, mas passa a ser possível o que denomina reutilização do projeto. Uma classe ou hierarquia de classes que pode ser genericamente utilizada em outros projetos é que se chama de padrão de projeto ou *design pattern*.

Enquanto mecanismo de especialização, a herança nos oferece uma relação "é um" entre classes (*"is a" relationship*): uma bola de futebol de salão é uma bola de futebol que por sua vez é uma bola. Uma sentença é uma parte de um parágrafo que é uma parte de um documento. Inúmeros outros exemplos poderiam ser dados. A descoberta de relações do tipo "é um" é a chave para determinarmos quando novas classes devem ser ou não derivadas de outras existentes.

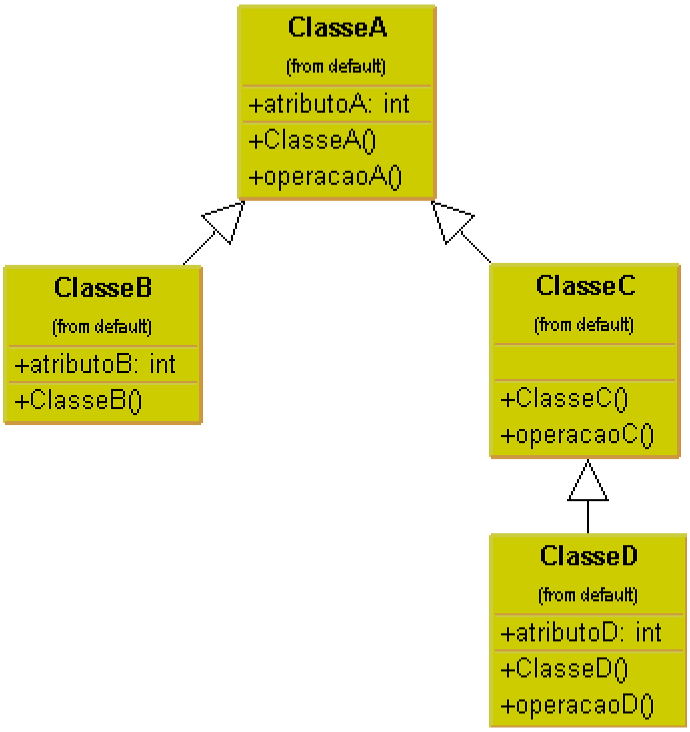


Figura 20 Exemplo de Hierarquia de Classes

Desta forma o mecanismo de especialização pode conduzir a duas situações distintas:

(i) Extensão
A herança pode ser utilizada para a construção de novas classes que ampliam de forma especializada as operações e atributos existentes na classe base. Com isto temos a adição de novos elementos a uma classe.

(ii) Restrição
As subclasses podem ocultar ou alterar consistentemente operações e atributos da superclasse (*overhiding*) sem modificar a interface proposta por estas. Com isto temo a modificação de elementos já existentes numa classe para adequação com os novos elementos adicionados.

O acesso à atributos e operações declarados em uma classe base por parte das suas classes derivadas não é irrestrito. Tal acessibilidade depende dos nível de acesso permitido através dos especificadores *private* (privado), *protected* (protegido), *package* (pacote) e *public* (público) conforme a Tabela 15.

Métodos e Atributos da Classe	Implementação da Classe	Instância da Classe	SubClasse da Classe	Instância da SubClasse
privados (*private*)	sim	não	não	não
protegidos (*protected*)	sim	não	sim	não
pacote (*package*)	sim	sim (no mesmo pacote)	sim (no mesmo pacote)	sim (no mesmo pacote)
públicos (*public*)	sim	sim	sim	sim

Tabela 15 Especificadores de Acesso do Java

Ao especificarmos acesso privado (*private*) indicamos que o atributo ou operação da classe é de uso interno e exclusivo da implementação da própria classe, não pertencendo a sua interface de usuário tão pouco a sua interface de programação. Membros privados não podem ser utilizados externamente as classes que os declaram

O acesso protegido (*protected*) e público (*public*) servem, respectivamente, para definir membros destinado a interface de programação e a interface de usuário. Os membros públicos podem, sem restrição ser utilizados por instâncias da classe, por subclasses e por instâncias das subclasses. Membros protegidos podem ser utilizados apenas na implementação de subclasses, não sendo acessíveis por instâncias da classe ou por instâncias das subclasses.

Finalmente membros que não recebem as especificações *private* (privado), *protected* (protegido) ou *public* (público) são considerados como do caso especial *package* (pacote), isto é, são como membros públicos para classes do mesmo pacote mas comportam-se como membros privados para classes de outros pacotes. Esta modalidade de acesso visa facilitar a construção de conjuntos de classes correlacionadas, isto é, de pacotes. O acesso *package* é o considerado *default* pelo Java.

Desta forma é possível evitar que pequenas alterações propaguem-se provocando grandes efeitos colaterais pois alterações na implementação que não alteram a interface existente do objeto podem ser realizadas sem que seja necessário modificar-se as aplicações deste objeto.

Ao invés de termos criado duas classes **Liquidificador** e **Liquidificador2** para representar liquidificadores com controle de velocidade analógico e digital (Exemplo 18 e Exemplo 20) poderíamos ter criado uma classe base genérica **LiquidificadorGenerico** contendo as características comuns destes aparelhos e duas classes derivadas **LiquidificadorAnalogico** e **LiquidificadorDigital** que implementassem as características particulares conforme ilustrado no diagrama UML da Figura 21.

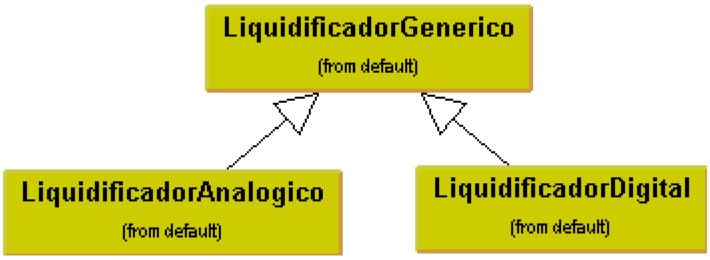


Figura 21 Hierarquia de Classes: Liquidificador

Seguem implementações possíveis para estas classes.

```
// LiquidificadorGenerico.java

public class LiquidificadorGenerico {
  // atributos
  protected int velocidade;
  protected int velocidadeMaxima;

  // construtores
  public LiquidificadorGenerico() {
    velocidade = 0;
    velocidadeMaxima = 2;
  }

  public LiquidificadorGenerico(int v) {
    this()
    ajustarVelocidadeMaxima(v);
  }
```

```
  // metodos
  protected void ajustarVelocidadeMaxima(int v) {
    if (v>0)
      velocidadeMaxima = v;
  }

  protected void ajustarVelocidade(int v) {
    if (v>=0 && v<=velocidadeMaxima)
      velocidade = v;
  }

  public int obterVelocidadeMaxima() {
    return velocidadeMaxima;
  }
}

  public int obterVelocidade() {
    return velocidade;
  }
}
```

Exemplo 40 Classe LiquidificadorGenerico

Note que o atributo **velocidade** passa a ser protegido, isto é, passa a ser acessível apenas na implementação de classes derivadas da classe **LiquidificadorGenerico**. Adicionou-se um novo atributo, também protegido, **velocidadeMáxima** que contêm a máxima velocidade admissível para o liquidificador, cujo valor default 2 é definido no construtor da classe.

Através da sobrecarga implementa-se um construtor cópia que permite definir o valor máximo admissível para as velocidades do objeto.

Os método **ajustarVelocidade** e **ajustarVelocidadeMaxima** também são protegidos sugerindo que nas classes derivadas sejam implementados os métodos de controle de velocidade que serão disponibilizados para os usuários destas classes. Outro ponto importante é que a classe **LiquidificadorGenerico** não limita a quantidade de velocidades possíveis para um dado liquidificador, ficando a cargo das classes derivadas tais restrições e seu controle. Os métodos **obterVelocidade** e **obterVelocidadeMaxima** são públicos e destinam-se a fornecer informações sobre a velocidade atual e a maior velocidade possível do objeto.

Esta nova implementação de uma classe básica para criação de uma hierarquia de classes é uma estratégia bastante mais genérica e flexível pois não limita a forma de controle da velocidade tão pouco o limite máximo da velocidade mas garante o uso de velocidades adequadas, permitindo implementações simplificadas das demais classes destinadas a representação dos liquidificadores analógicos (**LiquidificadorAnalogico**) e digitais (**LiquidificadorDigital**).

```
// LiquidificadorAnalogico.java

public class LiquidificadorAnalogico extends
LiquidificadorGenerico {
  // construtor
  public LiquidificadorAnalogico() {
    velocidade = 0;
  }

  // metodos
  public void aumentarVelocidade() {
    ajustarVelocidade(velocidade + 1);
  }
```

```
  public void diminuirVelocidade() {
    diminuirVelocidade(velocidade - 1);
  }
}
```

Exemplo 44 Classe LiquidificadorAnalogico

A nova classe **LiquidificadorAnalogico** compartilha o atributo **velocidade** e as operações **ajustarVelocidade** e **obterVelocidade** com a classe base **LiquidificadorGenerico**. Nesta classe são implementados apenas um construtor (apenas para evidenciar a inicialização do atributo **velocidade**) e dois métodos para o controle de velocidade do liquidificador: **aumentarVelocidade** e **diminuirVelocidade**. Note que a interface desta nova classe **LiquidificadorAnalogico** é idêntica a interface da classe **Liquidificador** apresentada no Exemplo 18, ou seja, com modificações mínimas o mesmo programa de teste pode ser utilizado (é necessário apenas trocar o nome da classe do objeto testado de **Liquidificador** para **LiquidificadorAnalogico**).

```
// LiquidificadorDigital.java

public class LiquidificadorDigital extends
LiquidificadorGenerico{
  // construtor
  public LiquidificadorDigital() {
    velocidade = 0;
  }
  // metodo
  public void trocarVelocidade(int v) {
    // aciona método protegido para troca de velocidade
    ajustarVelocidade(v);
  }
}
```

Exemplo 45 Classe LiquidificadorDigital

De forma análoga a classe **LiquidificadorDigital** compartilha os mesmos atributos e operações com a classe base **LiquidificadorGenerico**, implementando um construtor apenas por questões de clareza e um método para o controle de velocidade do liquidificador mantendo a mesma interface do Exemplo 20. Segue um outro diagrama UML completo para estas classes (Figura 22).

5.4 Polimorfismo

Através dos mecanismos de encapsulamento, novos métodos e atributos podem ser adicionados sem alteração das interfaces de programação e de usuário existentes de uma certa classe, constituindo um poderoso mecanismo de extensão de classes, mas o mecanismo de herança oferece possibilidades muito maiores.

A herança permite que novas classes sejam criadas e adicionadas a uma hierarquia sem a necessidade de qualquer modificação do código existente das classes e das aplicações que utilizam estas classes pois cada classe define estritamente seus próprios métodos e atributos. As novas classes adicionadas a hierarquia podem estender, especializar ou mesmo restringir o comportamento das classes originais.

Mas quando analisamos a hierarquia das classes no sentido inverso, isto é, nos dirigindo a superclasse da hierarquia, ocorre a generalização das diferentes classes interligadas. Isto significa que todas as classes pertencentes a uma mesma família de classes podem ser genericamente tratadas como sendo uma classe do tipo mais primitivo existente, ou seja, como sendo da própria superclasse da hierarquia.

Na Figura 22 temos a hierarquia das classe Liquidificador, onde podemos perceber que a classe **LiquidificadorGenerico**, como superclasse desta hierarquia, oferece tratamento genérico para as classse **LiquidificadorDigital** e **LiquidificadorAnalogico** bem como para quaisquer novas classes que sejam derivadas destas últimas ou da própria superclasse.

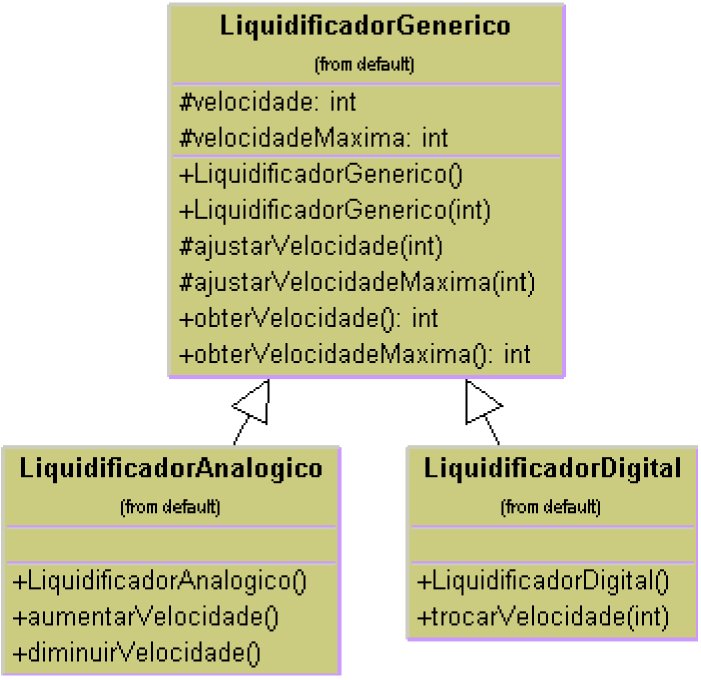


Figura 22 Classes Liquidificador

O tratamento generalizado de classes permite escrever programas que podem ser mais facilmente extensíveis, isto é, que podem acompanhar a evolução de uma hierarquia de classe sem necessariamente serem modificados. Enquanto a herança oferece um poderoso mecanismo de especialização, o polimorfismo oferece um igualmente poderoso mecanismo de generalização, constituindo uma outra dimensão da separação da interface de uma classe de sua efetiva implementação.

Imaginemos que seja necessário implementar numa classe **LiquidificadorInfo** um método que seja capaz de imprimir a velocidade atual de objetos liquidificador os quais podem ser tanto do tipo digital como analógico. Poderíamos, utilizando a sobrecarga, criar um método de mesmo nome que recebesse ou um **LiquidificadorAnalogico** ou um **LiquidificadorDigital**:

```
public class LiquidificadorInfo {

  public void velocidadeAtual(LiquidificadorAnalogico l) {
    System.out.println("Veloc. Atual: "+l.obterVelocidade());
  }

  public void velocidadeAtual(LiquidificadorDigital l) {
    System.out.println("Veloc. Atual: "+l.obterVelocidade());
  }
}
```

Embora correto, existem duas grandes desvantagens nesta aproximação: (i) para cada tipo de Liquidificador existente teríamos um método sobrecarregado avolumando o código que deveria ser escrito (e depois mantido) além disso (ii) a adição de novas subclasse que representassem outros tipos de liquidificador exigiria a implementação adicional de outros métodos semelhantes.

Através do polimorfismo pode-se chegar ao mesmo resultado mas de forma mais simples pois a classe **LiquidificadorGenerico** permite o tratamento generalizado de todas as suas subclasses.

```
public class LiquidificadorInfo {

  public void velocidadeAtual(LiquidificadorGenerico l) {
    System.out.println("Veloc. Atual: "+l.obterVelocidade());
  }
}
```

Nesta situação o polimorfismo permite que um simples trecho de código seja utilizado para tratar objetos diferentes relacionados através de seu ancestral comum, simplificando a implementação, melhorando a legibilidade do programa e também aumentando sua flexibilidade.

Outra situação possível é a seguinte: como qualquer método de uma superclasse pode ser sobreposto (*overhided*) em uma subclasse, temos que o comportamento deste método pode ser diferente em cada uma das subclasses, no entanto através da superclasse podemos genericamente usar tal método, que produz um resultado distinto para cada classe utilizada. Assim o polimorfismo também significa que uma mesma operação pode se comportar de forma diferente em classes diferentes. Para a orientação à objetos uma operação é uma ação ou transformação que um objeto pode realizar ou está sujeito e, desta forma, os métodos são implementações específicas das operações desejadas para uma certa classe.

Conclui-se através destes exemplos que o polimorfismo é uma característica importantíssima oferecida pela orientação à objetos que se coloca ao lado dos mecanismos de abstração de dados, herança e encapsulamento.

5.5 Composição

O termo composição significa a construção de novas classes através do uso de outras classes existentes, isto é, a nova classe possui internamente atributos que representam objetos de outras classes.

No código apresentado no Exemplo 37 Classe Baralho, temos que um dos atributos da classe **Baralho** é um vetor de objetos **Carta**. A implementação ocorreu desta forma porque um baralho é um conjunto bem definido de cartas, onde podemos dizer que o baralho “tem” cartas.

Uma classe **Cidade** poderia conter como atributos o seu nome, número de habitantes e suas coordenadas representadas através de uma classe **PontoGeográfico** (veja 3.9 Dicas para o Projeto de Classes) como abaixo:

```
public class Cidade {
  public string nome;
  public int populacao;
  public PontoGeográfico coordenadas;

  public Cidade() {
    nome = new String("");
    populacao = 0;
    coordenadas = new PontoGeografico();
  }
}
```

Enquanto o campo **populacao** é representado por um tipo primitivo, tanto o campo **nome** como o campo **coordenadas** são representados por outros objetos, isto é, são campos cujo tipo é uma outra classe, no caso, **String** e **PontoGeografico** respectivamente. Isto significa que um objeto pode ser construído através da associação ou agregação de outros objetos, o que se denomina composição.

A composição ou agregação é uma característica frequentemente utilizada quando se deseja representar uma relação do tipo “tem um” (*“has a” relationship*) indicando que um objeto tem ou outro objeto como parte de si. O baralho tem cartas assim como a cidade tem um nome e tem uma população. Desta forma é igualmente comum que nas classes que se utilizem de composição ocorra no construtor da classe ou em outro ponto do código a instanciação dos atributos do tipo objeto ou o recebimento de objetos construídos em outras partes do código.

Note que a relação definida para herança é bastante diferente, pois indica que um objeto é algo de um certo tipo (relação "é um" – *"is a" relatioship*), tal como o liquidificador analógico é um (é do tipo) liquidificador genérico.

5.6 Classes Abstratas

Em algumas circunstâncias desejamos orientar como uma classe deve ser implementada, ou melhor, como deve ser a interface de uma certa classe. Em outros casos, o modelo representado é tão amplo que certas classes tornam-se por demais gerais, não sendo possível ou razoável que possuam instâncias. Para estes casos dispomos das classes abstratas (*abtract classes*).

Tais classes são assim denominadas por não permitirem a instanciação, isto é, por não permitirem a criação de objetos do seu tipo. Sendo assim seu uso é dirigido para a construção de classes modelo, ou seja, de especificações básicas de classes através do mecanismo de herança.

Uma classe abstrata deve ser estendida, ou seja, deve ser a classe base de outra, mais específica que contenha os detalhes que não puderam ser incluídos na superclasse (abstrata). Outra possível aplicação das classes abstratas é a criação de um ancestral comum para um conjunto de classes que, se originados desta classe abstrata, poderão ser tratados genericamente através do polimorfismo.

Um classe abstrata, como qualquer outra, pode conter métodos mas também pode adicionalmente conter métodos abstratos, isto é, métodos que deverão ser implementados em suas subclasses não abstratas.

Vejamos um exemplo: ao invés de criarmos classes isoladas para representar quadrados, retângulos e círculos, poderíamos definir uma classe abstrata **Forma** que contivesse um vetor de pontos, um método comum que retornasse a quantidade de pontos da forma e métodos abstratos para desenhar e calcular a área desta forma. Note que seria impossível implementar métodos para o cálculo da área ou para o desenho de uma forma genérica, dirigindo-nos para uma classe abstrata como a exemplificado a seguir:

```
// Forma.java

public abstract class Forma {
  // atributos
  public java.awt.Point pontos[];

  // metodos comuns
  public int contagemPontos() {
    if (pontos != null)
      return pontos.length;
    else
      return 0;
  }

  // metodos abstratos
  public abstract void plotar(java.awt.Graphics g);
  public abstract double area();
}
```

Exemplo 46 Classe Abstrata Forma

A partir desta classe abstrata podem ser criadas outras classes derivadas que contenham os detalhes específicos de cada forma pretendida, tal como nos exemplos simplificados a seguir.

```
// Quadrado.java

public class Quadrado extends Forma {
  public Quadrado() {
    pontos = new java.awt.Point[4];
  }

  // implementacao dos metodos abstratos
  public void plotar(java.awt.Graphics g) {
    // codigo especifico vai aqui
  }

  public double area() {
    // codigo especifico vai aqui
  }
}
```

Exemplo 47 Classe Quadrado

```
// Circulo.java

public class Circulo extends Forma {
  public Circulo() {
    pontos = new java.awt.Point[1];
  }

  // implementacao dos metodos abstratos
  public void plotar(java.awt.Graphics g) {
    // codigo especifico vai aqui
  }

  public double area() {
    // codigo especifico vai aqui
  }
}
```

Exemplo 48 Classe Circulo

Se construíssemos um diagrama UML para esta hierarquia de classes obteríamos a Figura 23. Note o colorido diferentes utilizado para classes concretas e abstratas e o uso do itálico na denominação de classes e operações abstratas.

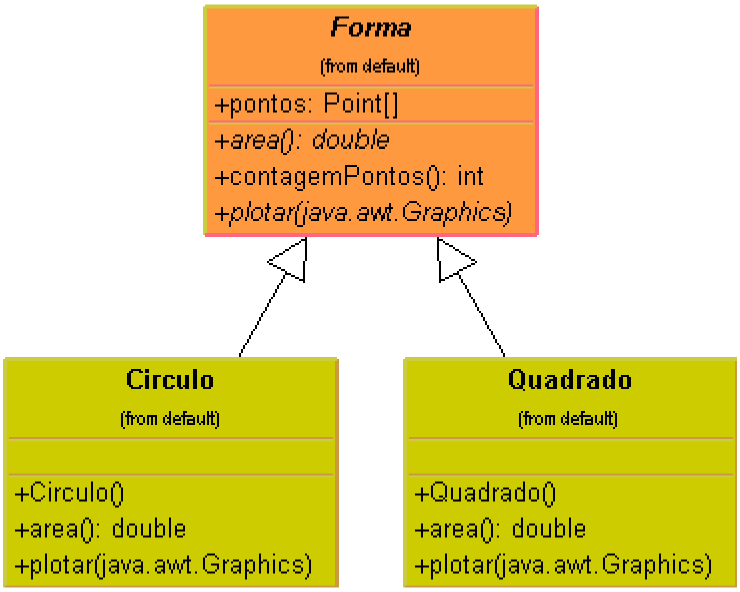


Figura 23 Hierarquia da Classe Forma e suas SubClasses

Como comentário final sobre as classes abstratas, existem algumas restrições na declaração de seus métodos: (i) os métodos estáticos (*static*) não podem ser abstratos, (ii) construtores não podem ser abstratos e (iii) os métodos abstratos não podem ser privados.

5.7 Interfaces

O Java permite a definição de um tipo especial de classe denominada interface. Uma interface é uma classe que permite obter resultados semelhantes ao obtidos com a herança múltipla, isto é, permite o compartilhamento das interfaces das classes envolvidas sem o compartilhamento de suas implementações, levando ao extremo o conceito da orientação à objetos.

Isto significa que, embora o Java não oferece os mecanismos de herança múltipla (quando uma classe compartilha a interface e implementação de várias classes base simultaneamente), é possível obter-se a melhor parte da herança múltipla (ao compartilhamento apenas da interface da classe) através deste novo mecanismo. O nome interface se origina exatamente do fato que o que se pretende é uma definição compartilhável da interface propriamente dita de uma classe mas não de sua implementação efetiva.

Clarificando, uma interface tem construção semelhante a de uma classe abstrata (uma classe que contêm a declaração de variáveis membro e métodos dos quais alguns podem ser implementados, como visto na seção 5.6) embora se distinga por dois fatos: (i) nenhum dos métodos declarados pode ser implementado e (ii) todas variáveis membro deve ser estáticas e finais.

A vantagem de utilizar-se as interfaces é a definição de um protocolo entre classes, isto é, uma especificação do que uma classe deve oferecer e implementar em termos de seus métodos o que resulta numa poderosa abstração.

Um interface pode ser declarada como exemplificado a seguir:

```
interface NomeDaInterface {
  // declaração de atributos final static
  ⋮
  // declaração de métodos (sem implementação)
  ⋮
}
```

É possível criarmos uma interface estendendo outra, isto é, através da herança simples de outra interface, mas não de uma classe simples e vice-versa, daí:

```
interface NomeDaInterface extends InterfaceBase{
  // declarações de atributos e métodos
  ⋮
}
```

As classes **java.awt.Polygon** e **java.awt.Rectangle**, entre outras, implementam a interface **java.awt.Shape** que define um único método **getBounds** que se destina a prover uma interface comum para os vários tipos de objetos geométricos na obtenção dos limites definidos para tais objetos. Várias outras classes da API do Java implementam uma ou mais interfaces como meio compartilhar definições comuns, o que em última instância permite suportar tratamento polimórfico para seus objetos.

5.8 RTTI

Java oferece através da classe **java.lang.Class** e do pacote **java.lang.reflect** (reflexão) um conjunto de classes destinadas a extração de informações de outras classes em tempo de execução, o que é com frequência chamado de RTTI (*Run Time Type Information*). Embora o uso direto destas classes não seja comum, possibilitam a obtenção de informações que podem não estar disponíveis durante a compilação, como no caso de pacotes integráveis a ambientes de desenvolvimento tais como o s **JavaBeans**.

A classe **java.lang.Class** oferece vários métodos que permite relacionar quais métodos e construtores existem dentro de uma dada classe bem como a identificação de sua superclasse. Os principais métodos desta classe estão listados na Tabela 16.

Método	**Descrição**
forName(String)	Retorna um objeto tipo **Class** para a *string* dada.
getClassLoader()	Retorna o carregador da classe.
getConstructors()	Retorna um vetor contendo todos construtores da classe.
getDeclaredConstructors()	Retorna um vetor contendo apenas os construtores declarados na classe.
getDeclaredMethods()	Retorna um vetor contendo apenas os métodos declarados na classe.
getMethods()	Retorna um vetor contendo todos métodos da classe incluindo os métodos compartilhados com as superclasses.
getName()	Obtêm no nome completamente qualificado da classe.
getSuperclass()	Obtêm o nome da superclasse.

Tabela 16 Métodos da Classe java.lang.Class

Os métodos da classe **java.lang.Class** são tipicamente utilizados em associação ao pacote **java.lang.reflect**, que permite obter informações mais detalhadas sobre os campos, construtores e métodos de uma classe bem como de seus argumentos, valores de retorno e modificadores. O pacote **java.lang.reflect** contêm as seguintes classes:

Classe	**Descrição**
Array	Provê métodos estáticos capazes de obter informações para criar e acessar dinamicamente *arrays*.
Constructor	Provê métodos estáticos capazes de obter informações sobre construtores de uma dada classe em tempo de execução.
Field	Provê métodos estáticos capazes de obter informações sobre campos (atributos) de uma dada classe em tempo de execução.
Method	Provê métodos estáticos capazes de obter informações sobre métodos (de instância, classe e abstratos) de uma dada classe em tempo de execução.
Modifier	Provê métodos estáticos capazes de obter informações sobre ao modificadores de uma classe.

Tabela 17 Classes do Pacote java.lang.reflect

A seguir temos o código de uma pequena aplicação que extrai informações de classes existentes, isto é, retorna uma relação dos construtores e métodos disponíveis em uma dada classe.

```
// ClassInfo.java

import java.lang.reflect.*;

public class ClassInfo {
  static final String usage =
    "Uso:\n\tClassInfo [+] pacote.class [palavra]\n" +
    "Exibe relacao de metodos da classe 'pacote.class' " +
    "ou apenas\naqueles envolvendo 'palavra' (opcional).\n" +
    "A opcao '+' indica que tambem devem ser listados os " +
    "metodos\n e construtores das superclasses.\n";
```

```
  public static void main(String args[]) {
    Class classe;
    Method metodos[];
    Constructor construtores[];
    boolean incluirHeranca;
    int argBusca = 0;
    try {
      incluirHeranca = args[0].equals("+");
      if (incluirHeranca) {
        argBusca = (args.length>2 ? 2 : 0);
        classe = Class.forName(args[1]);
        metodos = classe.getMethods();
        construtores = classe.getConstructors();
      } else {
        argBusca = (args.length>1 ? 1 : 0);
        classe = Class.forName(args[0]);
        metodos = classe.getDeclaredMethods();
        construtores = classe.getDeclaredConstructors();
      }
      if (argBusca==0) {
        for (int i=0; i<construtores.length; i++)
          System.out.println(construtores[i].toString());
        for (int i=0; i<metodos.length; i++)
          System.out.println(metodos[i].toString());
      } else {
        for (int i=0; i<construtores.length; i++)
          if (construtores[i].toString().indexOf(args[argBusca])
!= -1)
            System.out.println(construtores[i].toString());
        for (int i=0; i<metodos.length; i++)
          if (metodos[i].toString().indexOf(args[argBusca])!=-1)
            System.out.println(metodos[i].toString());
      }
    } catch (ClassNotFoundException e) {
      System.out.println("Classe não encontrada: " + e);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println(usage);
    }
    System.exit(0);
  }
}
```

Exemplo 49 Classe ClassInfo

Executando-se esta aplicação obtêm-se uma listagem dos métodos contidos em uma classe. Caso tal relação seja muito extensa, pode-se redirecionar o seu resultado para um arquivo auxiliar que pode ser posteriormente consultado através de um editor de textos qualquer como indicado:

```
java ClassInfo java.lang.String > aux
```

```
notepad aux
```

Este utilitário simples permite substituir a documentação da API nos casos onde se deseje apenas saber quais métodos estão disponíveis e quais seus argumentos

6 Aplicações Gráficas com a AWT

Aplicações gráficas são aquelas destinadas a execução dentro dos ambientes gráficos oferecidos por vários sistemas operacionais. Uma GUI (*Graphical User Interface*) é um ambiente pictórico que oferece uma interface mais simples e intuitiva para os usuários.

Cada sistema operacional pode oferecer GUIs com aparências (*look and feel*) distintas tais como o modelo COM da *Microsoft*, o *Presentation Manager* da IBM, o *NeWS* da *Sun* ou o *X-Window System* do MIT.

A linguagem Java oferece capacidades únicas no desenvolvimento de aplicações gráficas que, sem modificação ou recompilação, podem ser executadas em diferentes ambientes gráficos, uma vantagem significativa além do próprio desenvolvimento de aplicações gráficas.

6.1 Componentes e a AWT

Um dos elementos chave do sucesso das GUIs é a utilização de componentes padronizados para representação e operação da interface em si. Um componente GUI é um objeto visual com o qual o usuário pode interagir através do teclado ou mouse, permitindo realizar uma ou algumas dentre diversas operações: a entrada de valores numéricos ou de texto, a seleção de itens em listas, a marcação de opções, o desenho livre, o acionamento de operações etc.

Os componentes GUI (GUI *components*), também chamados de *widgets* (*window gadgets* – dispositivos de janela) são os elementos construtivos das interfaces GUI, isto é botões, caixas de entrada de texto, caixas de lista, caixas de seleção, botões de seleção única, botões de seleção múltipla, barras de rolagem etc.

O Java oferece uma ampla biblioteca de componentes GUI e de capacidades gráficas na forma de classe pertencentes ao seu pacote **java.awt**, mais conhecido como AWT (*Abstract Window Toolkit*). A AWT oferece classes comuns e abstratas relacionadas a renderização e obtenção de informações do sistema gráfico oferecido pela plataforma sobre a qual opera a máquina virtual Java. Uma hierarquia parcial das classes da AWT pode ser vista na Figura 24.

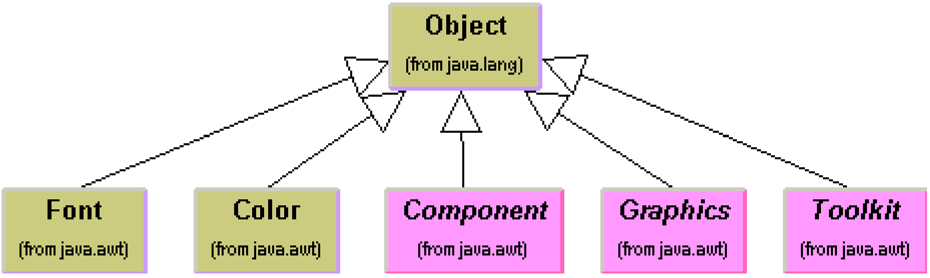


Figura 24 Hierarquia Parcial de Classes java.awt

Todas as classes básicas da AWT são extensões da classe **java.lang.Object** que é em última análise a superclasse de todas as classes Java. A classe **java.awt.Font** trata dos fontes, de suas características e representação. A classe **java.awt.Color** oferece meios para representação de cores. A classe abstrata **java.awt.Component** é a base para construção de todos os componentes GUI oferecidos pelo Java através da AWT. A classe **java.awt.Graphics** oferece um sofisticado conjunto de primitivas de desenho. A classe **java.awt.Toolkit** permite obter informações sobre o sistema gráfico em uso pela JVM.

A idéia central da AWT se baseia num máximo denominador comum entre as plataformas suportadas, isto é, a AWT implementa componentes e facilidade que podem ser reproduzidas de forma equivalente entre as diversas plataformas para as quais são destinadas JVM, garantindo o máximo de portabilidade. Na prática isto significa que um dado componente, um botão por exemplo, deve possuir representação nativa equivalente no *Microsoft Windows*, no *Unix X-Windows* ou *Motif* e em qualquer outra plataforma suportada. Mesmo que a aparência do componente seja distinta, os princípios básicos de funcionamento e operação devem ser os mesmos.

6.2 Construindo uma Aplicação Gráfica

Quando nos decidimos pela construção de uma aplicação gráfica é necessário projetar-se uma interface para a aplicação, isto é, é necessário selecionar-se quais componentes serão utilizados, qual o objetivo de sua utilização e qual a disposição desejada para tais componentes. A partir deste planejamento básico devemos adicionar código para praticamente tudo pois o JDK não oferece uma interface para construção visual de interfaces gráficas tal como o *Microsoft VisualBasic* ou o *Borland Delphi*.

A implementação de uma aplicação gráfica envolve três passos simples:

(i) Instanciação do componentes selecionados

Cada componente a ser adicionado na interface deve ser individualmente instanciado. Como um componente é uma classe, basta instanciar um objeto da classe desejada para criarmos um componente. Se for necessário referenciar-se o componente posteriormente deve ser mantida uma variável para tal finalidade. Usualmente os componentes da interface são associados a variáveis-membro da classe para que possam ser referenciados por todos os métodos desta. Um exemplo de instanciação de componentes poderia ser:

```
// declaração de atributos da classe
private Button b1;
private Button b1;
⋮
// construtor da classe
b1 = new Button(“Ok”);
b2 = new Button(“Fechar”);
```

(ii) Adição do componente na interface

Todos os componentes que deverão ser exibidos na interface devem ser individualmente adicionado a esta, ou seja, cada um dos componentes deve ser adicionado em um dos *containers* existentes na interface sendo especificado o seu posicionamento através de um dos gerenciadores de *layout*. Os botões instanciados no passo anterior poderiam ser adicionados a interface como sugerido abaixo:

```
                             // Adição de componentes:
add(b1);                     // utilizando o layout default
add(b2, BorderLayout.SOUTH); // utilizando o Border layout
```

(iii) Registro dos métodos processadores de eventos

A interface das aplicações é verdadeiramente renderizada pela GUI oferecida pelo sistema operacional. Assim sendo, quando ocorre a interação do usuário com a aplicação através do teclado ou *mouse*, o sistema operacional envia para a aplicação responsável pela janela uma série detalhada de mensagens narrando o tipo e forma da interação. Para que a aplicação possa processar a interação do usuário métodos especiais (*event listeners*) capazes de receber tais mensagens devem ser adicionados a aplicação e associados aos componentes que desejamos ser capazes de reagir à interação com o usuário. Um exemplo de registro de tais métodos para os botões do exemplo seria:

```
// registro dos listeners
b1.addActionListener(this);
b2.addActionListener(new ButtonHandler());
```

Seguindo-se estes passos torna-se possível criarmos aplicações gráficas que utilizem componentes da GUI. Para que isto seja efetivamente possível necessitamos conhecer quais os componentes básicos, como instanciá-los, como adicioná-los a interface e, finalmente, como criar e registrar os métodos processadores de eventos necessários ao funcionamento das interfaces GUI.

6.3 Componentes Básicos

Como mencionado anteriormente a classe abstrata **java.awt.Component** é a base para construção de todos os componentes GUI oferecidos pelo Java através do pacote **java.awt**. Esta classe oferece a infra-estrutura necessária para criação de objeto que possuam uma representação gráfica que pode ser exibida na tela e com a qual o usuário pode interagir.

Como classe abstrata não pode ser instanciada diretamente mas utilizada como um *framework* para a criação de novos componentes (classes derivadas) ou como uma classe que permite o tratamento generalizado dos componentes derivados através do polimorfismo. Se utilizada como classe base para criação de componentes, podem ser criados componentes leves (*lightweight components*), isto é, componentes não associados com janelas nativas.

Esta classe serve de origem para construção dos seguintes componentes da AWT:

Button (botão)
Canvas (canvas)
Checkbox (caixa de opção)
CheckboxGroup (grupo de botões de opção)
CheckboxMenuItem (item de menu opção)
Choice (caixa de seleção)
Dialog (janela de diálogo)
FileDialog (janela de diálogo para arquivos)
Frame (janela completa)
Label (rótulo de texto)
List (caixa de lista)
Menu (menu *pull-down*)
MenuBar (barra de menus)
MenuItem (item de menu)
MenuShortcut (atalho para item de menu)
Panel (painel)
PopupMenu (menu *pop-up*)
ScrollPane (painel deslocável)
ScrollBar (barra de rolagem)
TextArea (caixa multilinha de texto)
TextField (caixa de entrada de texto)
Window (janela simples)

Uma hierarquia parcial dos componentes da AWT baseados na classe **java.awt.Component** é apresentada na Figura 25. O entendimento desta hierarquia é fundamental para que seus componentes possam ser efetivamente utilizados.

Além dos componentes mencionados, a classe **java.awt.Component** também é base para construção da classe abstrata **Container**. Esta classe representa um *container*, ou seja, uma área onde outros componentes podem ser adicionados e posicionados. Classes derivadas da **java.awt.Container** também são **Containers**, isto é, também podem conter outros componentes. Exemplos de componentes deste tipo são as janelas (**java.awt.Frame**, **java.awt.Window** etc.) e painéis (**java.awt.Panel**).

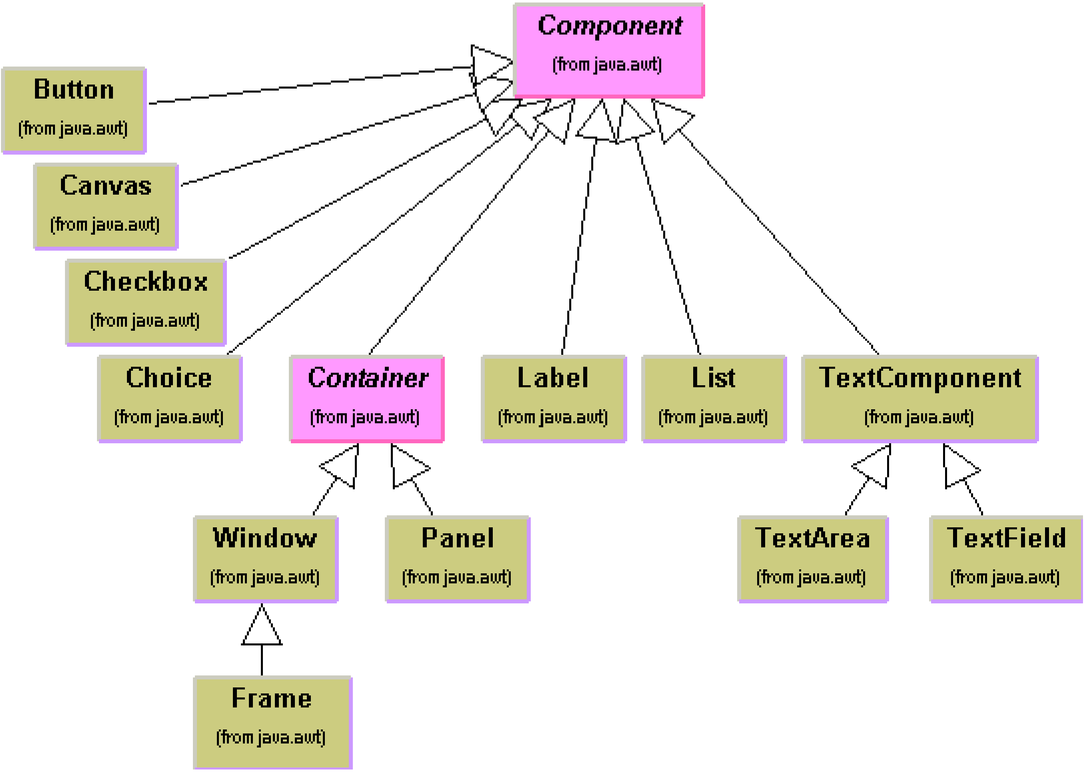


Figura 25 Hierarquia Parcial de Classes java.awt (Componentes GUI)

A classe **java.awt.Component** contêm portanto métodos comuns a todos os demais componentes da AWT. Destacamos na Tabela 18 e Tabela 19 apenas alguns dos muitos métodos disponíveis.

Método	Descrição
add(PopupMenu)	Adiciona um menu *popup* ao componente.
addKeyListener(KeyListener)	Adiciona o processador de eventos de teclas.
addMouseListener(MouseListener)	Adiciona o processador de eventos do *mouse*.
getBackground()	Obtêm a cor do segundo plano do componente.
getBounds()	Obtêm os limites deste componente como um objeto **Rectangle**.
getComponentAt(int, int)	Retorna o componente contido na posição dada.
getCursor()	Obtêm o cursor atual do componente.
getFont()	Obtêm o fonte atual do componente.
getGraphics()	Obtêm o contexto gráfico do componente.
getLocation()	Obtêm a posição (esquerda, topo) do componente.
getSize()	Obtêm o tamanho do componente como um objeto **Dimension**.
isEnabled()	Determina se o componente está habilitado.
isVisible()	Determina se o componente está visível.

Tabela 18 Métodos da classe java.awt.Component (Parte A)

Método	Descrição
paint(Graphics)	Renderiza o componente.
repaint()	Atualiza a renderização do componente.
setBounds(int, int, int, int)	Move e redimensiona o componente.
setCursor(Cursor)	Especifica o cursor para o componente.
setEnabled(Boolean)	Habilita ou desabilita o componente.
setFont(Font)	Especifica o fonte para o componente.
setLocation(int, int)	Move o componente para a posição dada.
setSize(int, int)	Especifica o tamanho para o componente.
setVisible(Boolean)	Especifica se o componente é ou não visível.
update(Graphics)	Atualiza a exibição do componente.

Tabela 19 Métodos da classe java.awt.Component (Parte B)

6.3.1 Frame

O componente **java.awt.Frame** é uma janela do sistema gráfico que possui uma barra de título e bordas, comportando-se como uma janela normal da GUI. Como é uma subclasse do **java.awt.Container** (veja Figura 25) pode conter outros componentes sendo esta sua finalidade principal. O alinhamento *default* para componentes adicionados a um **Frame** é o **java.awt.BorderLayout**, como será discutido mais a frente. Além disso, implementa a interface **java.awt.MenuContainer** de forma que também possa conter uma barra de menus e seus menus associados.

A seguir temos a Tabela 20 que lista os construtores e alguns dos métodos desta classe:

Método	Descrição
Frame()	Constrói uma nova instância, invisível e sem título.
Frame(String)	Constrói uma nova instância, invisível e com o título dado.
dispose()	Libera os recursos utilizados por este componente.
getTitle()	Obtêm o título da janela.
isResizable()	Determina se a janela é ou não dimensionável.
setMenuBa(MenuBar)	Adiciona a barra de menu especificada a janela.
setResizable(Boolean)	Especifica se a janela é ou não dimensionável.
setTitle(String)	Especifica o título da janela.

Tabela 20 Métodos da Classe java.awt.Frame

Nesta classe ainda existem constantes[1] para especificação de diversos cursores como exemplificado na Tabela 21. Verifique a documentação do Java a existência de outros cursores cujo formato deve ser consultado na documentação do sistema operacional.

A utilização do componente **java.awt.Frame** determina a estratégia básica para criação de aplicações gráficas. Usualmente cria-se uma nova classe que estende a classe **java.awt.Frame** e com isto obtêm-se duas facilidades: (i) uma janela para apresentação gráfica da aplicação e (ii) um *container* para disposição de outros componentes GUI da aplicação. Na classe do primeiro **Frame** que desejamos exibir adiciona-se o método **main** responsável por instanciar e exibir o **Frame** através do método **show** (que pertence a classe **java.awt.Window**).

---

[1] Como será notado, a atual API do Java peca na denominação de constantes. Embora a recomendação seja de nomes escritos exclusivamente com maiúsculas, constantes presentes em outras classes nem sempre respeitam esta regra.

Constante	Descrição
CROSSHAIR_CURSOR	Cursor em forma de cruz.
DEFAULT_CURSOR	Cursor *default*.
HAND_CURSOR	Cursor tipo apontador.
MOVE_CURSOR	Cursor para indicação de possibilidade de movimento.
TEXT_CURSOR	Cursor de edição de texto.
WAIT_CURSOR	Cursor de sinalização de espera.

Tabela 21 Cursores da classe java.awt.Frame

Observe o exemplo dado a seguir:

```
// FrameDemo1.java

import java.awt.*;

public class FrameDemo1 extends Frame{
  // construtor
  public FrameDemo1() {
    super("Frame Demo 1");
    setSize(320, 240);
    setLocation(50, 50);
  }
  // main
  static public void main (String args[]) {
    FrameDemo1 f = new FrameDemo1();
    f.show();
  }
}
```

Exemplo 50 Classe FrameDemo1 (Uso Básico de Frames)

Compilando e executando a aplicação obtemos o resultado ilustrado na Figura 26.

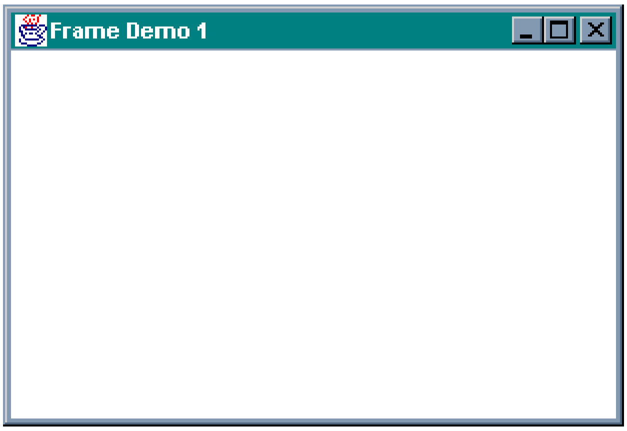


Figura 26 Aplicação FrameDemo1

Apesar da aparência esperada, das capacidades de dimensionamento e iconização, a janela não é fechada quando acionamos a opção “Fechar”. Isto ocorre por não haver métodos processadores de eventos capazes de receber tal mensagem encerrando a aplicação. Para encerrar a aplicação devemos encerrar a JVM com um CTRL+C na janela de console exibida.

Para que uma janela tenha seus eventos processados é necessário implementarmos um processador de eventos específico. Existem basicamente duas opções para a solução desta questão: (i) utilizar a classe **java.awt.event.WindowAdapter** ou (ii) utilizar a interface **java.awt.event.WindowListener**.

A classe **java.awt.event.WindowAdapter** é abstrata e necessita portanto ser estendida para ser implementada. Como nossa aplicação também necessita estender a classe **java.awt.Frame** temos um impasse: Java não oferece herança múltipla, portanto devemos derivar a classe de nossa aplicação da classe **java.awt.Frame** e implementar uma classe distinta para o processador de eventos baseado na classe **java.awt.event.WindowAdapter**. Por outro lado esta classe adaptadora só necessita um único método sendo assim bastante simples.

A segunda solução exige que implementemos nossa classe como uma interface do tipo **java.awt.event.WindowListener**, o que pode ser facilmente obtido a partir da classe de nossa aplicação pois o Java admite a implementação de múltiplas interfaces numa mesma classe. Embora necessitemos de uma única classe a interface **java.awt.event.WindowListener** exige a implementação de 7 diferentes métodos, dos quais apenas um será utilizado (esta interface será detalhada em seções seguintes).

Vejamos o código do exemplo anterior modificado para implementação da interface **java.awt.event.WindowListener**. Note a adição de uma nova diretiva *import* para indicarmos o uso do pacote **java.awt.event**.

```java
// FrameDemo2.java

import java.awt.*;
import java.awt.event.*;

public class FrameDemo2 extends Frame
                        implements WindowListener {
  // main
  static public void main(String args[]) {
    FrameDemo2 f = new FrameDemo2();
    f.show();
  }

  // construtor
  public FrameDemo2() {
    super("Frame Demo 2");
    setSize(320, 240);
    setLocation(50, 50);
    addWindowListener(this);
  }

  // WindowListener
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}
```

Exemplo 51 Classe FrameDemo2 (Uso da Interface WindowListener)

O resultado obtido é o semelhante ao do exemplo anterior, conforme ilustrado na Figura 26, exceto pelo fato de que agora a janela pode ser fechada através da opção correspondente ou do botão existente na barra de título da janela. O método da interface **WindowListener** responsável pelo processamento da mensagem de fechamento de janela é o **windowClosing**, que contêm apenas uma chamada ao método estático **exit**, da classe **java.lang.System**, que encerra a aplicação.

A outra solução exige a implementação de uma classe em separado (um segundo arquivo fonte) para o **java.awt.event.WindowAdapter** que possui um único método, **windowClosing**, responsável por processar o evento de “fechar janela”, como mostrado abaixo:

```
// CloseWindowAndExit.java

import java.awt.*;
import java.awt.event.*;

class CloseWindowAndExit extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

Exemplo 52 Classe CloseWindowAndExit (Uso da Classe WindowAdapter)

Desta forma o código original da aplicação exibido no Exemplo 50 deve ser modificado como indicado a seguir para utilizar a classe **CloseWindowAndExit** que implementa um **WindowAdapter**.

```
// FrameDemo3.java

import java.awt.*;

public class FrameDemo3 extends Frame{

  public FrameDemo3() {
    super("Frame Demo 3");
    setSize(320, 240);
    setLocation(50, 50);
  }

  static public void main (String args[]) {
    FrameDemo3 f = new FrameDemo3();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
}
```

Exemplo 53 Classe FrameDemo3 (Uso de Classe WindowAdapter)

Executando esta aplicação notamos que sua funcionalidade é a mesma do exemplo anterior, ou seja, o uso da classe **java.awt.event.WindowAdapter** implementada em separado surtiu o mesmo efeito que a incorporação da interface **java.awt.event.WindowListener** ao código básico da aplicação.

Concluímos que o uso da classe em separado com o **WindowAdapter** (Exemplo 53) confere maior simplicidade ao código além de possibilitar sua reutilização em outras aplicações. Ao mesmo tempo é mais restritiva pois trata apenas o evento de fechamento de janelas, sugerindo que o uso da interface **WindowListener** é mais apropriado quando torna-se necessário processar outros eventos associados a janela da aplicação.

Como último exemplo destinado especificamente ao uso dos **Frames**, mostramos um uso da classe **java.awt.Toolkit** na determinação do tamanho da tela através do retorno de um objeto **java.awt.Dimension**, que encapsula a altura (**height**) e largura (**width**) de um componente. Com o uso destas classes determinamos dinamicamente o tamanho e posição da janela da aplicação.

Nesta aplicação também exemplificamos o uso do objeto **java.awt.Color**. Este objeto encapsula uma representação de cores no formato RGB (*Red Green Blue*), possuindo um construtor que aceita três valores inteiros entre 0 e 255 para cada uma das componentes da cor, possibilitando a especificação de uma dentre 65.536 cores diferentes.

Consulte a documentação da API do Java para conhecer as constantes existentes para cores padronizadas, tais como **Color.black**, **Color.white**, **Color.red**, **Color.blue**, **Color.yellow**, **Color.magenta, Color.green**, **Color.gray** etc.

Além destas constantes de cor existem outras associadas ao sistema, isto é, as cores utilizadas pela GUI do sistema, na classe **java.awt.SystemColor** que indicam as cores utilizadas por controles (**SystemColor.control**), sombras (**SystemColor.controlShadow**), barras de título (**SystemColor.activeCaption**), menus (**SystemColor.menu**) etc.

```
// FrameDemo4.java

import java.awt.*;

public class FrameDemo4 extends Frame {

  public FrameDemo4() {
    super("Frame Demo 4");
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    setSize(d.width/2, d.height/2);
    setLocation(d.width/2, d.height/2);
    setBackground(new Color(0,128,128));
  }

  static public void main (String args[]) {
    FrameDemo4 f = new FrameDemo4();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
}
```

Exemplo 54 Classe FrameDemo4

6.3.2 Label

O componente **java.awt.Label**, conhecido também como rótulo de texto ou simplesmente rótulo, é destinado a exibição de texto dentro de quaisquer componentes do tipo *container*. Os rótulos podem exibir uma única linha de texto a qual não pode ser modificada nem editada pelo usuário, caracterizando um componente passivo. Métodos apropriados são fornecidos para que o texto de um rótulo possa ser programaticamente obtido e modificado bem como seu alinhamento. Na Tabela 22 temos os construtores e os principais métodos desta classe.

O alinhamento pode ser especificado ou determinado através de três constantes disponíveis nesta classe: **Label.LEFT**, **Label.CENTER** e **Label.RIGHT**. Além destes métodos estão disponíveis outros, pertencentes as classes ancestrais, tal como ilustra a Figura 25.

Método	Descrição
Label()	Constrói um novo rótulo sem texto.
Label(String)	Constrói um novo rótulo com o texto dado.
Label(String, int)	Constrói um novo rótulo com o texto e alinhamento dados.
getAlignment()	Obtêm o alinhamento do rótulo.
getText()	Obtêm o texto do rótulo.
setAlignment(int)	Especifica o alinhamento do rótulo.
setText(String)	Especifica o texto do rótulo.

Tabela 22 Métodos da Classe java.awt.Label

A seguir o exemplo de uma aplicação muito simples que instancia e adiciona alguns rótulos a sua interface. Note que é possível adicionar-se rótulos a interface sem manter-se uma referência para o mesmo o que é útil quando seu conteúdo é sabidamente fixo, embora tal componente se torne inacessível programaticamente.

```
// LabelDemo.java

import java.awt.*;

public class LabelDemo extends Frame{
  private Label l1;
  private Label l2;

  public LabelDemo() {
    super("Label Demo");
    setSize(100, 200);
    setLocation(50, 50);
    // instanciacao
    l1 = new Label("Java:");
    l2 = new Label("Write Once...", Label.CENTER);
    // alteracao do layout do Frame
    setLayout(new FlowLayout());
    // adicao dos componentes
    add(l1);
    add(l2);
    add(new Label("...Run Everywhere!", Label.RIGHT));
  }

  static public void main (String args[]) {
    LabelDemo f = new LabelDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
}
```

Exemplo 55 Classe LabelDemo

Observe que o *layout* do **Frame** é modificado para **java.awt.FlowLayout** (por questões de simplificação) ao invés do *default* **java.awt.BorderLayout**. O resultado desta aplicação é ilustrado a seguir:

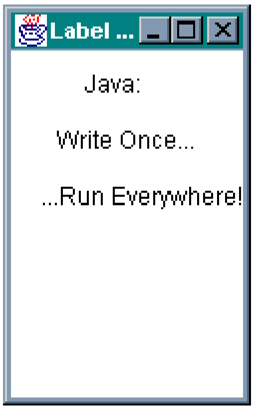


Figura 27 Aplicação LabelDemo

Se a janela for redimensionada, os rótulos são automaticamente reposicionados, o que em algumas situações anula o efeito pretendido com o alinhamento individual de cada texto associado aos componentes **java.awt.Label**.

6.3.3 Button

Os botões, componentes encapsulados na classe **java.awt.Button**, são como painéis rotulados com um texto que, quando acionados, podem provocar a execução de alguma rotina ou sequência de comandos. O acionamento dos botões é visualmente indicado através de um efeito de afundamento de sua superfície. O texto associado ao botão, tal como nos rótulos, não pode ser alterado pelo usuário embora isto possa ser realizado através da aplicação.

Os principais construtores e métodos desta classe são listados na tabela a seguir:

Método	Descrição
Button()	Constrói um novo botão sem texto.
Button (String)	Constrói um novo botão com o texto dado.
addActionListener(ActionListener)	Registra uma classe *listener* (processadora de eventos) **ActionListener** para o botão.
getLabel()	Obtêm o texto do botão.
setLabel(String)	Especifica o texto do botão.

Tabela 23 Métodos da Classe java.awt.Button

Sendo um componente ativo, para que o botão responda ao seu acionamento pelo usuário é necessário registrar-se uma classe que processe o evento especificamente gerado por tal ação. Estas classes devem implementar a interface **java.awt.event.ActionListener**, que exige a presença de um único método, o **actionPerformed** como exemplificado a seguir.

```
// ButtonDemo.java

import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends Frame implements ActionListener{
  private Label label1;
  private Button button1;
  private int count;

  public ButtonDemo() {
    super("Button Demo");
    setSize(280, 70);
    setLayout(new FlowLayout());
    // instanciacao, registro e adicao do botao
    button1 = new Button("Clique Aqui");
    button1.addActionListener(this);
    add(button1);
    // instanciacao e adicao do label
    label1 = new Label("Botão não foi usado.");
    add(label1);
    count=0;
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==button1) {
      count++;
      label1.setText("Botão já foi usado " + count + "
vez(es).");
    }
  }

  static public void main (String args[]) {
    ButtonDemo f = new ButtonDemo();
    f.addWindowListener(new CloseWindowAndExit());
```

```
    f.show();
  }
}
```

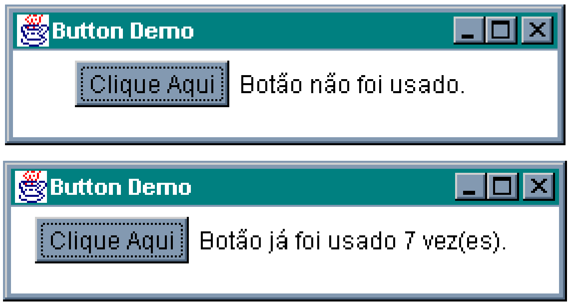
Exemplo 56 Classe ButtonDemo

A aplicação contabiliza a quantidade de vezes que o botão foi acionado indicando tal valor através do rótulo existente. Toda a ação está codificada no método **actionPerformed**, copiado abaixo:

```
public void actionPerformed(ActionEvent e) {
    if (e.getSource()==button1) {
      count++;
      label1.setText("Botão já foi usado " + count + "
vez(es).");
    }
  }
```

O método **actionPerformed** é o *listener* que recebe um evento de ação (**ActionEvent**) que é produzido por botões. O comando *if* apenas verifica a origem do evento através do método **getSource** pertencente à classe deste evento. Sendo o botão **button1** (a variável membro contêm uma referência para o componente que é comparada a referência do componente que originou o evento) incrementa-se a contagem e atualiza-se o texto do rótulo.

O resultado da execução desta aplicação é exibido em duas situações, a inicial e outra após o uso repetido do botão:



Botão não foi usado.

Botão já foi usado 7 vez(es).

Figura 28 Aplicação ButtonDemo

Se existissem dois botões, **button1** e **button2**, o mesmo *listener* poderia utilizado como sugerido no trecho abaixo:

```
// no construtor
    button1.addActionListener(this);
    button2.addActionListener(this);

public void actionPerformed(ActionEvent e) {
    if (e.getSource()==button1) {
      // ações correspondentes ao button1
      ⋮
    } else if (e.getSource()==button2)
      // ações correspondentes ao button2
      ⋮
    }
  }
```

Exemplo 57 *Listener* Único para 2 Componentes

Esta mesma técnica pode ser utilizada para mais componentes, embora com o crescimento do método **actionPerformed** se recomende sua divisão. Deve-se então buscar o equilíbrio entre simplicidade e clareza do código.

6.3.4 TextField

A AWT oferece através da classe **java.awtTextField** o componente caixa de entrada de texto ou apenas caixa de texto. Este componente possibilita a edição e entrada de uma única linha de texto de forma bastante conveniente para o usuário. Todo e qualquer dado inserido através deste componente é tratado primariamente como texto, devendo ser convertido explicitamente para outro tipo caso desejado.

A classe **java.awt.TextField** contêm, entre outros, os seguintes construtores e métodos:

Métodos	Descrição
TextField()	Constrói nova caixa de texto sem conteúdo.
TextField(String)	Constrói nova caixa de texto com conteúdo dado.
TextField(String, int)	Constrói nova caixa de texto com conteúdo dado e na largura especificada em colunas.
addActionListener(ActionListener)	Registra uma classe *listener* (processadora de eventos) **ActionListener** para o **TextField**.
echoCharIsSet()	Verifica que caractere de eco está acionado.
getColumns()	Obtêm número atual de colunas.
getEchoChar()	Obtêm caractere atual de eco.
setColumns(int)	Especifica número de colunas.
setEchoChar(char)	Especifica caractere de eco.

Tabela 24 Métodos da Classe java.awt.TextField

Note que nos métodos relacionados não existe menção ao controle do texto contido pelo componente. Observando a hierarquia dos componentes da AWT (Figura 25) podemos perceber que a classe **java.awt.TextField** é derivada da classe **java.awt.TextComponent** que oferece uma infra-estrutura comum para os componentes de texto básicos (**java.awt.TextField** e **java.awt.TextArea**). Na Tabela 25 temos os métodos mais importantes da classe **java.awt.TextComponent**.

Método	Descrição
addTextListener(TextListener)	Adiciona uma classe que implementa um *listener* **java.awt.event.TextListener** para os eventos associados ao texto deste componente.
getCaretPosition()	Obtêm a posição do cursor de edição de texto.
getSelectedText()	Obtêm o texto atualmente selecionado.
getText()	Obtêm todo o texto do componente.
isEditable()	Verifica se o componente é ou não editável.
select(int, int)	Seleciona o texto entre as posições dadas.
selectAll()	Seleciona todo o texto.
setCaretPosition(int)	Especifica a posição do cursor de edição de texto.
setEditable(Boolean)	Especifica se o componente é ou não editável.
setText(String)	Especifica o texto contido pelo componente.

Tabela 25 Métodos da Classe java.awt.TextComponent

Utilizando os métodos implementados diretamente na classe **TextField** e também aqueles em sua classe ancestral **TextComponent**, podemos utilizar eficientemente este componente. Note que podem existir dois *listeners* associados a este componente: um **ActionListener**, que percebe quando o usuário aciona a tecla “Enter” quando o foco esta no componente **TextField**, e um **TextListener** que percebe eventos associados a edição do texto.

No exemplo a seguir demostraremos o uso básico do componente **TextField** e seu comportamento com relação ao acionamento da tecla “Enter”:

```
// TextFieldDemo.java

import java.awt.*;
import java.awt.event.*;

public class TextFieldDemo extends Frame implements
ActionListener{
  private TextField tf1, tf2, tf3, tf4;
  private Button button1;

  public TextFieldDemo() {
    super("TextField Demo");
    setSize(250, 150);
    setLayout(new FlowLayout());
    // instanciacao, registro e adicao dos textfields
    tf1 = new TextField();           // textfield vazio
    add(tf1);
    tf2 = new TextField("", 20);     // textfield vazio de 20
colunas
    add(tf2);
    tf3 = new TextField("Hello!");   // textfield com texto
    add(tf3);
    tf4 = new TextField("Edite-me!", 30);// textfield com texto
e 30 cols
    add(tf4);
    tf4.addActionListener(this);
    // instanciacao, registro e adicao do botao
    button1 = new Button("tf3->tf2");
    button1.addActionListener(this);
    add(button1);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==button1) {
      tf2.setText(tf3.getText());
      tf3.setText("");
    } if (e.getSource()==tf4) {
      tf3.setText(tf4.getText());
      tf4.setText("");
    }
  }

  static public void main (String args[]) {
    TextFieldDemo f = new TextFieldDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
}
```

Exemplo 58 Classe TextFieldDemo

O texto digitado na última caixa de entrada de texto é automaticamente transferido para a terceira caixa de entrada quando acionamos a tecla “Enter” durante a edição do texto. Acionando-se o botão existente transferimos o texto da terceira caixa de entrada para a segunda, mostrando uma outra possibilidade. Em ambos os casos a caixa de texto de origem é limpa. Experimente compilar, executar e testar a aplicação, que exibe uma janela semelhante a ilustrada na Figura 29.

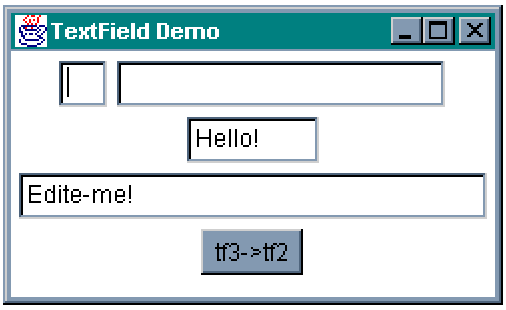


Figura 29 Aplicação TextFieldDemo

6.3.5 Panel

O componente **java.awt.Panel** ou Painel é um componente tipo *container*, ou seja, é um componente que permite dispor outros componente inclusive outros painéis., oferendo grande versatilidade na disposição de elementos na interface da aplicação. Como será visto em 6.4 Os Gerenciadores de Layout, cada painel pode assumir uma diferente disposição para seus componentes, ampliando as possibilidades de seu uso.

A classe **java.awt.Panel** possui dois construtores:

Método	Descrição
Panel()	Constrói um painel com o alinhamento *default* (**FlowLayout**).
Panel(LayoutManager)	Constrói um painel com o alinhamento especificado.

Tabela 26 Métodos da Classe java.awt.Panel

Além destes construtores, a classe **java.awt.Panel** compartilha os seguintes métodos com sua classe ancestral **java.awt.Container**.

Método	Descrição
add(Component)	Adiciona o componente especificado ao *container*.
add(Component, int)	Adiciona o componente especificado ao *container* na posição indicada.
doLayout()	Causa o rearranjo dos componentes do *container* segundo seu *layout*.
getComponent(int)	Obtêm uma referência para o componente indicado.
getComponentsCount()	Retorna o número de componentes do *container*.
getComponentAt(int, int)	Obtêm uma referência para o componente existente na posição dada.
getLayout()	Obtêm o gerenciador de *layout* para o *container*.
paint(Graphics)	Renderiza o *container*.
remove(int)	Remove o componente indicado.
removeAll()	Remove todo os componentes do *container*.
setLayout(LayoutManager)	Especifica o gerenciador de *layout* para o *container*.
updates(Graphics)	Atualiza a renderização do *container*.

Tabela 27 Métodos da Classe java.awt.Container

Da mesma forma que adicionamos componentes a um objeto **java.awt.Frame**, faremos isto aos **java.awt.Panel** pois ambas as classes são derivadas de

java.awt.Container. A seguir um exemplo de aplicação onde adicionamos três painéis a um **Frame** e componentes diferentes em cada um.

```
// PanelDemo.java

import java.awt.*;
import java.awt.event.*;

public class PanelDemo extends Frame
                       implements ActionListener {
  private Label l1, l2;
  private TextField entrada;
  private Button bLimpar, bTransf, bOk;
  private Panel pTop, pBot, pRight;

  // método main
  public static void main(String args[]) {
    PanelDemo f = new PanelDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public PanelDemo() {
    super("Panel Demo");
    setSize(400, 120);

    // instanciacao dos componentes
    l1 = new Label("Entrada");
    l2 = new Label("Saída");
    entrada = new TextField(20);
    bLimpar = new Button("Limpar");
    bLimpar.addActionListener(this);
    bTransf = new Button("Transferir");
    bTransf.addActionListener(this);
    bOk = new Button("Ok");
    bOk.addActionListener(this);
    pTop = new Panel(new FlowLayout(FlowLayout.LEFT));
    pTop.setBackground(Color.lightGray);
    pBot = new Panel(new GridLayout(1,2));
    pRight = new Panel();
    pRight.setBackground(Color.gray);

    // adicao dos componentes
    pTop.add(l1);
    pTop.add(entrada);
    add(pTop, BorderLayout.CENTER);
    pRight.add(l2);
    add(pRight, BorderLayout.EAST);
    pBot.add(bLimpar);
    pBot.add(BTransf);
    pBot.add(bOk);
    add(pBot, BorderLayout.SOUTH);
  }

  // interface ActionListener
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==bLimpar) {
      // limpa entrada
      entrada.setText("");
    } else if (e.getSource()==bTransf) {
      // entrada p/ saida
```

```
      l2.setText(entrada.getText());

    } else {
      // fechar aplicacao
      System.exit(0);
    }
  }
}
```

Exemplo 59 Classe PanelDemo

O resultado desta aplicação é ilustrado na Figura 30.

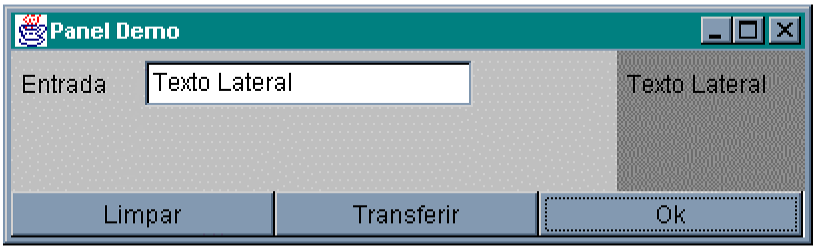


Figura 30 Aplicação PanelDemo

Note a distinção dos painéis através de suas diferentes cores. No painel superior esquerdo temos dois componente adicionados e arranjados através do gerenciador de *layout* **java.awt.FlowLayout** alinhado a esquerda. O painel superior direito usa o seu *layout default*. O painel inferior arranja seu componente como numa grade regular através do *layout* **java.awt.GridLayout**.

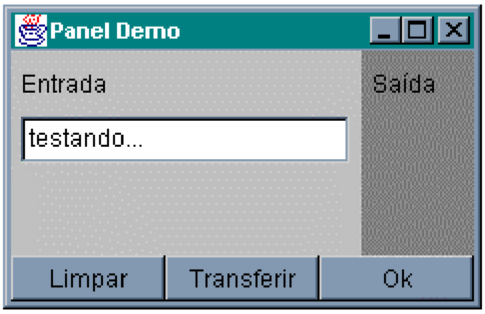


Figura 31 Aplicação PanelDemo Redimensionada

Se a janela da aplicação for redimensionada, notaremos que os botões permanecerão com o mesmo arranjo embora de tamanho diferente sendo que seus rótulos podem ser parcialmente ocultos caso tornem-se pequenos demais. No painel superior esquerdo os componentes serão arranjados em duas linhas caso tal painel se torne estreito. O outro painel não apresentará mudanças sensíveis exceto que o texto exibido pode ficar parcialmente oculto caso este painel também se torne por demais estreito.

Nos exemplos de aplicação dos próximo componentes a serem tratados iremos utilizar outros painéis para demostrar sua utilização.

6.3.6 TextArea

Este componente, conhecido como caixa de entrada multilinha ou *memo*, permite a criação de uma área para entrada e edição de texto contendo múltiplas linhas de forma a poder conter até mesmo mais texto do que possa ser exibido. Pertence a classe **java.awt.TextArea** e, como o componente **java.awt.TextField**, compartilha das características de seu componente ancestral **java.awt.TextComponent**, cujos principais métodos estão listados na Tabela 25.

O componente **TextArea** pode ser ajustado para permitir ou não a edição e para exibir barras de rolagem para auxiliar o controle da exibição do texto. A classe **java.awt.TextArea** contêm, entre outros, os seguintes construtores e métodos:

Método	**Descrição**
TextArea()	Constrói um componente sem texto com as duas barras de rolagem.
TextArea(int, int)	Constrói um componente sem texto capaz de exibir a quantidade de linhas e colunas de texto especificada com as duas barras de rolagem.
TextArea(String)	Constrói um componente contendo o texto especificado com as duas barras de rolagem.
TextArea(String, int, int)	Constrói um componente com texto dado capaz de exibir a quantidade de linhas e colunas de texto especificada com as duas barras de rolagem.
TextArea(String, int, int, int)	Constrói um componente com texto dado capaz de exibir a quantidade de linhas e colunas de texto especificada e também as barras de rolagem indicadas.
append(String)	Anexa o texto dado ao final do texto contido pelo componente.
getColumns()	Obtêm o número de colunas de texto.
getRows()	Obtêm o número de linhas de texto.
insert(String, int)	Insere o texto dado na posição indicada.
replaceRange(String, int, int)	Substitui o texto existente entre as posições indicadas pelo texto dado.
setColumns(int)	Especifica o número de colunas de texto.
setRows(int)	Especifica o número de linhas de texto.

Tabela 28 Métodos da Classe java.awt.TextArea

As exibição das barras de rolagem não pode ser modificada após a construção do componente. Sua especificação utiliza um construtor que aceita como parâmetro as constantes **TextArea.SCROLLBARS_BOTH**, **TextArea.SCROLLBARS_NONE**, **TextArea.SCROLLBARS_VERTICAL_ONLY**, **TextArea.SCROLLBARS_HORIZONTAL_ONLY**.

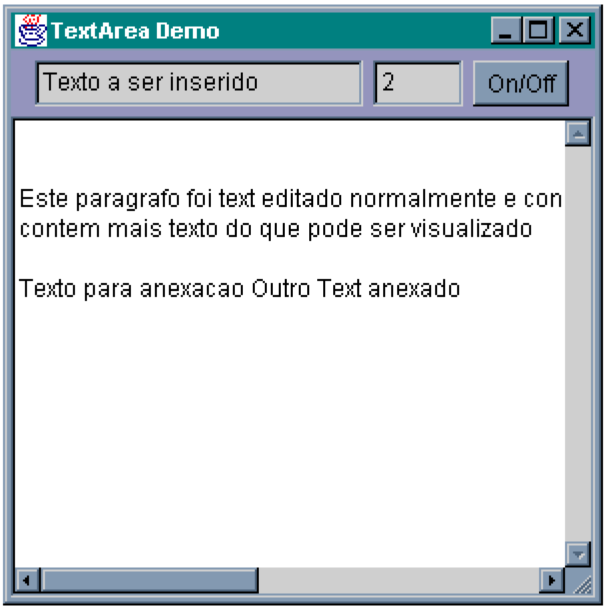
Abaixo um exemplo de utilização do componente **java.awt.TextArea**.

```
// TextAreaDemo.java

import java.awt.*;
import java.awt.event.*;

public class TextAreaDemo extends Frame
                          implements ActionListener {
  private TextArea taEditor;
  private Button bToggle;
  private TextField tfEntrada, tfPosicao;

  public static void main(String args[]) {
    TextAreaDemo f = new TextAreaDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  public TextAreaDemo() {
    super("TextArea Demo");
```

```
    setSize(300, 300);

    // Instancia Componentes
    tfEntrada = new TextField(20);
    tfEntrada.addActionListener(this);
    tfPosicao = new TextField(3);
    tfPosicao.addActionListener(this);
    bToggle = new Button("On/Off");
    bToggle.addActionListener(this);
    taEditor = new TextArea();

    // Cria Painel
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.setLayout(new FlowLayout());
      // Adiciona componentes ao painel
      p.add(tfEntrada);
      p.add(tfPosicao);
      p.add(bToggle);
    // Adiciona painel no topo do frame
    add(p, BorderLayout.NORTH);
    // Adiciona texarea no centro do frame
    add(taEditor, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==bToggle) {
      // Inverte habilitacao de edicao
      taEditor.setEditable(!taEditor.isEditable());
    } else if (e.getSource()==tfEntrada) {
      // anexa entrada ao texto
      taEditor.append(tfEntrada.getText());
      tfEntrada.setText("");
    } else if (e.getSource()==tfPosicao) {
      // insere entrada no texto na posicao indicada
      taEditor.insert(tfEntrada.getText(),
                      Integer.parseInt(tfPosicao.getText()));
      tfEntrada.setText("");
      tfPosicao.setText("");
    }
  }
}
```

Exemplo 60 Classe TextAreaDemo

Esta aplicação utiliza um componente **TextArea** para edição de texto, dois componentes **TextField** (uma para entrada de texto e outro para indicação de posição) e um componente **Button**. O componente **TextArea** pode ser usado livremente para edição suportando inclusive operações de recortar, copiar e colar (*cut, copy and paste*) sendo que tal capacidade de edição pode ser ativada e desativada através do botão rotulado "On/Off". As caixas de texto podem ser usadas para anexação de conteúdo (pressionando-se 'Enter" na maior delas) ou para inserção (pressionando-se 'Enter" na caixa de entrada menor quando esta contêm um valor inteiro). Na figura a seguir ilustra-se esta aplicação em execução:



Texto a ser inserido

2

Este paragrafo foi text editado normalmente e con
contem mais texto do que pode ser visualizado

Texto para anexacao Outro Text anexado

Figura 32 Aplicação TextAreaDemo

6.3.7 List

O componente **java.awt.List**, conhecido como caixa de lista ou *listbox*, permite a exibição de uma lista de itens com as seguintes características:

(i) os itens não podem ser editados diretamente pelo usuário,
(ii) uma barra de rolagem vertical é exibida automaticamente quando a caixa de lista contêm mais itens do que pode exibir;
(iii) pode ser configurado para permitir a seleção de um único item ou múltiplos itens e
(iv) quando se clica num item não selecionado este passa a estar selecionado e vice-versa.

Este componente basicamente produz dois tipos de eventos: **java.awt.event.ItemEvent** quando ocorre a seleção de um item e **java.awt.event.ActionEvent** quando o usuário dá um duplo clique sobre um item exibido pela lista. Estes eventos requerem tratamento por *listeners* diferentes respectivamente como as interfaces: **java.awt.event.ItemListener** e **java.awt.event.ActionListener**. Recomenda-se não utilizar-se o evento **ActionListener** quando a lista estiver operando em modo de múltipla seleção pois isto pode dificultar a interação do usuário.

A classe **java.awt.List** contêm, entre outros, os seguintes construtores e métodos:

Método	Descrição
List()	Constrói uma lista vazia operando em seleção única.
List(int)	Constrói uma lista vazia capaz de exibir a quantidade de itens indicada operando em seleção única.
List(int, Boolean)	Constrói uma lista vazia capaz de exibir a quantidade de itens indicada no modo de operação indicado.
add(String)	Adiciona o texto especificado como um novo item ao final da lista.
add(String, int)	Adiciona o texto especificado como um novo item na posição indicada na lista.

addActionListener(ActionEvent)	Registra uma classe *listener* (processadora de eventos) **ActionListener** para o componente.
addItemListener(ItemEvent)	Registra uma classe *listener* (processadora de eventos) **ItemListener** para o componente.
deselect(int)	Remove seleção do item indicado.
getItem(int)	Obtêm o item indicado.
getItemCount()	Obtêm a quantidade de itens da lista.
getSelectedIndex()	Obtêm a posição do item selecionado.
getSelectedItem()	Obtêm o item selecionado.
isIndexSelected(int)	Determina se o item especificado está selecionado.
isMultipleMode()	Determina se a lista está operando em modo de múltipla seleção.
remove(int)	Remove o item indicado da lista.
removeAll()	Remove todos os itens da lista.
select(int0	Seleciona o item indicado na lista.
setMultipleMode(Boolean)	Ativa ou não o modo de seleção múltipla.

Tabela 29 Métodos da Classe java.awt.List

No exemplo de aplicação utilizando uma caixa de lista desejamos usar uma caixa de entrada para adicionar novos itens ao componente lista enquanto um rótulo deve exibir o item selecionado, caso exista alguma seleção. Através de um duplo clique itens deve ser possível a remoção de itens da lista.

Para que ocorra a adição e remoção de itens da lista utilizaremos o evento **ActionEvent** processado pelo *listener* correspondente e que deve ser implementado através do método **actionPerformed**. Para que ocorra a exibição do item selecionado devemos monitorar o evento **ItemEvent**, cujo *listener* **ItemListener** deve ser implementado através do método **itemStateChanged**. Este último *listener*, embora receba um evento diferente, pode ser utilizado da mesma forma que até então fazíamos com o método **actionPerformed**.

Desta forma temos que a classe construída para nossa aplicação implementará simultaneamente duas interfaces: **ActionListener** e **ItemListener**. Se necessário, seria possível a implementação de outras interfaces pois o embora o Java não suporte a herança múltipla, é possível implementar-se múltiplas interfaces numa mesma classe, obtendo-se resultados semelhantes mas com menor complexidade.

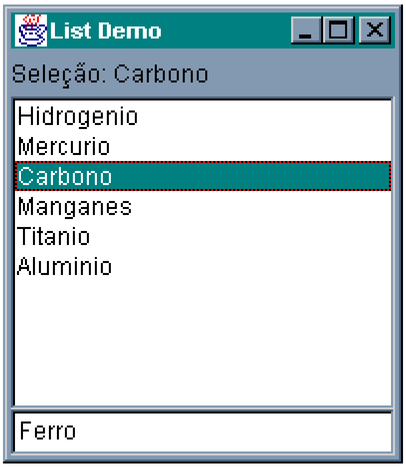
O código do exemplo proposto de utilização do componente **java.awt.List** é listado a seguir.

```
// ListDemo.java

import java.awt.*;
import java.awt.event.*;

public class ListDemo extends Frame
                      implements ActionListener, ItemListener {
  private List lista;
  private TextField tfEntrada;
  private Label lbExibicao;

  // método main
  public static void main(String args[]) {
    ListDemo f = new ListDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
```

```
  // construtor
  public ListDemo() {
    super("List Demo");
    setSize(200, 300);

    // instanciacao dos componentes
    tfEntrada = new TextField(20);
    tfEntrada.addActionListener(this);
    lbExibicao = new Label("Sem Seleção");
    lbExibicao.setBackground(SystemColor.control);
    lista = new List(6);
    lista.addActionListener(this);
    lista.addItemListener(this);

    // adicao componentes ao frane
    add(lbExibicao, BorderLayout.NORTH);
    add(lista, BorderLayout.CENTER);
    add(tfEntrada, BorderLayout.SOUTH);
  }

  // interface ActionListener
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==tfEntrada) {
      // inclui entrada na lista
      lista.add(tfEntrada.getText());
      tfEntrada.setText("");
    } else if (e.getSource()==lista) {
      // remove item indicado
      tfEntrada.setText(lista.getSelectedItem());
      lista.remove(lista.getSelectedIndex());
      lbExibicao.setText("Sem Seleção");
    }
  }

  // interface ItemListener
  public void itemStateChanged(ItemEvent e) {
    if (e.getSource()==lista) {
      // testa se existe selecao
      if (lista.getSelectedIndex()>-1)
        // exibe item selecionado
        lbExibicao.setText("Seleção: "+lista.getSelectedItem());
      else
        // limpa exibicao
        lbExibicao.setText("Sem Seleção");
    }
  }
}
```

Exemplo 61 Classe ListDemo

Quando executada a aplicação possui a seguinte aparência:



Seleção: Carbono

Hidrogenio
Mercurio
Carbono
Manganes
Titanio
Aluminio

Ferro

Figura 33 Aplicação ListDemo

6.3.8 Choice

O componente **java.awt.Choice** implementa uma lista de itens onde somente o item selecionado é exibido. A lista, exibida como um menu suspenso (*pop-down menu*) pode ser vista através do acionamento de um botão integrado ao componente, por isto é conhecido também como caixa de seleção ou *combobox*. Da mesma forma que no componente **java.awt.List**, uma barra de rolagem vertical é automaticamente exibida quando a lista não pode mostrar simultaneamente todos os itens que contêm. A seleção opera apenas em modo simples, ou seja, apenas um item pode ser selecionado de cada vez, sendo que a escolha de um item não selecionado o seleciona e vice-versa.

Este componente gera apenas eventos do tipo **ItemEvent**, requerendo assim a implementação da interface **ItemListener** para seu processamento.

A Tabela 30 contêm o construtores e os principais métodos disponíveis na classe **java.awt.Choice**:

Método	**Descrição**
Choice()	Constrói uma caixa de seleção.
add(String)	Adiciona o texto especificado como um novo item ao final da lista.
addItemListener(ItemEvent)	Registra uma classe *listener* (processadora de eventos) **ItemListener** para o componente.
getItem(int)	Obtêm o item indicado.
getItemCount()	Obtêm a quantidade de itens da lista.
getSelectedIndex()	Obtêm a posição do item selecionado.
getSelectedItem()	Obtêm o item selecionado.
insert(String, int)	Insere o item dado na posição especificada.
remove(int)	Remove o item indicado da lista.
removeAll()	Remove todos os itens da lista.
select(int0	Seleciona o item indicado na lista.

Tabela 30 Métodos da Classe java.awt.Choice

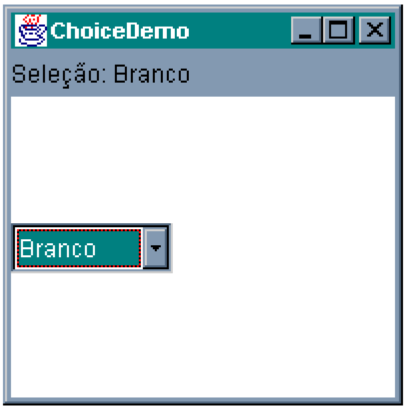
É dado a seguir um exemplo de aplicação da caixa de seleção onde esta é programaticamente populada com o nome de algumas cores sendo que sua seleção altera a cor de fundo da janela para a cor correspondente. Observe que a única interface implementada é a **ItemListener**.

```
// ChoiceDemo.java

import java.awt.*;
```

```
import java.awt.event.*;

public class ChoiceDemo extends Frame
                        implements ItemListener {
  private Choice combo;
  private Label lbExibicao;

  // método main
  public static void main(String args[]) {
    ChoiceDemo f = new ChoiceDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public ChoiceDemo() {
    super("ChoiceDemo");
    setSize(200, 200);

    // instanciacao dos componentes
    combo = new Choice();
    combo.add("Branco");
    combo.add("Vermelho");
    combo.add("Azul");
    combo.add("Verde");
    combo.addItemListener(this);
    lbExibicao = new Label("Seleção: Branco");
    lbExibicao.setBackground(SystemColor.control);

    // adicao componentes ao frane
    add(lbExibicao, BorderLayout.NORTH);
    add(combo, BorderLayout.WEST);
  }

  // interface ItemListener
  public void itemStateChanged(ItemEvent e) {
    if (e.getSource()==combo) {
      // testa se existe selecao
      if (combo.getSelectedIndex()>-1) {
        // exibe item selecionado
        lbExibicao.setText("Seleção: "+combo.getSelectedItem());
        switch(combo.getSelectedIndex()) {
          case 0: setBackground(Color.white);
            break;
          case 1: setBackground(Color.red);
            break;
          case 2: setBackground(Color.blue);
            break;
          case 3: setBackground(Color.green);
            break;
        }
        repaint();
      } else
        // limpa exibicao
        lbExibicao.setText("Sem Seleção");
    }
  }
}
```

Exemplo 62 Classe ChoiceDemo

Quando executada a aplicação se mostra da seguinte forma. Note que por *default* o primeiro item adicionado torna-se o item selecionado até ocorrer outra seleção por parte do usuário ou uso do método **select**.



Seleção: Branco

Figura 34 Aplicação ChoiceDemo

6.3.9 CheckBox

A caixa de opção, como também é conhecido o componente **java.awt.Checkbox**, é um componente utilizado para representar graficamente uma opção que pode ser ligada (*on* = *true*) ou desligada (*off* = *false*). É geralmente utilizada para exibir um grupo de opções as quais podem ser selecionadas independentemente pelo usuário, ou seja, permitem múltiplas seleções.

Assim como o componente **java.awt.Choice**, o **Checkbox** gera apenas eventos do tipo **ItemEvent**, requisitando a implementação da interface **ItemListener** nas aplicações que pretendam utilizá-lo. Na tabela a seguir encontram-se os principais construtores e métodos desta classe.

Método	Descrição
Checkbox(String)	Cria um **Checkbox** com o texto dado.
CheckBox(String, Boolean)	Cria um **Checkbox** com o texto dado no estado especificado.
addItemListener(ItemEvent)	Registra uma classe *listener* (processadora de eventos) **ItemListener** para o componente.
getCheckBoxGroup(CheckboxGroup)	Obtêm o grupo ao qual o componente **Checkbox** está associado.
getLabel()	Obtêm o texto associado ao componente.
getState()	Obtêm o estado do componente.
setCheckBoxGroup(CheckboxGroup)	Especifica o grupo ao qual o componente **Checkbox** está associado.
setLabel(String)	Especifica o texto associado ao componente.
setState(Boolean)	Especifica o estado do componente.

Tabela 31 Métodos da Classe java.awt.Checkbox

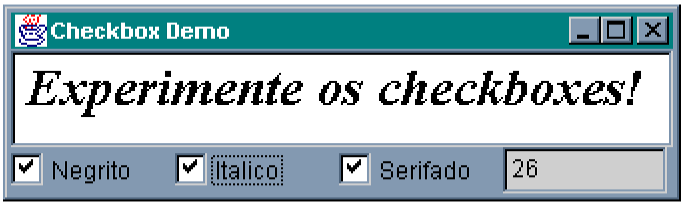
Abaixo um exemplo de aplicação do componente **Checkbox**. Três destes componentes são utilizado para especificar o efeito negrito (*bold*) e itálico (*italic*) do texto e tipo do fonte (serifado ou não serifado) exibido por um rótulo. Uma caixa de entrada é usada para especificar o tamanho do fonte. Desta forma temos que as interfaces

ActionListener e **ItemListener** são implementadas nesta classe para prover suporte aos dois tipos de componente empregados.

```
// CheckboxDemo.java

import java.awt.*;
import java.awt.event.*;

public class CheckboxDemo extends Frame
                        implements ActionListener,ItemListener {
  private Checkbox cbBold, cbItalic, cbSerif;
  private TextField tfExibicao, tfSize;

  // método main
  public static void main(String args[]) {
    CheckboxDemo f = new CheckboxDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public CheckboxDemo() {
    super("Checkbox Demo");
    setSize(300, 100);

    // instanciacao dos componentes
    cbBold = new Checkbox("Negrito", false);
    cbBold.addItemListener(this);
    cbItalic = new Checkbox("Italico", false);
    cbItalic.addItemListener(this);
    cbSerif = new Checkbox("Serifado", false);
    cbSerif.addItemListener(this);
    tfExibicao = new TextField("Experimente os checkboxes!");
    tfSize = new TextField("12");
    tfSize.addActionListener(this);

    // adicao componentes ao frame
    add(tfExibicao, BorderLayout.CENTER);
    Panel p = new Panel();
    p.setLayout(new GridLayout(1, 4));
    p.setBackground(SystemColor.control);
    p.add(cbBold);
    p.add(cbItalic);
    p.add(cbSerif);
    p.add(tfSize);
    add(p, BorderLayout.SOUTH);
  }

  // interface ActionListener
  public void actionPerformed(ActionEvent e) {
    changeFont();
  }

  // interface ItemListener
  public void itemStateChanged(ItemEvent e) {
    changeFont();
  }

  private void changeFont() {
      // testa estado de cada checkbox
    int negr = (cbBold.getState() ? Font.BOLD : Font.PLAIN);
    int ital = (cbItalic.getState() ? Font.ITALIC : Font.PLAIN);
```

```
    int size = Integer.parseInt(tfSize.getText());
    if (cbSerif.getState())
      tfExibicao.setFont(new Font("Serif",negr+ital,size));
    else
      tfExibicao.setFont(new Font("SansSerif",negr+ital,size));
  }
}
```

Exemplo 63 Classe CheckboxDemo



Experimente os checkboxes!

☑ Negrito ☑ Italico ☑ Serifado 26

Figura 35 Aplicação CheckBoxDemo

Note que vários **Checkbox** podem estar ativos ao mesmo tempo.

6.3.10 CheckBoxGroup

Como visto, quando são adicionado vários componente **java.awt.Checkbox** a uma aplicação, é possível selecionar-se um, vários ou mesmo todos os componentes, caracterizando uma situação de múltipla escolha, o que nem sempre é apropriado. Quando desejamos que seja feita uma única escolha dentre um conjunto, necessitamos de um componente com outro componente.

A AWT, ao invés de oferecer um segundo componente permite que vários componentes **java.awt.Checkbox** sejam associados através de um outro denominado **java.awt.CheckboxGroup**. O **CheckboxGroup** não é um componente visual e apenas proporciona a associação de vários **Checkbox** de forma que se comportem como componentes conhecidos como botões de opções ou *radiobuttons*. Os botões de opção associados garantem que apenas um esteja seleciona em qualquer instante, assim quando um deles é selecionado qualquer outro que estivesse selecionado é de-selecionado, garantindo uma escolha única dentre o conjunto de opções apresentadas ao usuário.

Na Tabela 32 temos os principais métodos da classe **java.awt.CheckboxGroup**.

Método	Descrição
CheckboxGroup()	Cria um grupo de **Checkboxes.**
getSelectedCheckbox()	Obtêm o componente **Checkbox** do grupo correntemente selecionado.
setSelectedCheckbox(Checkbox)	Determina qual o componente **Checkbox** do grupo selecionado.

Tabela 32 Métodos da Classe java.awt.CheckboxGroup

A utilização de um **CheckboxGroup** requer que os eventos **ItemEvent** associados a cada um de seus **CheckBox** associados sejam monitorados através da implementação da interface **ItemListener**. Segue um exemplo de aplicação deste componente:

```
// CheckboxGroupDemo.java

import java.awt.*;
import java.awt.event.*;

public class CheckboxGroupDemo extends Frame
                               implements ItemListener {
  private Checkbox cbWhite, cbRed, cbBlue, cbGreen;
  private CheckboxGroup cbg;
```

```
  // método main
  public static void main(String args[]) {
    CheckboxGroupDemo f = new CheckboxGroupDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public CheckboxGroupDemo() {
    super("CheckboxGroup Demo");
    setSize(200, 200);

    // instanciacao dos componentes
    cbg = new CheckboxGroup();
    cbWhite = new Checkbox("Branco", cbg, true);
    cbRed = new Checkbox("Vermelho", cbg, false);
    cbBlue = new Checkbox("Azul", cbg, false);
    cbGreen = new Checkbox("Verde", cbg, false);
    cbWhite.addItemListener(this);
    cbRed.addItemListener(this);
    cbBlue.addItemListener(this);
    cbGreen.addItemListener(this);

    // adicao componentes ao frame
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.setLayout(new GridLayout(4,1));
    p.add(cbWhite);
    p.add(cbRed);
    p.add(cbBlue);
    p.add(cbGreen);
    add(p, BorderLayout.WEST);
  }

  // interface ItemListener
  public void itemStateChanged(ItemEvent e) {
    if (e.getSource()==cbWhite) {
      setBackground(Color.white);
    } else if (e.getSource()==cbRed) {
      setBackground(Color.red);
    } else if (e.getSource()==cbBlue) {
      setBackground(Color.blue);
    } else {
      setBackground(Color.green);
    }
    repaint();
  }
}
```

Exemplo 64 Classe CheckboxGroupDemo

Nesta aplicação utilizamos quatro componente **CheckBox** associados por um intermédio de um componente **CheckBoxGroup**. A seleção de um dos **Checkbox** efetua a troca da cor de fundo do **Frame**.

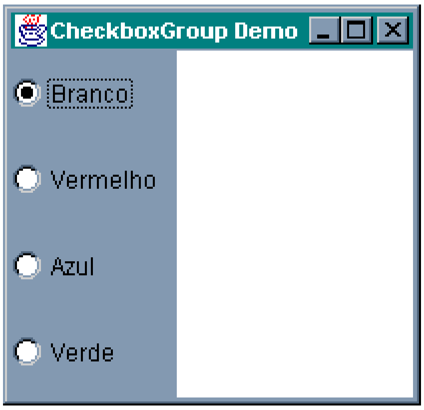


Figura 36 Aplicação CheckboxGroupDemo

Note que apenas um dos **Checkbox** existentes permanece selecionado quando associados, garantindo uma única escolha.

6.3.11 Image

Diferentemente do que poderia esperar, a classe abstrata **java.awt.Image** não é um componente visual tão pouco pertence a hierarquia de componentes ilustrada da Figura 25, mas é uma classe que serve de base para a implementação de outros componentes que realizem a representação de imagens gráficas. Isto acontece porque a representação de imagens depende de detalhes específicos de cada plataforma, de forma que para cada uma sejam implementadas de forma diferente.

Ainda assim esta classe apresenta alguns métodos de interesse, como listados abaixo:

Método	Descrição
flush()	Libera todos os recursos utilizados por uma imagem.
getGraphics()	Obtêm um contexto gráfico para renderização *off-screen*.
getHeight(ImageObserver)	Obtêm a altura da imagem se conhecida.
getWidth(ImageObserver)	Obtêm a largura da imagem se conhecida.

Tabela 33 Métodos da Classe java.awt.Image

Note que alguns métodos recebem como parâmetro um objeto **ImageObserver** que na verdade é uma interface que define que será notificado quanto a atualizações da imagem. No caso de imagens obtidas através da Internet, a carga destas pode ser demorada, sendo que a classe que implementa um **ImageObserver** é notificada quanto a necessidade de atualização da exibição da imagem. Como a classe **Component** implementa esta interface, qualquer componente da AWT pode ser o responsável por manter a imagem atualizada. Recomenda-se que o componente *container* da imagem seja tal responsável.

Usualmente esta classe pode ser utilizada para visualizar arquivos GIF (*Graphics Interchange Format*) ou JPG (*Joint Photographic Experts Group*).Uma forma de obter uma imagem é utilizando o método **getImage** disponível no objeto **Toolkit** que encapsula os métodos específicos da plataforma utilizada, como abaixo:

```
Image image;
String nomeArquivo;
  ⋮
image = Toolkit.getDefaultToolkit().getImage(nomeArquivo);
```

O nome do arquivo pode conter um nome de arquivo simples (que será procurado no diretório corrente da aplicação) ou nome de arquivo que inclui uma especificação de caminho absoluto ou relativo.

Obtida a imagem, devemos decidir onde será exibida. Para isto devemos sobrepor o método **paint** do componente que servirá *container* para imagem, ou seja, se desejamos exibir a imagem numa janela, utilizamos o método **paint** do objeto **Frame** associado. Se a imagem será exibida num painel dentro de uma janela, devemos utilizar o método **paint** do painel, o que exigirá a construção uma classe em separado pois não podemos sobrepor o método **paint** de um painel a partir da classe que o contem. Em ambos os casos, a imagem poderá ser efetivamente renderizada através do método **drawImage** que pertence a classe **java.awt.Graphics** como exemplificado a seguir.

```
public void paint(Graphics g) {
  ⋮
  g.drawImage(image, posicaoX, posicaoY, this);
  ⋮
}
```

Neste exemplo o método **drawImage** do contexto gráfico do *container* recebe como parâmetros o objeto imagem contendo a imagem a ser desenhada, as posições x e y da origem da imagem dentro do *container* e o **ImageObserver**, no caso o próprio *container*. A imagem deve ser obtida em outro ponto do código, por exemplo no construtor da classe ou na implementação de algum método pertencente a um *listener* como resposta a ações do usuário.

No Exemplo 65 temos a exibição de uma imagem diretamente no **Frame** da aplicação. O nome do arquivo da imagem é especificado pelo usuário através de um componente **TextField**.

```
// ImageDemo.java

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

public class ImageDemo extends Frame
                       implements ActionListener {

  private TextField tfArquivo;
  private Image image;

  public static void main(String args[]) {
    ImageDemo f = new ImageDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  public ImageDemo() {
    super("Image Demo");
    setSize(300, 300);
    setBackground(SystemColor.lightGray);

    // instanciando componente
    tfArquivo = new TextField();
    tfArquivo.addActionListener(this);

    // Adicionando compomente
    add(tfArquivo, BorderLayout.SOUTH);
  }

  public void actionPerformed(ActionEvent e) {
    if (image != null)
      image.flush();
    image =
Toolkit.getDefaultToolkit().getImage(tfArquivo.getText());
```

```
    repaint();
  }

  public void paint(Graphics g) {
    if (image != null) {
      Insets i = getInsets();
      g.drawImage(image, i.left, i.top, this);
    }
  }
}
```

Exemplo 65 Classe ImageDemo

No método **actionPerformed**, onde obtêm a imagem especificada pelo usuário, note o uso do método **flush** da classe **java.awt.Image** que libera os recursos utilizados por imagens previamente carregadas.

No método **paint** só ocorre a renderização se o objeto imagem é válido. Ainda no método **paint** desta classe note o uso de um objeto **java.awt.Insets**. Através do método **getInsets**, que retorna o um objeto **Insets** associado ao **Frame**, obtêm-se a altura da barra de títulos e espessura das bordas da janela, possibilitando renderizar a imagem dentro do espaço útil da janela da aplicação.

\\Peter\DAI\PMP\Papers\atomic-coffee.gif

Figura 37 Aplicação ImageDemo

6.3.12 Canvas

O componente **java.awt.Canvas** é uma área retangular destinada a suportar operações de desenho definidas pela aplicação assim como monitorar eventos de entrada realizados pelo usuário. Este componente não encapsula outra funcionalidade exceto a de oferecer suporte para renderização, constituindo uma classe base para criação de novos componentes. Esta classe oferece um construtor e dois métodos dos quais selecionamos:

Método	Descrição
Canvas()	Cria um novo objeto **Canvas**.
paint(Graphics)	Renderiza o objeto **Canvas** através de seu contexto gráfico.

Tabela 34 Métodos da Classe java.awt.Canvas

Note que os eventos gerados por este objeto são os mesmos produzidos pela classe **java.awt.Component** (veja Tabela 18 e Tabela 19), basicamente **KeyEvent**, **MouseEvent** e **MouseMotionEvent** que podem ser monitorados pelos seus respectivos *listeners*.

Como exemplo criaremos um novo componente do tipo *lightweight* baseado na classe **java.awt.Canvas**. Este componente nos oferecerá a possibilidade de adicionarmos imagens às nossas aplicações como componentes comuns e denominaremos sua classe de **Picture**. Basicamente estenderemos a classe **Canvas** de modo que seu construtor receba o nome do arquivo que contém a imagem ajustando o tamanho do componente sendo que o método **paint** se encarregará de renderizar a imagem tal como feito no Exemplo 65.

O diagrama de classes para o nosso componente é ilustrado na Figura 38 enquanto que o código para sua classe figura no

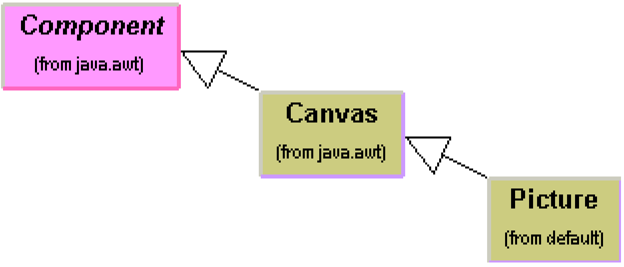


Figura 38 Hierarquia da Classe Picture

```
// Picture.java

import java.awt.*;
import java.awt.event.*;

public class Picture extends Canvas {
  private Image image;
  private int iHeight, iWidth;

  // Construtor Default
  // Componente assume o tamanho da imagem
  public Picture (String imageName) {
    this(imageName, 0, 0);
  }

  // Construtor Parametrizado
  // Componente assume o tamanho fornecido
  public Picture(String imageName, int width, int height) {
    image = Toolkit.getDefaultToolkit().getImage(imageName);
    MediaTracker  tracker = new MediaTracker(this);
    tracker.addImage(image, 0);
    try {
      tracker.waitForID(0);
    } catch (Exception e) {
      e.printStackTrace();
    }
    iWidth = (width==0 ? image.getWidth(this) : width);
    iHeight = (height==0 ? image.getHeight(this) : height);
    setBackground(SystemColor.control);
    setSize(iWidth, iHeight);
  }

  public void paint (Graphics g) {
    // renderiza imagem com ampliacao/reducao dada
    g.drawImage(image, 0, 0, iWidth, iHeight, this);
  }
}
```

Figura 39 Classe Picture

A aplicação que construiremos apenas adiciona três componentes **Picture** a um **Frame** exibindo suas possibilidades básicas.

```
// CanvasDemo.java

import java.awt.*;
import java.awt.event.*;

public class CanvasDemo extends Frame {
  private Picture normal, bigger, smaller;

  // método main
  public static void main(String args[]) {
    CanvasDemo f = new CanvasDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public CanvasDemo() {
    super("Canvas Demo");
    setSize(250, 170);
    setLayout(new FlowLayout());
    smaller = new Picture("cupHJbutton.gif", 32, 32);
    normal = new Picture("cupHJbutton.gif");
    bigger = new Picture("cupHJbutton.gif", 128, 128);
    add(smaller);
    add(normal);
    add(bigger);
  }
}
```

Exemplo 66 Classe CanvasDemo

O resultado desta aplicação é colocado a seguir.

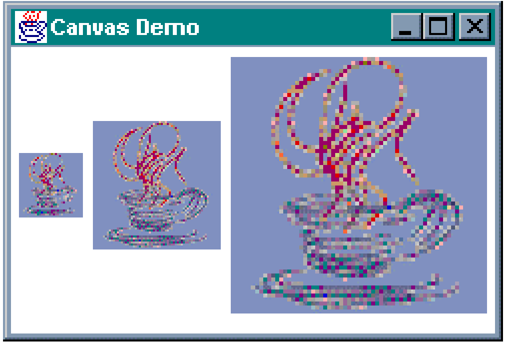


Figura 40 Aplicação CanvasDemo

6.4 Os Gerenciadores de Layout

No desenvolvimento de aplicações gráficas a aparência final da aplicação é de grande importância, pois além dos aspectos estéticos, a conveniente organização dos componentes reduz a quantidade de movimentos com o mouse ou agiliza a navegação entre componentes utilizando o teclado.

Quando criamos uma aplicação Java utilizamos componentes tipo *container* para dispor os demais componentes de uma aplicação. Tais *containers* são o **java.awt.Frame** e o **java.awt.Panel**. Como pudemos observar nos exemplos anteriores, ao invés de especificarmos o posicionamento dos componentes em termos de coordenadas x e y relativas a borda esquerda (*left*) e borda superior (*top*) do *container* onde são adicionados, o

Java oferece um esquema simplificado mas muito versátil para o posicionamento dos componentes. Este esquema é baseado no uso dos gerenciadores de *layout* ou *Layout Managers*.

Os *Layout Manager* são classes que implementam interfaces específicas que determinam como os componentes de um dado *container* serão arranjados, ou seja, indicam como um componente do tipo *container* deve organizar a distribuição e posicionamento dos componentes a ele adicionados.

A AWT nos oferece cinco *Layout Managers* que são:

(i) **FlowLayout**,
(ii) **GridLayout,**
(iii) **BorderLayout**,
(iv) **CardLayout** e
(v) **GridBagLayout**.

Na Figura 41 temos a hierarquia de classes dos principais *Layout Managers* e as interfaces relacionadas.

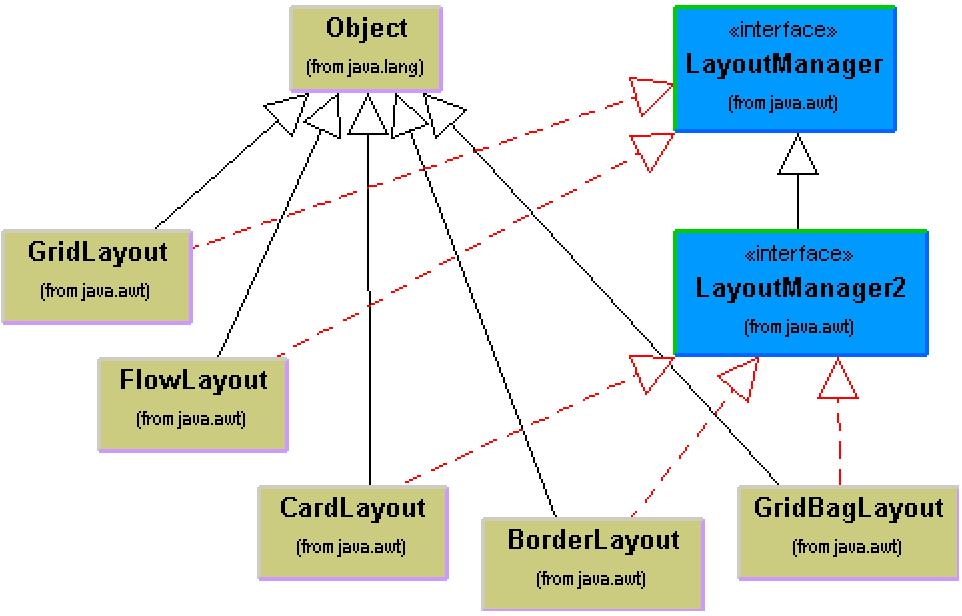


Figura 41 Hierarquia de Classe dos LayoutManagers

Trataremos a seguir dos quatro primeiros gerenciadores de *layout* citados, isto é, do **FlowLayout**, **GridLayout**, **BorderLayout** e **CardLayout**.

6.4.1 FlowLayout

O gerenciador mais simples é o **FlowLayout** *manager* que é o padrão para *applets* e painéis. Neste esquema os componentes são arranjados da esquerda para direita na ordem em que são adicionados ao *container*. Não existindo mais espaço na linha formada pelos componentes adicionados é criada outra linha abaixo desta que segue o mesmo critério de formação. Desta forma este gerenciador de *layout* distribui os componentes em linhas tal como os editores de texto fazem com palavras de um parágrafo. Abaixo um exemplo:

```
// FlowLayoutDemo.java

import java.awt.*;

public class FlowLayoutDemo extends Frame {

  // método main
  public static void main(String args[]) {
```

```
    FlowLayoutDemo f = new FlowLayoutDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public FlowLayoutDemo() {
    super("FlowLayout Demo");
    setSize(200, 200);
    setLayout(new FlowLayout());
    for (int i=0; i<9; i++)
      add(new Button("Botão "+(i+1)));
  }
}
```

Exemplo 67 Classe FlowLayoutDemo

O método **main** instancia e exibe o primeiro **Frame** da aplicação. No construtor da classe temos a determinação do título e do tamanho da janela assim como a especificação do **FlowLayout** como gerenciador de *layout* a ser utilizado. Finalmente são instanciados e adicionados nove botões sem funcionalidade ao **Frame** (sem a manutenção de referências e sem a adição de *listeners*). Abaixo temos o resultado exibido pela aplicação. Experimente redimensionar a janela e observe o efeito obtido.

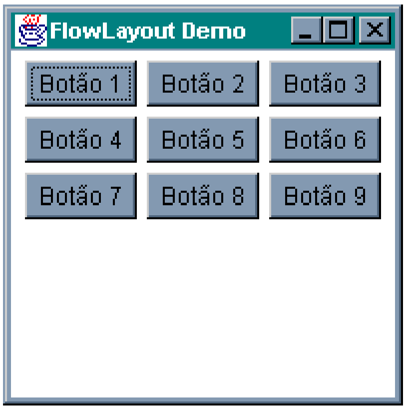


Figura 42 Aplicação FlowLayoutDemo

A classe **java.awt.FlowLayout** contêm os seguintes construtores e métodos:

Método	Descrição
FlowLayout()	Cria um gerenciador **FlowLayout** padrão: alinhamento centralizado, espaçamento horizontal e vertical de 5 unidades.
FlowLayout(int)	Cria um gerenciador **FlowLayout** com o alinhamento dado e espaçamento horizontal e vertical de 5 unidades.
FlowLayout(int, int, int)	Cria um gerenciador **FlowLayout** com o alinhamento, espaçamento horizontal e vertical dados.
getAlignment()	Obtêm o alinhamento deste *layout*.
getHgap()	Obtêm o espaçamento horizontal.
getVgap()	Obtêm o espaçamento vertical.
setAlignment(int)	Especifica o alinhamento deste *layout*.
setHgap(int)	Especifica o espaçamento horizontal.
setVgap(int)	Especifica o espaçamento vertical.

Tabela 35 Métodos da Classe java.awt.FlowLayout

Dispõe-se também das constantes **FlowLayout.LEFT**, **FlowLayout.CENTER** e **FlowLayout.RIGHT** quem podem ser utilizadas para especificar ou determinar o alinhamento deste *layout manager*.

6.4.2 GridLayout

O **GridLayout** *manager* permite arranjar os componentes numa grade retangular onde todas as células da grade possuem o mesmo tamanho de forma que toda a área do *container* seja ocupada. O número de linhas e colunas pode ser especificado, de modo que neste esquema a medida em que os componentes são adicionados estes vão sendo arranjados nas células preenchendo-as em ordem, isto é, da primeira para a última.

Os componentes adicionados aos *containers* gerenciados por este *layout* são expandidos de forma a ocupar todo o espaço da célula em que são adicionados. Quaisquer modificações no tamanho do *container* são propagadas para os componentes, isto é, a grade é redefinida e os componentes são redimensionados e rearranjados mantendo a organização de grade.

Abaixo um exemplo simples de aplicação cujo *layout* é ajustado para o **GridLayout** sendo adicionados nove botões sem funcionalidade, tal qual a classe apresentada no Exemplo 67:

```
// GridLayoutDemo.java

import java.awt.*;

public class GridLayoutDemo extends Frame {

  // método main
  public static void main(String args[]) {
    GridLayoutDemo f = new GridLayoutDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public GridLayoutDemo() {
    super("GridLayout Demo");
    setSize(200, 200);
    setLayout(new GridLayout(3, 3));
    for (int i=0; i<9; i++)
      add(new Button("Botão "+(i+1)));
  }
}
```

Exemplo 68 Classe GridLayoutDemo

O método **main**, como nas aplicações anteriores, serve para instanciar e exibir primeiro **Frame** correspondente ao objeto que encapsula a aplicação desejada. O construtor da classe determina o título a ser exibido pela janela, especificando o tamanho desta e o novo gerenciador de *layout* a ser utilizado. Finalmente são instanciados e adicionados os botões ao **Frame**, sem a necessidade de manter-se referências para estes e sem a adição de *listeners* pois da mesma forma que no exemplo anterior não terão funcionalidade associada.

Na figura seguinte temos o resultado apresentado pela aplicação. Redimensione a janela observando o efeito provocado comparando também com o resultado da aplicação ilustrada na Figura 42.

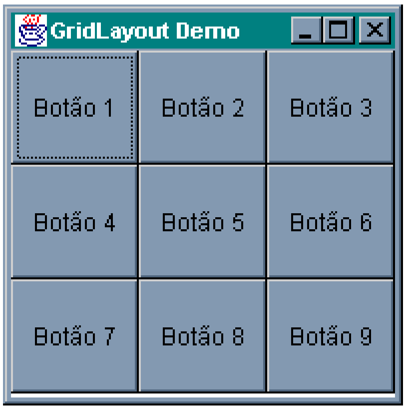


Figura 43 Aplicação GridLayoutDemo

A classe **java.awt.GridLayout** contêm os seguintes construtores e métodos que podem utilizados para customizar este gerenciador:

Método	**Descrição**
GridLayout()	Cria um gerenciador **GridLayout** padrão com uma linha e uma coluna.
GridLayout(int, int)	Cria um gerenciador **GridLayout** com o número de linhas e colunas especificado.
GridLayout(int, int, int, int)	Cria um gerenciador **GridLayout** com o número de linhas e colunas especificado e com o alinhamento, espaçamento horizontal e vertical dados.
getColumns()	Obtêm o número de colunas do *layout*.
getHgap()	Obtêm o espaçamento horizontal.
getRows()	Obtêm o número de linhas do *layout*.
getVgap()	Obtêm o espaçamento vertical.
setColumns()	Especifica o número de colunas do *layout*.
setHgap(int)	Especifica o espaçamento horizontal.
setRows()	Especifica o número de linhas do *layout*.
setVgap(int)	Especifica o espaçamento vertical.

Tabela 36 Métodos da Classe java.awt.GridLayout

6.4.3 BorderLayout

O gerenciador de *layout* **java.awt.BorderLayout** administra a disposição de componentes de um *container* em cinco regiões distintas que são: superior (*north*), inferior (*south*), esquerda (*west*), direita (*east*) e central (*center*). A adição de um componente deve ser explicitamente realizada, não importando a ordem com que isto é feito pois a organização das regiões é fixa. Cada região só pode ser ocupada por um único componente, de forma que se um segundo componente é adicionado num certa região o primeiro componente torna-se obscurecido não podem ser visualizado ou acessado. Os componentes adicionados são redimensionados de forma a ocuparem toda a região onde foram inseridos. Abaixo uma ilustração da distribuição das regiões:

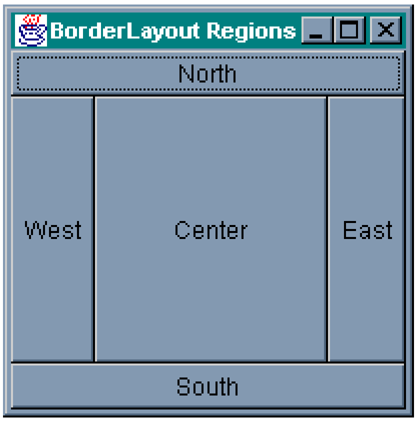


Figura 44 Regiões do java.awt.BorderLayout

Isto sugere que as regiões sejam adequadamente ocupadas por painéis de forma que nestes possam ser distribuídos vários outros componentes que podem assim serem organizados independentemente em cada painel. Note ainda que o **BorderLayout** é o gerenciador de *layout* padrão para os **Frames**.

A seguir um exemplo de aplicação onde dispomos alguns componentes aproveitando as facilidades oferecidas pelo **BorderLayout**.

```
// BorderLayoutDemo.java

import java.awt.*;

public class BorderLayoutDemo extends Frame {

  private TextArea taEditor;
  private Button bAbrir, bSalvar, bFechar;

  // método main
  public static void main(String args[]) {
    BorderLayoutDemo f = new BorderLayoutDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public BorderLayoutDemo() {
    super("BorderLayout Demo");
    setSize(300, 200);
    // Painel Topo
    Panel p1 = new Panel();
    p1.setLayout(new FlowLayout(FlowLayout.LEFT));
    p1.setBackground(SystemColor.control);
    for (int i=0; i<4; i++)
      p1.add(new Picture("cupHJbutton.gif", 16, 16));
    add("North", p1);
    // Painel Lateral
    p1 = new Panel();
    p1.setBackground(SystemColor.control);
    Panel p2 = new Panel();
    p2.setBackground(SystemColor.control);
    p2.setLayout(new GridLayout(3, 1, 5, 5));
    p2.add(bAbrir = new Button("Abrir"));
    p2.add(bSalvar = new Button("Salvar"));
    p2.add(bFechar = new Button("Fechar"));
    p1.add(p2);
    add("East", p1);
```

```
        add("Center", taEditor = new TextArea());
    }
}
```

Exemplo 69 Classe BorderLayoutDemo

Executando-se esta aplicação temos o seguinte resultado.

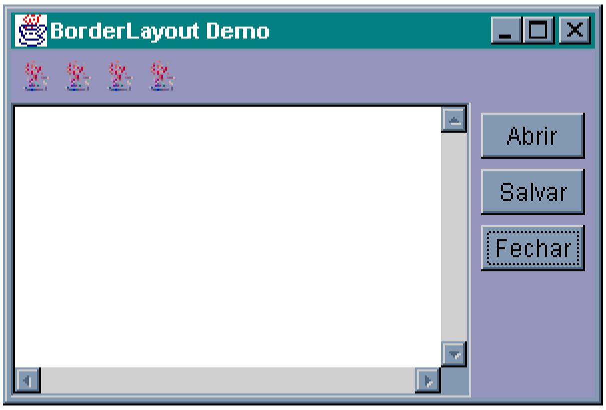


Figura 45 Aplicação BorderLayoutDemo

A classe **java.awt.BorderLayout** contêm os seguintes construtores e métodos que podem utilizados para customizar este gerenciador:

Método	Descrição
BorderLayout()	Cria um novo gerenciador de *layout* sem espaçamento entre as regiões.
BorderLayout(int, int)	Cria um novo gerenciador de *layout* com o espaçamento horizontal e vertical especificados.
getHgap()	Obtêm o espaçamento horizontal.
getVgap()	Obtêm o espaçamento vertical.
setHgap(int)	Especifica o espaçamento horizontal.
setVgap(int)	Especifica o espaçamento vertical.

Tabela 37 Métodos da Classe java.awt.BorderLayout

6.4.4 CardLayout

Como visto, os gerenciadores de *layout* **FlowLayout**, **GridLayout** e **BorderLayout** são bastante flexíveis embora simples. A classe **java.awt.CardLayout** define um gerenciador de *layout* mais sofisticado que trata os componentes adicionados a um *container* como cartas de uma pilha de cartas, isto é, num dado momento apenas uma carta é visível. A ordenação das cartas depende da ordem de inserção dos componentes, sendo que esta classe oferece meios para navegar-se através dos componentes gerenciados, isto é, permite modificar a exibição para primeiro componente, o último, o anterior ou posterior. Também é possível associar-se uma *string* de identificação a cada componente permitindo a navegação direta para o mesmo.

Com este gerenciador de *layout* é possível criar-se pilhas de outros componentes onde qualquer um pode ser exibido de cada vez, para tanto devemos utilizar um **java.awt.Panel** ou **java.awt.Frame** como *container*. A utilização de painéis como componentes adicionados a um outro administrado por um *layout manager* desta espécie permite a criação de “folhas” contendo diversos outros componentes arranjados conforme os **layouts** de seus respectivos painéis. Obviamente a utilização deste gerenciado de layout é mais trabalhosa em termos de código mas os resultados que podem ser obtidos são bastante satisfatórios.

Na Figura 46 temos o exemplo de um aplicação que contêm seu painel central administrado através de um objeto **CardLayout**, onde os botões do painel lateral podem ser usados para controlar a exibição das páginas (cartas) da área central.

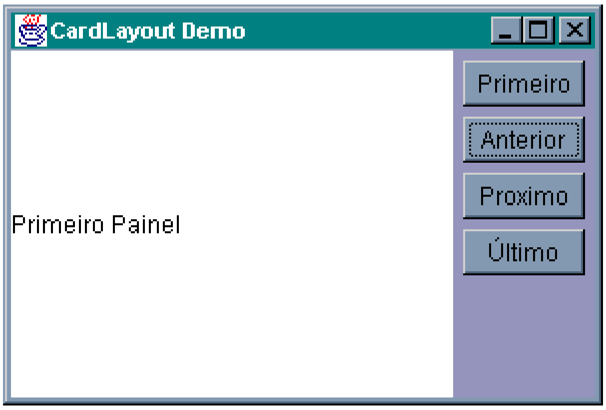


Figura 46 Aplicação CardLayoutDemo

Esta aplicação utiliza seu *layout default*, o **BorderLayout**, exibindo cinco diferentes páginas na área central através de um painel gerenciado por um **CardLayout**. No painel central temos que as três primeiras são compostas por componentes simples e as restantes por painéis adicionais. O painel lateral, que também usa seu *layout* default **FlowLayout**, contêm um painel auxiliar onde os botões são arranjados através do **GridLayout**. Temos assim que os quatro gerenciadores de *layout* vistos são usados neste exemplo. O código correspondente é exibido no Exemplo 70.

```
// CardLayoutDemo.java

import java.awt.*;
import java.awt.event.*;

public class CardLayoutDemo extends Frame
                            implements ActionListener {
  private Panel cards;
  private Button bPrim, bAnt, bProx, bUlt;

  // método main
  public static void main(String args[]) {
    CardLayoutDemo f = new CardLayoutDemo();
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }

  // construtor
  public CardLayoutDemo() {
    super("CardLayout Demo");
    setSize(300, 200);
    // Painel Lateral de Controle
    Panel p1 = new Panel();
    p1.setBackground(SystemColor.control);
    Panel p2 = new Panel();
    p2.setBackground(SystemColor.control);
    p2.setLayout(new GridLayout(4, 1, 5, 5));
    p2.add(bPrim = new Button("Primeiro"));
    bPrim.addActionListener(this);
    p2.add(bAnt  = new Button("Anterior"));
    bAnt.addActionListener(this);
    p2.add(bProx = new Button("Proximo"));
    bProx.addActionListener(this);
```

```
    p2.add(bUlt  = new Button("Último"));
    bUlt.addActionListener(this);
    p1.add(p2);
    add("East", p1);
    // Painel Múltiplo
    cards = new Panel();
    cards.setLayout(new CardLayout());
    cards.add(new Label("Primeiro Painel"), "C1");
    cards.add(new Picture("cupHJbutton.gif"), "C2");
    cards.add(new TextArea(), "C3");
    p2 = new Panel();
    p2.setBackground(SystemColor.control);
    p2.add(new Label("Quarto Painel"));
    p2.add(new Picture("cupHJbutton.gif", 32, 32));
    cards.add(p2, "C4");
    p2 = new Panel();
    p2.setBackground(SystemColor.lightGray);
    cards.add(p2, "C5");
    add("Center", cards);
  }

  public void actionPerformed(ActionEvent e) {
    CardLayout cl = (CardLayout) cards.getLayout();
    if (e.getSource()==bPrim) {
      cl.first(cards);
    } else if (e.getSource()==bAnt) {
      cl.previous(cards);
    } else if (e.getSource()==bProx) {
      cl.next(cards);
    } else if (e.getSource()==bUlt) {
      cl.last(cards);
    }
  }
}
```

Exemplo 70 Classe CardLayoutDemo

Da classe **java.awt.CardLayout** selecionamos os seguintes construtores e métodos:

Método	Descrição
CardLayout()	Cria um novo gerenciador de *layout* sem espaçamento entre as regiões.
CardLayout(int, int)	Cria um novo gerenciador de *layout* com o espaçamento horizontal e vertical especificados.
first(Container)	Exibe o primeiro componente adicionado ao *layout* do *container* especificado
getHgap()	Obtêm o espaçamento horizontal.
getVgap()	Obtêm o espaçamento vertical.
last(Container)	Exibe o último componente adicionado ao *layout* do *container* especificado
next(Container)	Exibe o próximo componente adicionado ao *layout* do *container* especificado
previous(Container)	Exibe o componente anterior adicionado ao *layout* do *container* especificado
setHgap(int)	Especifica o espaçamento horizontal.
setVgap(int)	Especifica o espaçamento vertical.

Tabela 38 Métodos da Classe java.awt.CardLayout

6.5 O Modelo de Eventos da AWT

A partir do Java versão 1.1 foi introduzido um novo modelo de eventos muitas vezes conhecido como o novo modelo de eventos da AWT. Neste novo modelo são identificados objetos que atuarão como fontes (*sources*) e outros que atuarão como receptores (*listeners*) de eventos trabalhando juntos no que se chama modelo de delegação. Através do modelo de delegação os objetos receptores se tornam-se responsáveis por receber e processar os eventos produzidos pelos objetos fonte. Na verdade um *listener* (objeto receptor) recebe a incumbência de processar certos tipos de eventos, possibilitando a criação de *listeners* especializados em determinadas tarefas.

Um evento é uma ação produzida por ações do usuário, ou seja, é o resultado da interação do usuário com a interface apresentada pela aplicação podendo ser o acionamento de alguma tecla, a movimentação do *mouse* ou o acionamento deste sobre algum dos componente da GUI. Usualmente cada tipo de componente pode disparar certos tipos de eventos que podem ser "ouvidos" por mais de um *listener*.

Cada tipo de evento é um objeto de uma classe distinta enquanto que cada *listener* é uma interface que implementa um tipo particular de receptor e processador de eventos e com isto temos três objetos envolvidos no modelos de eventos: o objeto fonte, o objeto *listener* e o objeto evento enviado da fonte para o *listener* designado.

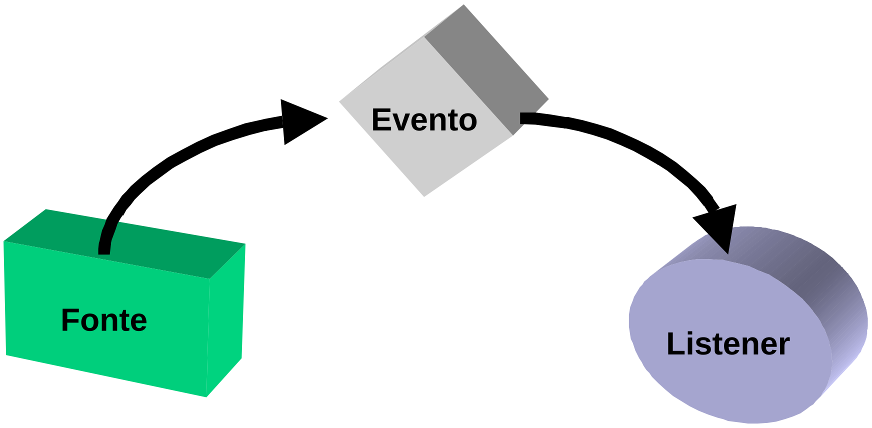


Figura 47 Novo Modelo Esquemáticos de Eventos AWT

Com este modelo é então possível:

(i) Definir-se classes separadas para o processamento de eventos específicos de um único componente ou de um grupo componentes que produzam o mesmo evento;

(ii) Definir-se explicitamente qual objeto será responsável pelo processamento de eventos gerados por um determinado componente;

(iii) Utilizar-se de diferentes estratégias para codificação: classe única, classes internas (*inner classes*) ou classes separadas.

(iv) Definir-se novos tipos de eventos e interfaces processadoras de eventos e

(v) Oferecer suporte para construção de **Beans** (um tipo especial de componente multiplataforma).

Na AWT cada componente produz eventos específicos e assim suas classes implementam métodos **add___Listener** e **remove___Listener** para cada evento **___** produzido existindo uma interface **___Listener** apropriada que deve ser implementada pelo objeto ao qual se delegará a responsabilidade de tratar os eventos produzido por outros objetos. Todos os eventos fazem parte de uma hierarquia de classes onde a superclasse é **java.awt.AWTEvent**, existindo eventos orientados para ações, ajustes, edição de texto, seleção de itens e eventos genéricos de componentes. A classe

java.awt.event.ComponentEvent por sua vez dá origem às classes de eventos relacionados com *containeres*, foco, janelas e eventos de entrada. Esta última oferece suporte para os eventos relacionados ao mouse e acionamento de teclas compondo assim uma hierarquia de quatro níveis como ilustrado na Figura 48.

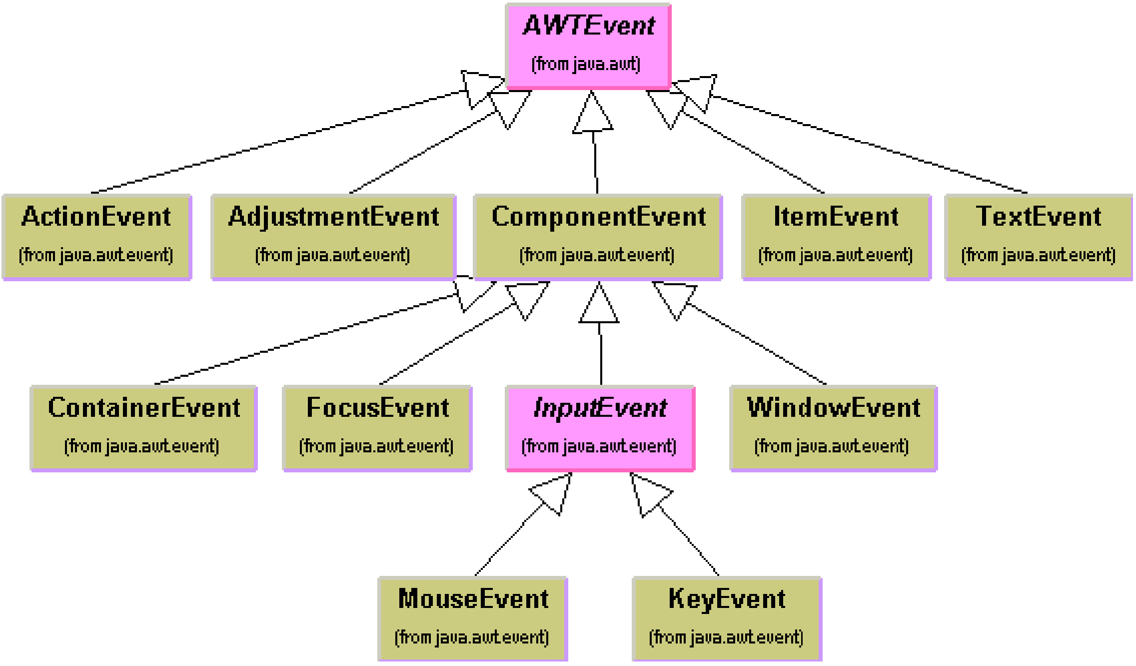


Figura 48 Hierarquia de Eventos da AWT

Apresentamos na Tabela 39 e Tabela 40 os principais métodos das classes **java.util.EventObject** e **java.awt.AWTEvent** compartilhados com toda a hierarquia de classes de eventos da AWT.

Método	Descrição
getSource()	Retorna uma referência para o objeto ou componente que gerou o evento.

Tabela 39 Métodos da Classe EventObject

Método	Descrição
consume()	Consome o evento.
getId()	Retorna o ID deste evento.
isConsumed()	Verifica se o evento está ou não consumido.

Tabela 40 Métodos da Classe AWTEvent

Especificamente a classe **java.awt.AWTEvent** especifica um conjunto de constantes que são máscaras de seleção para certos eventos, ou seja, máscaras que permitem filtrar certos eventos determinando se são ou não de uma dado tipo desejado. Ao registrar-se um *listener* para um componente, o valor apropriado de máscara é associado internamente ao componente. Exemplos de máscaras: **ACTION_EVENT_MASK**, **ITEM_EVENT_MASK, WINDOW_EVENT_MASK**, etc.

A seguir, na Tabela 41, constam quais são os eventos gerados pelos principais componentes da AWT:

Componente	Action Event	Adjusment Event	Container Event	Focus Event	Item Event	Key Event	Mouse Event	Text Event	Window Event
Applet			✓	✓		✓	✓		
Button	✓			✓		✓	✓		
Canvas				✓		✓	✓		
Checkbox				✓	✓	✓	✓		
CheckboxMenuItem	✓				✓				
Choice				✓	✓	✓	✓		
Frame			✓	✓		✓	✓		✓
Label				✓		✓	✓		
List	✓			✓		✓	✓		
Menu	✓								
MenuItem	✓								
Panel			✓	✓		✓	✓		
PopupMenu	✓								
ScrollBar		✓		✓		✓	✓		
TextArea				✓		✓	✓	✓	
TextField	✓			✓		✓	✓	✓	
Window			✓	✓		✓	✓		✓

Tabela 41 Componentes e Seus Eventos

Note que os eventos **FocusEvent**, **KeyEvent** e **MouseEvent** são gerados por quase todos os componente enquanto outros tais como o **AdjustmentEvent**, o **ContainerEvent** , **ItemEvent** ou **WindowEvent** são gerados por componentes específicos.

Desta forma já sabemos identificar quais os eventos produzidos por quais componentes, restando agora determinar quais interfaces são necessárias para processar tais eventos. Na Tabela 42 temos uma relação de métodos e interfaces associadas a um certo tipo de evento.

Para processar um evento produzido por um certo componente deve-se implementar todos os métodos indicados na classe delegada de tal responsabilidade, isto é, na classe que implementa o *listener* para tal evento. Em muitas situações só se deseja processar eventos específicos através dos métodos apropriados, tornando cansativo o uso das interfaces nestas situações, tal qual exemplificado quando foram apresentados o **Frames** na seção 6.3.1.

Para evitar-se a implementação de métodos vazios (*dummy methods*), pode-se recorrer ao uso das classes adaptadoras (*adapter classes*) que implementam tais métodos *dummy*, liberando o programador de tal tarefa pois através da extensão destas classes torna necessário implementar apenas os métodos de interesse em sobreposição aos métodos vazios pré-definidos na classe base adaptadora.

Note que só existem classes adaptadoras para interfaces que possuem mais de um método. Qualquer método de uma interface pode ser implementado através da classe adaptadora. Outra questão é que uma certa classe pode implementar múltiplas interfaces mas pode apenas estender uma classe pois o Java não suporta a herança múltipla. Devido a esta restrição muitas vezes o uso de classes adaptadoras exige a criação de novas classes em separado (veja o Exemplo 52) ao invés da incorporação do métodos da interfaces na classe da aplicação.

Evento	Interface ou Adaptador	Métodos
ActionEvent	ActionListener	actionPerformed
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged
ComponentEvent	ComponentListener ComponentAdapter	componentMoved componentHidden componentResized
ContainerEvent	ContainerListener ContainerAdapter	componentAdded componentRemoved
FocusEvent	FocusListener FocusAdapter	focusGained focusLost
ItemEvent	ItemListener	itemStateChanged
KeyEvent	KeyListener KeyAdapter	keyPressed keyReleased keyTyped
MouseEvent	MouseListener MouseAdapter	mousePressed mouseReleased mouseEntered mouseExited mouseClicked
	MouseMotionListener MouseMotionAdapter	mouseDragged mouseMoved
TextEvent	TextListener	textValueChanged
WindowEvent	WindowListener WindowAdapter	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated

Tabela 42 Eventos, Interfaces Associadas e Seus Métodos

6.5.1 ActionListener

A interface **java.awt.event.ActionListener** é responsável por processar eventos de ação **java.awt.event.ActionEvent** gerados pelo acionamento de botões (**Button**), seleção de itens (**MenuItem**) em menus suspensos (*drop-down*) ou independentes (*popup*), pelo acionamento do “Enter” em caixas de entrada (**TextField**) e pelo duplo-clique em caixas de lista (**List**). Sua implementação exige a inclusão do método **actionPerformed** nestas classes.

Vários dos exemplos apresentados nas seções anteriores já utilizaram esta interface e este método. Como complementação, o evento **ActionEvent** possui os seguintes métodos específicos associados:

Método	Descrição
getActionCommand()	Retorna a *string* associada a este componente
getModifiers()	Obtêm os modificadores utilizados no instante da geração deste evento.

Tabela 43 Métodos da Classe java.awt.event.ActionEvent

6.5.2 ItemListener

A interface **java.awt.event.ItemListener** é responsável por processar eventos de ação **java.awt.event.ItemEvent** gerados pelo acionamento de caixas de opção (**Checkbox**), botões de opção (**CheckboxGroup**) e itens em menus do tipo opção (**CheckboxMenuItem**). Sua implementação exige a inclusão do método **itemStateChanged** nestas classes.

Alguns dos exemplos apresentados nas seções anteriores utilizaram esta interface e este método tais como o Exemplo 63, o Exemplo 64 e o Exemplo 73. Seguem alguns dos métodos específicos associados a este evento.

Método	**Descrição**
getItem()	Retorna o item que originou o evento.
getStateChange()	Retorna o tipo de mudança de estado ocorrida.

Tabela 44 Métodos da Classe java.awt.event.ItemEvent

6.5.3 TextListener

A interface **java.awt.event.TextListener** é utilizada para responder aos eventos provocados pela alteração do texto contido em componentes do tipo **TextField** e **TextArea**. Sua interface exige que sejam implementado o método **textValueChanged** cujo evento associado, **java.awt.event.TextEvent**, possui apenas um método específico de interesse listado abaixo:

Método	**Descrição**
paramString()	Retorna a *string* associada ao evento.

Tabela 45 Métodos da Classe java.awt.event.ItemEvent

Existem algumas constantes na classe **TextEvent** para auxiliar o tratamento deste evento.

6.5.4 KeyEvent

A interface **java.awt.event.KeyListener**, presente em praticamente todos os componentes da AWT é utilizada para responder aos eventos provocados pelo acionamento do teclado quando o foco está sobre o componente. Permite o pré-processamento da entrada de dados fornecida pelo usuário e outras ações sofisticadas. Sua interface exige que sejam implementado o método **keyPressed, keyReleased** e **keyTyped**. O evento associado, **java.awt.event.KeyEvent** possui os seguintes métodos específico de interesse listados abaixo:

Método	**Descrição**
getKeyChar()	Retorna o caractere associado à tecla que originou o evento.
getKeyCode()	Retorna o código de tecla associado à tecla que originou o evento.
getKeyModifiersText(int)	Retorna uma *string* descrevendo as teclas modificadoras “Shift”, “Ctrl” ou sua combinação.
getKeyText(int)	Retorna uma *string* descrevendo a tecla tal como “Home”, “F1” ou “A”.
is ActionKey()	Determina se a tecla associada é ou não uma tecla de ação.
setKeyChar(char)	Substitui o caractere associado à tecla que originou o evento.

Tabela 46 Métodos da Classe java.awt.event.KeyEvent

Esta classe possui um grande conjunto de constantes que podem ser utilizadas para identificação das teclas acionadas tal como exemplificado na Tabela 51.

6.5.5 MouseListener e MouseMotionListener

O evento **MouseEvent** é utilizado por duas diferentes interfaces: **java.awt.event.MouseListener** e **java.awt.event.MouseMotionListener**. A primeira interface se destina a processar eventos de acionamento dos botões do *mouse* e detecção da entrada e saída de um *mouse* por sobre um componente. A segunda visa o processamento da movimentação do *mouse*.

A interface **java.awt.event.MouseListener** exige a implementação de vários métodos: **mousePressed**, **mouseReleased**, **mouseEntered**, **mouseExited** e **mouseClicked** como ilustrado no Exemplo 76 e no Exemplo 79. Para reduzir a quantidade de código necessária pode-se optar pelo extensão da classe **java.awt.event.MouseAdapter** que oferece uma implementação nula dos métodos acima.

A interface **java.awt.event.MouseMotionListener** exige a implementação dos seguintes métodos: **mouseDragged** e **mouseMoved**. Como na para interface **MouseListener**, pode se optar pela extensão da classe **MouseMotionAdapter** para reduzir-se a quantidade de código necessária.

O evento MouseEvento apresenta os seguintes métodos específicos:

Método	Descrição
getClickCount()	Retorna o número de cliques relacionado a este evento.
getPoint()	Retorna um objeto Point contendo as coordenadas do local onde ocorreu o acionamento do mouse.
getX()	Retorna a coordenada x do local onde ocorreu o acionamento do mouse.
getY()	Retorna a coordenada y do local onde ocorreu o acionamento do mouse.
isPopupTrigger()	Verifica se este evento está associado a exibição de menus *popup* nesta plataforma.
translatePoint(int, int)	Translada a coordenada do evento com os valores especificados.

Tabela 47 Métodos da Classe java.awt.event.MouseEvent

6.5.6 FocusListener

Para o processamento de eventos associados a mudança de foco entre os componentes deve ser implementada a interface **java.awt.event.FocusListener**, que exige a codificação dos métodos **focusGain** e **focusLost** que utilizam o evento **java.awt.event.FocusEvent** gerado pela maioria dos componentes da AWT.

Dado a existência de mudanças de foco temporárias (por exemplo, quando se usam barras de rolagem associadas a um componente **TextArea**, **List** ou **Choice**) ou permanentes (quando o usuário utiliza a tecla “Tab” para navegar entre componentes da interface), é útil o método abaixo:

Método	Descrição
isTemporary()	Determina se o evento de alteração de foco é temporário ou permanente.

Tabela 48 Métodos da Classe java.awt.event.FocusEvent

Existem algumas constantes na classe **FocusEvent** para auxiliar o tratamento deste evento.

6.5.7 AdjustmentListener

Esta interface, **java.awt.event.AdjustmentListener**, é utilizada para reagir aos eventos de ajuste provocados por componentes **java.awt.ScrollBar**. Estes componentes produzem o evento **java.awt.event.AdjustmentEvent** tratado pelo método **adjustmentValueChanged** que deve ser obrigatoriamente implementado.

Método	Descrição
getAdjustable()	Retorna o objeto que originou este evento.
getAdjustmentType()	Retorna o tipo de ajuste efetuado.
getValue()	Retorna o valor do ajuste associado ao evento

Tabela 49 Métodos da Classe java.awt.event.AdjustmentEvent

A classe **AdjustmentEvent** possui diversas constantes associadas aos tipos de ajustes possíveis.

6.6 Menus

Como esperado, a AWT oferece suporte bastante completo para criação de aplicações utilizando barras de menus (*menu bars*), menus suspensos (*pull-down menus*) e menus independentes (*popup menus*) através conjunto de classes resumidamente descrito a seguir:

Classe	Descrição
MenuComponent	Classe abstrata que define o comportamento e estrutura básica dos elementos componentes de *menus*.
MenuItem	Classe que define uma entrada simples de menu.
MenuBar	Classe que define uma barra de menus para uma aplicação.
Menu	Classe que define um menu suspenso.
MenuShortCut	Classe que define um atalho para um item de menu.
CheckboxMenuItem	Classe que define uma entrada de menu tipo opção.
PopupMenu	Classe que define um menu independente.

Tabela 50 Classes para Criação de Menus

Na Figura 49 temos a hierarquia das principais classes relacionadas com os componentes de menu.

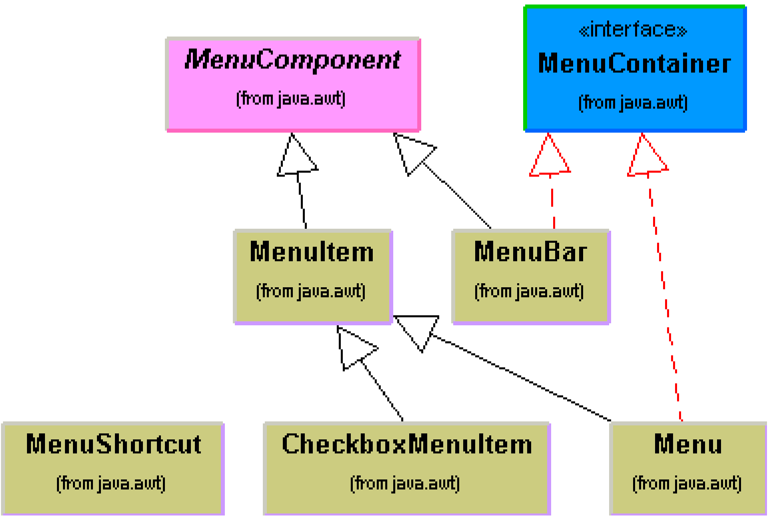


Figura 49 Hierarquia de Classes para Construção de Menus

6.6.1 Os Componentes MenuBar, Menu e MenuItem

Adicionar uma barra de menus contendo menus e itens de menus é bastante simples no Java. Basta seguir intuitivamente a estrutura visual apresentada:

(i) adicionar uma barra de menu (**java.awt.MenuBar**) a janela (**java.awt.Frame**);
(ii) adicionar os menus suspensos (**java.awt.Menu**) a barra de menu;
(iii) adicionar os itens de menus desejados (**java.awt.MenuItem** ou **java.awt.CheckboxMenuItem**) a cada um dos menus suspensos e
(iv) adicionar os *listeners* necessários para ser possível que os menus e itens de menus exibam funcionalidade.

Para criamos a barra de menu fazemos:

```
MenuBar barraMenu = new MenuBar();
```

Para cada menu suspenso declaramos e instanciamos um objeto **Menu**, como exemplificado para menus suspensos “Arquivo”, “Editar” e “Ajuda”.

```
Menu menuArquivo = new Menu("Arquivo");
Menu menuEditar = new Menu("Editar");
Menu menuAjuda = new Menu("Ajuda");
```

Para cada item de menu de cada menu item, declaramos e instanciamos objetos **MenuItem**. Abaixo temos o código correspondente para itens de um possível menu “Arquivo”.

```
MenuItem miNovo = new MenuItem("Novo");
MenuItem miAbrir = new MenuItem("Abrir...");
MenuItem miSalvar = new MenuItem("Salvar");
MenuItem miSalvarComo = new MenuItem("Salvar Como...");
MenuItem miSair = new MenuItem("Sair");
```

Agora adicionamos os itens de menu acima ao menu suspenso “Arquivo”. Note o uso do método **addSeparator** para adicionar-se separadores aos itens de um menu suspenso:

```
menuArquivo.add(miNovo);
menuArquivo.add(miAbrir);
menuArquivo.add(miSalvar);
menuArquivo.add(miSalvarComo);
menuArquivo.addSeparator();
menuArquivo.add(miSair);
```

O mesmo deve ser feito para os demais menus da aplicação. Finalmente adicionamos os menus suspensos a barra de menu e esta ao **Frame** através do método **setMenuBar**:

```
barraMenu.add(menuArquivo);
barraMenu.add(menuEditar);
barraMenu.add(menuAjuda);
setMenuBar(barraMenu);
```

Segue o código necessário para a adição dos três menus exemplificados e seus itens a uma aplicação.

```
// MenuDemo.java

import java.awt.*;

public class MenuDemo extends Frame{
  public MenuDemo() {
    super("Menu Demo");
    setSize(300, 100);
    setLocation(50, 50);
    // barra de menu
    MenuBar barraMenu = new MenuBar();
    // menus suspensos
    Menu menuArquivo = new Menu("Arquivo");
```

```
    Menu menuEditar = new Menu("Editar");
    Menu menuAjuda = new Menu("Ajuda");
    // itens do menu Arquivo
    MenuItem miNovo = new MenuItem("Novo");
    MenuItem miAbrir = new MenuItem("Abrir...");
    MenuItem miSalvar = new MenuItem("Salvar");
    MenuItem miSalvarComo = new MenuItem("Salvar Como...");
    MenuItem miSair = new MenuItem("Sair");
    // itens do menu Editar
    MenuItem miRecortar = new MenuItem("Recortar");
    MenuItem miCopiar = new MenuItem("Copiar");
    MenuItem miColar = new MenuItem("Colar");
    // itens do menu Editar
    MenuItem miAjuda = new MenuItem("Ajuda...");
    MenuItem miSobre = new MenuItem("Sobre...");
    // adicionando itens ao menu Arquivo
    menuArquivo.add(miNovo);
    menuArquivo.add(miAbrir);
    menuArquivo.add(miSalvar);
    menuArquivo.add(miSalvarComo);
    menuArquivo.addSeparator();
    menuArquivo.add(miSair);
    // adicionando itens ao menu Editar
    menuEditar.add(miRecortar);
    menuEditar.add(miCopiar);
    menuEditar.add(miColar);
    // adicionando itens ao menu Ajuda
    menuAjuda.add(miAjuda);
    menuAjuda.addSeparator();
    menuAjuda.add(miSobre);
    // adicionando menus suspensos a barra
    barraMenu.add(menuArquivo);
    barraMenu.add(menuEditar);
    barraMenu.add(menuAjuda);
    // adicionando barra de menu ao frame
    setMenuBar(barraMenu);
    addWindowListener(new CloseWindowAndExit());
  }

  static public void main (String args[]) {
    MenuDemo f = new MenuDemo();
    f.show();
  }
}
```

Exemplo 71 Classe MenuDemo

O resultado, que pode ser visto na Figura 50, é uma aplicação sem qualquer funcionalidade, servido apenas para demonstrar o processo de adição de menus a um **Frame**.

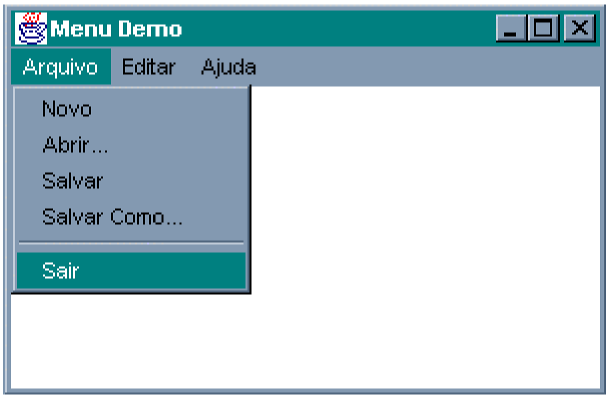


Figura 50 Aplicação MenuDemo

Para associarmos os itens de menu às funcionalidades desejadas para a aplicação é necessário adicionarmos os *listeners* adequados. Os objetos **Menu** e **MenuItem** enviam eventos do tipo **ActionEvent** quando acionados, assim é necessário implementar-se a interface **ActionListener** numa classe para o processamento de tais eventos associando-se tal classe a cada menu ou item de menu desejado.

Assim, o trecho de código do Exemplo 71 correspondente a instanciação dos 5 itens pertencentes ao menu "Arquivo" transforma-se no trecho de código a seguir dada a adição do *listener* para cada item desejado:

```
MenuItem miNovo = new MenuItem("Novo");
miNovo.addActionListener(this);
MenuItem miAbrir = new MenuItem("Abrir...");
miAbrir.addActionListener(this);
MenuItem miSalvar = new MenuItem("Salvar");
miSalvar.addActionListener(this);
MenuItem miSalvarComo = new MenuItem("Salvar Como...");
miSalvarComo.addActionListener(this);
MenuItem miSair = new MenuItem("Sair");
miSair.addActionListener(this);
```

Notamos que embora simples, a adição de menus e associação de *listeners* aos itens de menus é uma tarefa repetitiva e cansativa.

6.6.2 Simplificando a Criação de Menus

A criação de menus é uma tarefa estruturalmente semelhante para qualquer tipo de menu, assim poderia ser bastante simplificada através do uso de uma rotina que realizasse o trabalho mecânico associado a tais definições.

Horstmann e Cornell (1997) sugerem um método bastante útil para construção automatizada de menus, implementado no Exemplo 72, que recebe três argumentos: (i) um objeto **String** ou **Menu**, (ii) um vetor simples contendo os itens a serem adicionados ao Menu, na forma de **Strings** ou **MenuItem** e finalmente (iii) uma referência para a classe que implementa a interface **ActionListener** para todos os itens de menu adicionados.

```
// MenuTool.java

import java.awt.*;
import java.awt.event.*;

public final class MenuTool {

    public static final Menu makeMenu(Object base,
                                      Object[] items,
                                      Object target) {
    Menu m = null;
    if (base instanceof Menu)
```

```
      m = (Menu)base;
    else if (base instanceof String)
      m = new Menu((String)base);
    else
      return null;
    for (int i = 0; i < items.length; i++) {
      if (items[i] instanceof String) {
        MenuItem mi = new MenuItem((String)items[i]);
        if (target instanceof ActionListener)
          mi.addActionListener((ActionListener)target);
          m.add(mi);
        } else if (items[i] instanceof CheckboxMenuItem
                   && target instanceof ItemListener) {
          CheckboxMenuItem cmi = (CheckboxMenuItem)items[i];
          cmi.addItemListener((ItemListener)target);
          m.add(cmi);
        } else if (items[i] instanceof MenuItem) {
          MenuItem mi = (MenuItem)items[i];
          if (target instanceof ActionListener)
            mi.addActionListener((ActionListener)target);
          m.add(mi);
        } else if (items[i] == null)
          m.addSeparator();
    }
    return m;
  }
}
```

Exemplo 72 Classe MenuTool

A criação do menu “Arquivo” do Exemplo 71 seria reduzida ao código abaixo:

```
Menu menuArquivo = MenuTool.makeMenu("Arquivo",
                                new Object[] {"Novo",
                                "Abrir...", "Salvar",
                                "Salvar Como...", null, "Sair"},
                                this);
```

Note o uso de **null** para indicar a inclusão de um separador de itens de menu.

Para demostrarmos o quanto tal ferramenta simplifica a construção de menus, segue a modificação completa do Exemplo 71 para fazer uso desta classe auxiliar.

```
// MenuDemo2.java

import java.awt.*;
import java.awt.event.*;

public class MenuDemo2 extends Frame
                       implements ActionListener {
  private TextField tfStatus;

  public MenuDemo2() {
    super("Menu Demo2");
    setSize(300, 100);
    setLocation(50, 50);
    // barra de menu
    MenuBar barraMenu = new MenuBar();
    // menus suspensos
    Menu menuArquivo = MenuTool.makeMenu("Arquivo",
                                new Object[] {"Novo",
                                "Abrir...", "Salvar",
                                "Salvar Como...", null, "Sair"},
                                this);
    Menu menuEditar = MenuTool.makeMenu("Editar",
```

```
                               new Object[] {"Recortar",
                               "Copiar", "Colar"},
                               this);
    Menu menuAjuda = MenuTool.makeMenu("Ajuda",
                              new Object[] {"Ajuda...",
                              "Sobre..."},
                              this);
    barraMenu.add(menuArquivo);
    barraMenu.add(menuEditar);
    barraMenu.add(menuAjuda);
    // adicionando barra de menu ao frame
    setMenuBar(barraMenu);
    addWindowListener(new CloseWindowAndExit());
    // Barra de Status
    Panel p = new Panel();
    p.setLayout(new BorderLayout(5, 5));
    p.add("Center", tfStatus = new TextField());
    tfStatus.setBackground(SystemColor.control);
    tfStatus.setFont(new Font("SansSerif", Font.PLAIN, 10));
    tfStatus.setEditable(false);
    add("South", p);
  }

  static public void main (String args[]) {
    MenuDemo2 f = new MenuDemo2();
    f.show();
  }

  public void actionPerformed(ActionEvent e) {
    // Verifica se evento acionado por item de menu
    if (e.getSource() instanceof MenuItem) {
      MenuItem mi = (MenuItem) e.getSource();
      // Indicação na Barra de Status
      tfStatus.setText(mi.getLabel());
      // Testa item selecionado e aciona rotina
      if (mi.getLabel().equals("Sair"))
        System.exit(0);
    }
  }
}
```

Exemplo 73 Classe MenuDemo2

Percebemos rapidamente a drástica redução no volume do código, pois são necessárias apenas algumas linhas para a completa definição de um menu incluindo a associação de seus itens com um dados *listener*. Outro aspecto interessante é a forma com que ocorre o processamento dentro do método **actionPerformed**. Nele, selecionam-se os objetos **MenuItem** que enviaram eventos e, ao invés de utilizarmos variáveis de referência para identificar-se a origem (e.getSource() == miSair, por exemplo), comparamos o texto do item de menu com o texto recebido, simplificando mais uma vez o código que não mais necessita manter uma referência para cada item a ser processado no *listener*.

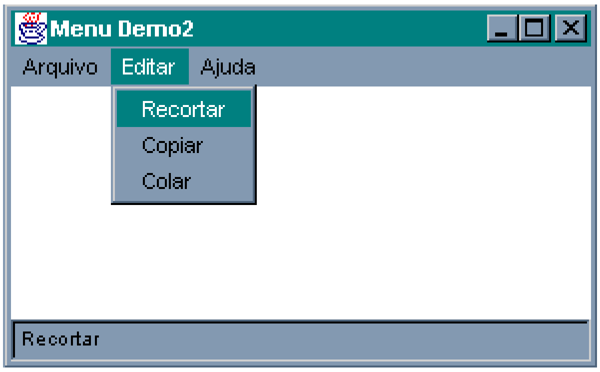


Figura 51 Aplicação MenuDemo2

6.6.3 Adicionando Atalhos para os Menus

Para adicionar-se teclas de atalho aos itens de menu devemos criar um instância do objeto **MenuItem** que receba um instância do objeto **MenuShortcut** como no exemplo a seguir:

```
MenuItem miSair = new MenuItem("Sair",
                               new MenuShortcut(KeyEvent.VK_R));
```

Desta forma, ao pressionar-se a sequência CTRL+R aciona-se a opção de menu "Sair", produzindo o efeito desejado. O Java restringe as teclas de atalho as combinações da tecla **Control** com as demais em virtude da necessária compatibilidade com as demais plataformas. A constante VK_R (*virtual key R*) é a denominação da tecla “R”. A relação completa da denominação das teclas podem ser obtidas na classe **java.awt.event.KeyEvent**. Seguem alguns outros exemplos:

Virtual Key Constant	**Descrição**
VK_0 .. VK_9	Teclas de 0 .. 9 (0x30 .. 0x39)
VK_A .. VK_Z	Teclas de A. .. Z (0x41 .. 0x5A)
VK_F1 .. VK_F12	Teclas F1 a F12
VK_ENTER	Tecla Enter

Tabela 51 Denominação de Teclas

O método **makeMenu** também suporta a adição de teclas de atalhos, itens de menu tipo opção (**CheckboxMenuItem**) bem como a inclusão de submenus como indicado abaixo:

```
Menu menuFonte = MenuTool.makeMenu("Fontes",
                           new Object[] {
                           new CheckboxMenuItem("Grande"),
                           new CheckboxMenuItem("Normal", true),
                           new CheckboxMenuItem("Pequeno")},
                           this);
Menu menuEditar = MenuTool.makeMenu("Editar",
                           new Object[] {
                           new MenuItem("Recortar",
                               new MenuShortcut(KeyEvent.VK_X)),
                           new MenuItem("Copiar",
                               new MenuShortcut(KeyEvent.VK_C)),
                           new MenuItem("Colar",
                               new MenuShortcut(KeyEvent.VK_V)),
                           null,
                           menuFonte},
                           this);
```

A inclusão de um submenu é feita através da inclusão de um menu comum a outro menu, produzindo o efeito indicado na ilustração a seguir.

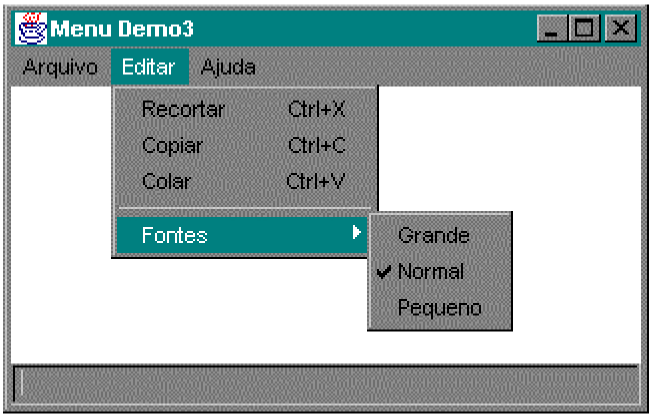


Figura 52 Aplicação MenuDemo3

7 Aplicações e Primitivas Gráficas

Além da família de componentes GUI, a linguagem Java oferece um conjunto bastante amplo de primitivas que podem ser utilizadas para renderização direta em aplicações. Como renderização entende-se o desenho de *strings*, pontos, linhas, formas geométricas planas e imagens realizado pelo comando direto das aplicações, isto é, efetuado programaticamente. Tais primitivas, também denominadas de primitivas gráficas, operam sobre um contexto gráfico (*graphic context*), um objeto especial através do qual são disponibilizadas todas as operações gráficas suportadas.

7.1 Contexto Gráfico

O contexto gráfico é um objeto abstrato definido pela classe **java.awt.Graphics**. Sendo um objeto abstrato, não pode ser instanciado diretamente, constituindo uma superclasse para implementações específicas da JVM, isto é, para cada plataforma em que a JVM opera é criada uma classe que estende a classe **java.awt.Graphics** onde são codificadas de fato tais facilidades gráficas considerando o uso das primitivas oferecidas pelo sistema operacional em questão.

As razões que justificam o emprego desta estratégia são duas: (i) é impossível implementar-se de forma genérica as primitivas gráficas desejadas para todas as plataformas existentes e (ii) a implementação realizada desta forma torna o contexto gráfico transparente para o usuário, ou seja, neutro com relação à plataforma escolhida.

A classe **java.awt.Component** é a superclasse também abstrata da grande maioria dos componentes existentes na AWT e oferece um método denominado **paint** responsável pela renderização do componente, isto é, por seu desenho na interface da aplicação. Este método recebe da JVM uma referência do contexto gráfico na qual opera a aplicação, podendo ser sobreposto (*overhided*) para que o programador controle como se dará tal renderização. O método **paint** é declarado como abaixo:

```
public void paint(Graphics g)
```

Se o programador necessita acionar tal método, para forçar o desenho de um componente num certo instante, preferencialmente deve acionar o método **repaint**, que limpa o segundo plano (*background*) antes das operações de desenho propriamente ditas, evitando também a necessidade de se determinar qual o contexto gráfico da aplicação (cuja referência é parâmetro necessário para o acionamento direto do método **paint**). A declaração do método **repaint** é dada a seguir:

```
public void repaint()
```

Por outro lado o programador, se uma chamada explícita ao método **paint** não é útil na maioria dos casos, pode ser desejado sua substituição para construção de aplicações ou componentes cuja renderização seja particular e determinada pelo próprio programador.

Embora este procedimento não seja frequente, o método **paint** pode ser sobreposto (*overhided*), isto é, reimplementado de modo que o programador possa construir aplicações gráficas para desenho, edição de imagens, animações, onde é necessário que controle da renderização esteja nas mãos do programador. Como o método **paint** recebe uma referência para o contexto gráfico da aplicação podem ser utilizadas as primitivas gráficas da classe **java.awt.Graphics** que oferece resumidamente as seguintes facilidades:

- encapsula informações de estado do contexto gráfico;
- efetua operações de desenho de arcos, imagens, linhas e algumas figuras geométricas planas elementares;
- efetua operações de cópia, limpeza e preenchimento de áreas e
- efetua o desenho de *string* e caracteres com o devido controle de fontes.

O objeto **java.awt.Graphics** subentende uma superfície imaginária na qual podem ser utilizadas as primitivas gráficas, ou seja, se o contexto gráfico se refere a uma determinada janela da aplicação então utilizá-lo significa desenhar na tela do computador, mais especificamente dentro da janela da aplicação. Se o contexto gráfico for associado a um objeto de impressão então temos que ele ser refere ao papel utilizado por este dispositivo para efetuar a impressão propriamente dita. Disto concluímos que o objeto **java.awt.Graphics** representa um determinado dispositivo de saída do sistema.

O sistema de coordenadas do objeto **java.awt.Graphics** tem como unidade de medida o pixel do dispositivo de saída e possui o seguinte referencial: a origem ocupa o canto superior esquerdo do dispositivo, cuja abcissa é crescente horizontalmente para direita e a ordenada é crescente verticalmente para baixo conforme ilustrado a seguir.

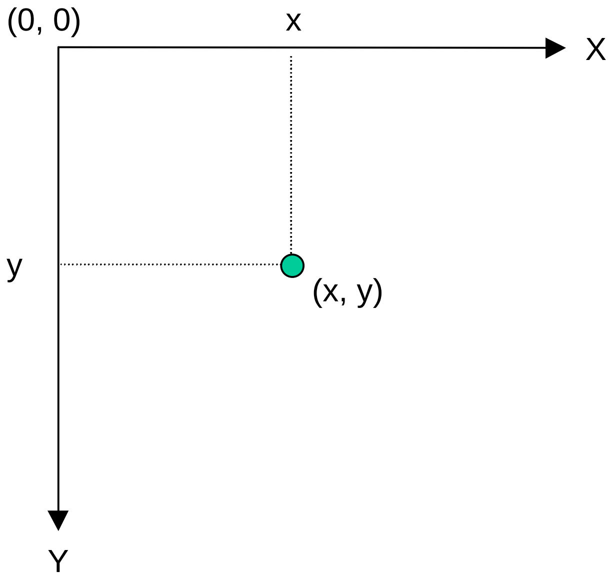


Figura 53 Sistema de Coordenadas do Java

Toda a renderização efetuada num contexto gráfico é realizada por uma caneta imaginária, cujo traço tem espessura de um pixel e segue as seguintes regras:

- as operações de desenho operam sobre áreas retangulares (mesmo que o efetivo resultado não seja um retângulo) e são realizadas da esquerda para direita e de cima para baixo;
- as operações de preenchimento são realizadas da esquerda para direita e de cima para baixo;
- a renderização de texto é realizada em duas etapas sendo que a primeira desenha a parte ascendente dos caracteres (parte acima da linha base) e depois é desenhada a parte descendente;
- todas as operações são realizadas com a cor corrente, usando o modo de pintura corrente e o fonte corrente tomando como referência a origem do sistema de coordenadas.

Além da classe **java.awt.Graphics**, existe a classe **java.awt.Toolkit** que provê métodos especiais para obter informações auxiliares sobre o sistema gráfico, tais como a resolução atual da tela e o conjunto de fontes suportado.

A seguir temos uma relação dos principais métodos disponíveis na classe **java.awt.Graphics**:

Método	**Descrição**
drawArc(int, int, int, int, int, int)	Desenha um arco circular ou elíptico cobrindo o retângulo dado.
drawImage(Image, int, int, IO)	Desenha uma imagem a partir do ponto dado, tanto quanto disponível.
drawLine(int, int, int, int)	Desenha um linha entre os pontos dados.
drawOval(int, int, int, int)	Desenha uma elipse inscrita no dado retângulo.
drawPolygon(int[], int[], int)	Desenha um polígono fechado definido pelos vetores de coordenadas x e y dados.
drawRect(int, int, int, int)	Desenha um retângulo.
drawString(String, int, int)	Desenha a *string* dada a partir do ponto dado.
fillArc(int, int, int, int, int, int)	Preenche o arco dado com a cor corrente.
fillOval(int, int, int, int)	Preenche a elipse dado com a cor corrente.
fillPolygon(int[], int[], int)	Preenche o polígono dado com a cor corrente.
fillRect(int, int, int, int)	Preenche o retângulo dado com a cor corrente.
getColor()	Obtêm a cor corrente.
getFont()	Obtêm o fonte corrente.
setColor(Color)	Especifica a cor corrente.
setFont(Font)	Especifica o fonte corrente.
translate(int, int)	Translada a origem do sistema de coordenadas para o dado ponto.

Tabela 52 Métodos da Classe java.awt.Graphics

Notem a existência de métodos destinado ao desenho de formas geométricas planas, controle de cor e fonte.

7.2 Primitivas Gráficas

Vejamos agora alguns exemplos de aplicações gráficas que utilizem o contexto gráfico e as primitivas da classe **java.awt.Graphics**.

7.2.1 Linhas

O método **drawLine** pode ser utilizado para desenhar linhas retas entre dois pontos (x1, y1) e (x2, y2) dados. Veremos dois exemplos de aplicação desta primitiva.

O primeiro exemplo traça um reticulado na área livre da janela, isto é, desenha linhas horizontais e verticais igualmente espaçadas como uma grade. O espaçamento *default* da grade é de 20 unidades mas pode ser modificado através do primeiro argumento da aplicação. O código pode ser visto no Exemplo 74.

```
// Reticula.java

import java.awt.*;

public class Reticula extends Frame {
  private int size;

  public static void main(String args[]) {
    int s = (args.length>0 ? Integer.parseInt(args[0]) : 20);
    Reticula f = new Reticula(s);
```

```
    f.show();
  }

  public Reticula(int s) {
    super("Retícula");
    setSize(200, 200);
    size = s;
    addWindowListener(new CloseWindowAndExit());
  }

  public void paint(Graphics g) {
    Dimension dim = getSize();
    for (int i=0; i<dim.width; i+=size)
      g.drawLine(i, 0, i, dim.height);
    for (int i=0; i<dim.height; i+=size)
      g.drawLine(0, i, dim.width, i);
  }
}
```

Exemplo 74 Classe Reticula.java

Executando-se esta aplicação obtêm-se o resultado ilustrado na

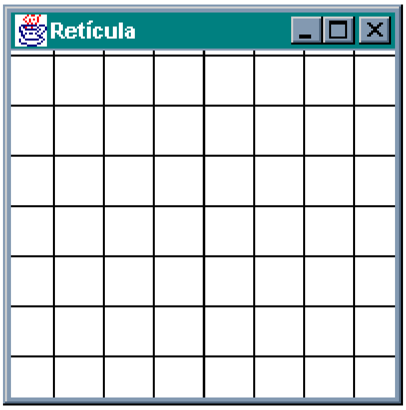


Figura 54 Aplicação Reticula

O segundo exemplo, listado a seguir, desenha 5 linhas em posições aleatórias efetuando a sobreposição do método **paint** associado ao componente **Frame** sobre o qual é implementada a aplicação. A aplicação também "imprime" o número da linha a partir da mesma coordenada de origem de cada linha utilizando o método **drawString**.

```
// Lines.java

import java.awt.*;
import java.awt.event.*;

public class Lines extends Frame {
  // main
  public static void main(String args[]) {
    Lines f = new Lines();
    f.show();
  }

  // construtor
  public Lines() {
    super("Linhas");
    setSize(200, 200);
    setLocation(50, 50);
    addWindowListener(new CloseWindowAndExit());
  }
```

```
  // paint
  public void paint(Graphics g) {
    for (int i=0; i<5; i++) {
      int x1 = (int) Math.round(Math.random()*200);
      int y1 = (int) Math.round(Math.random()*200);
      g.drawString(""+(i+1), x1, y1);
      int x2 = (int) Math.round(Math.random()*200);
      int y2 = (int) Math.round(Math.random()*200);
      g.drawLine(x1, y1, x2, y2);
    }
  }
}
```

Exemplo 75 Classe Lines

Compilando-se e executando-se esta aplicação obtemos o resultado ilustrado na Figura 55. Note que ao redimensionarmos ou encobrirmos a janela, as linhas mudam de posição, mostrando que o método **paint** é automaticamente acionado toda vez que é necessário redesenhar a janela da aplicação.

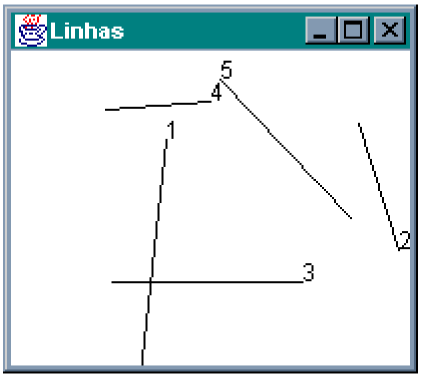


Figura 55 Aplicação Lines

Uma dica é a utilização do método **drawLine** para desenhar-se pontos simples, o que é realizado mais eficientemente do que através de outros métodos, por exemplo **fillRect**. Para desenhar um ponto na posição (x, y) poderíamos escrever:

```
g.drawLine(x, y, x, y);
```

O ponto assim desenhado utiliza a cor corrente do contexto gráfico, que pode ser alterada através do método **setColor**.

7.2.2 Retângulos e Quadrados

O método **drawRect** pode ser utilizado para desenhar retângulos ou quadrados através da definição da localização de seu vértice superior esquerdo (x, y) e da definição de sua largura *width* e altura *height*, como indicado:

```
g.drawRect(x, y, width, height);
```

Apresentaremos um exemplo mais sofisticado de aplicação onde utilizamos esta primitiva. A apresentação desenha quadrados de tamanho ajustável pelo usuário usando como posição do vértice superior esquerdo o local onde o usuário clica a aplicação. Diferentemente da aplicação do Exemplo 75, a posição dos quadrados é armazenada num vetor auto-ajustável (um objeto da classe **java.util.Vector**) através de objetos que contêm a posição e tamanho dos quadrados definidos como uma classe interna (*inner class*) **BoxInfo**. Desta forma o método **paint**, mesmo quando acionado desenha novamente os quadrados nas posições definidas pelo usuário, tal como programas comuns de desenho e outras aplicações gráficas.

Outra característica da aplicação é a possibilidade do usuário desenhar quadrados sólidos. Isto é feito utilizando-se o método **fillRect** ao invés de **drawRect** que recebe os mesmos argumentos (posição do vértice superior esquerdo e dimensões do retângulo) utilizando a cor corrente do contexto gráfico:

```
g.fillRect(x, y, width, height);
```

O usuário seleciona o desenho de quadrados vazados ou sólidos utilizando uma caixa de opção. Um botão extra possibilita redefinir o objeto **Vector** limpando a área de desenho definida pela janela da aplicação.

Os principais construtores e métodos da classe **java.util.Vector** são:

Método	**Descrição**
Vector()	Constrói um objeto **Vector** inicialmente vazio.
Vector(int)	Constrói um objeto **Vector** com a capacidade indicada.
Vector(int, int)	Constrói um objeto **Vector** com a capacidade e o incremento de capacidade indicados.
addElement(Object)	Adiciona o objeto ao vetor.
capacity()	Retorna a capacidade atual do vetor.
elementAt(int)	Retorna o objeto contido na posição dada.
insertElementAt(Objecto, int)	Insere o objeto dado na posição indicada.
removeAllElements()	Remove todos os elementos do vetor.
removeElementAt(intO	Remove o elemento da posição indicada.
setElementAt(Object, int)	Substitui o elemento da posição indicada pelo objeto fornecido.
setSize(int)	Especifica a capacidade do vetor.
size()	Retorna o número atual de elementos contidos no vetor.
trim(int)	Reduz a capacidade do vetor para o número de elementos contidos do vetor.

Tabela 53 Métodos da Classe java.util.Vector

Os objetos **Vector** armazenam objetos de qualquer tipo sendo extremamente flexíveis embora as operações sobre os objetos desta classe sejam mais lentas do que as operações em vetores comuns (*arrays*). A capacidade do vetor é a máxima quantidade de elementos que ele pode armazenar num dado instante. Seu tamanho corresponde a quantidade de elementos efetivamente armazenados. Tendo toda a sua capacidade ocupada (tamanho = capacidade), a adição de um novo elemento provoca o aumento automático da capacidade do vetor. Se um valor de incremento é especificado através de seu construtor, tal aumento se dá com esta granulosidade, caso contrário a capacidade é duplicada.

Estude o código fornecido para esta aplicação observando a utilização da classe **Vector**, da classe interna **BoxInfo** e da interface **MouseListener**.

```
// Boxes.java

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Boxes extends Frame
                   implements ActionListener, MouseListener {

  private Button bBigger, bSmaller, bClear;
  private TextField tfSize;
  private Checkbox cbFill;
  private Vector theBoxes;
```

```
  private int boxSize;

  public static final void main(String args[]) {
    Boxes b = new Boxes();
    b.show();
  }

  public Boxes() {
    super("Boxes");
    setSize(320, 240);
    boxSize = 10;
    theBoxes = new Vector(20);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(bSmaller = new Button("<<"));
    bSmaller.addActionListener(this);
    p.add(tfSize = new TextField(""+boxSize, 3));
    tfSize.setEditable(false);
    tfSize.setBackground(SystemColor.control);
    p.add(bBigger = new Button(">>"));
    bBigger.addActionListener(this);
    p.add(bClear = new Button("Limpar"));
    p.add(cbFill = new Checkbox("Sólido"));
    bClear.addActionListener(this);
    add("South", p);
    addMouseListener(this);
    addWindowListener(new CloseWindowAndExit());
  }

  public void paint(Graphics g) {
    for (int i=0; i<theBoxes.size(); i++) {
      BoxInfo box = (BoxInfo) theBoxes.elementAt(i);
      if (box.solid)
        g.fillRect(box.x, box.y, box.size, box.size);
      else
        g.drawRect(box.x, box.y, box.size, box.size);
    }
  }

  // Interface ActionListener
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==bSmaller)
      boxSize -= (boxSize>1 ? 1 : 0);
    else if (e.getSource()==bBigger)
      boxSize += (boxSize<100 ? 1 : 0);
    else {
      theBoxes = new Vector(20);
      repaint();
    }
    tfSize.setText("" + boxSize);
  }

  // Interface MouseListener
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
  public void mouseClicked(MouseEvent e) {
    theBoxes.addElement(new BoxInfo(e.getX(), e.getY(),
                                   boxSize, cbFill.getState()));
    repaint();
```

```
  }

  // InnerClass (Classe Interna)
  class BoxInfo {
    public int x;
    public int y;
    public int size;
    public boolean solid;

    public BoxInfo(int a, int b, int c, boolean d) {
      x = a; y = b; size = c; solid = d;
    }
  }
}
```

Exemplo 76 Classe Boxes

A execução da aplicação possibilita a obtenção de resultados como ilustrados na Figura 56.

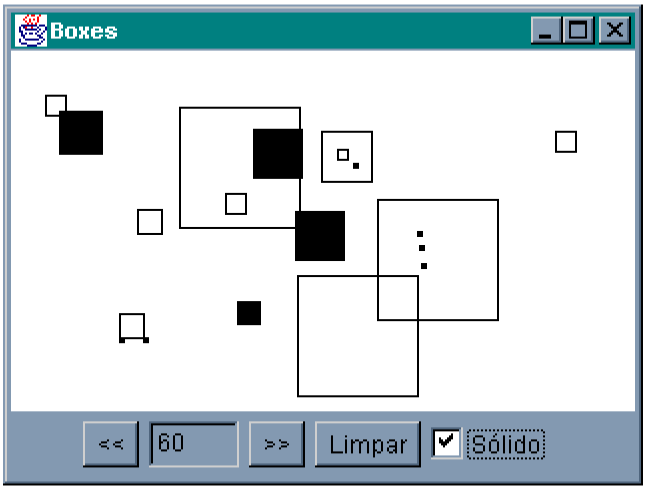


Figura 56 Aplicação Boxes

7.2.3 Elipses e Circunferências

O método **drawOval** pode ser utilizado para desenhar elipses ou circunferências de forma análoga ao método **drawRect**, ou seja, através da definição da localização do vértice superior esquerdo (x, y) do retângulo onde se inscreve a elipse e da definição de sua largura *width* e altura *height*, como indicado:

```
g.drawOval(x, y, width, height);
```

Dispõe-se também do método **fillOval**, capaz de produzir circunferências e elipses sólidas. O exemplo abaixo mostra o uso do método **fillOval**.

```
// Ovals.java

import java.awt.*;
import java.awt.event.*;

public class Ovals extends Frame
                   implements ActionListener {

  private Button bStart, bClear;
  private Point coord[];

  public static final void main(String args[]) {
    Ovals b = new Ovals();
```

```
    b.show();
  }

  public Ovals() {
    super("Ovals");
    setSize(320, 240);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(bStart = new Button("Iniciar"));
    bStart.addActionListener(this);
    p.add(bClear = new Button("Limpar"));
    bClear.addActionListener(this);
    add("South", p);
    addWindowListener(new CloseWindowAndExit());
  }

  public void paint(Graphics g) {
    if (coord==null)
      return;
    for (int i=0; i<coord.length; i++) {
      g.drawOval(coord[i].x, coord[i].y, 50, 50+2*i);
      try {
        Thread.sleep(250);
      } catch (Exception e) {
      }
    }
  }

  // Interface ActionListener
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==bStart) {
      coord = new Point[20];
      for (int i=0; i<20; i++)
        coord[i] = new Point((int)i*i/2, i*3);
    } else {
      coord = null;
    }
    repaint();
  }
}
```

Exemplo 77 Classe Ovals

A aplicação oferece botões para definição das coordenadas de 20 elipses ou para sua eliminação, de forma que o método **paint** seja acionado controladamente. O desenho de cada elipse é propositadamente retardado através da chamada de **Thread.sleep**, que faz com que execução do programa seja interrompida de tantos milisegundos quando especificados. Este procedimento não é indicado para aplicações mais sérias pois degrada a performance do sistema dado que o método **paint** é com frequência acionado quando a janela é parcial ou totalmente coberta e também quando é redimensionada. Sugere-se uma estratégia semelhante a aplicação **Boxes** (Exemplo 76).

Note também o uso de um vetor comum (*array*) de objetos **Point** que permitem armazenar as definições de um ponto (x, y) qualquer. O resultado da aplicação é ilustrado na Figura 57.

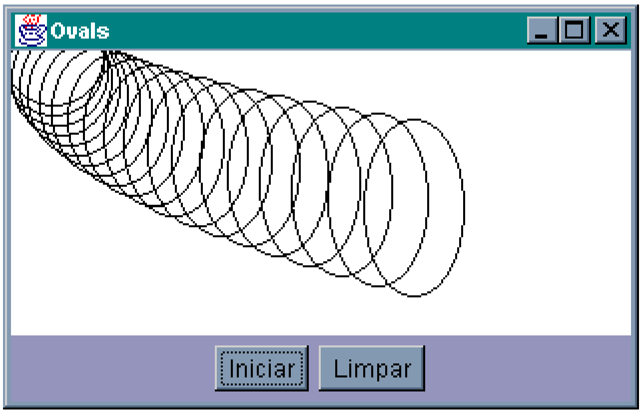


Figura 57 Aplicação Ovals

7.2.4 Polígonos

Para desenhar-se polígonos regulares ou irregulares utilizamos os métodos **drawPolygon** ou **fillPolygon** que permitem a obtenção de figuras vazadas ou sólidas respectivamente. Estes métodos podem receber dois vetores comuns que contem as coordenadas x e y de cada ponto do polígono e um valor inteiro que define a quantidade de pontos. Com estas informações Os métodos **drawPolygon** e **fillPolygon** desenham um polígono fechado.

Para desenhar-se um triângulo poderíamos escrever:

```
int x[] = new int[3];
int y[] = new int[3];
x[0] =  0; y[0] =  0; // ponto ( 0, 0)
x[0] = 50; y[0] =  0; // ponto (50, 0)
x[0] =  0; y[0] = 50; // ponto ( 0,50)
g.drawPolygon(x, y, 3);
```

O código exemplificado em seguida é de uma aplicação que recebe até 20 pontos (x, y) os quais são renderizados a partir do segundo ponto sempre como uma figura fechada. Se os dados forem inválidos a aplicação emite um sinal sonoro através da método **beep** disponível na classe **java.awt.Toolkit**.

```
// PolyDemo.java

import java.awt.*;
import java.awt.event.*;

public class PolyDemo extends Frame
                      implements ActionListener {

  private TextField tfX, tfY;
  private Button bDefine, bClear;
  private int x[], y[];
  private int points;

  public static final void main(String args[]) {
    PolyDemo f = new PolyDemo();
    f.show();
  }

  public PolyDemo() {
    super("Polygon Demo");
    setSize(320, 240);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(tfX = new TextField(3));
```

```
    p.add(tfY = new TextField(3));
    p.add(bDefine = new Button("Definir"));
    bDefine.addActionListener(this);
    p.add(bClear = new Button("Limpar"));
    bClear.addActionListener(this);
    add("South", p);
    addWindowListener(new CloseWindowAndExit());
    x = new int[20];
    y = new int[20];
    points = 0;
  }

  public void paint(Graphics g) {
    if (points==0)
      return;
    g.drawPolygon(x, y, points);
  }

  // Interface ActionListener
  public void actionPerformed(ActionEvent e) {
    if (e.getSource()==bDefine) {
      if (points < 20) {
        try {
          x[points] = Integer.parseInt(tfX.getText());
          y[points] = Integer.parseInt(tfY.getText());
          points++;
          tfX.setText("");
          tfY.setText("");
        } catch (NumberFormatException exc) {
          Toolkit.getDefaultToolkit().beep();
        }
      }
    } else {
      x = new int[20];
      y = new int[20];
      points = 0;
    }
    repaint();
  }
}
```

Exemplo 78 Classe PolyDemo

A aplicação quando executada pode produzir o seguinte resultado.

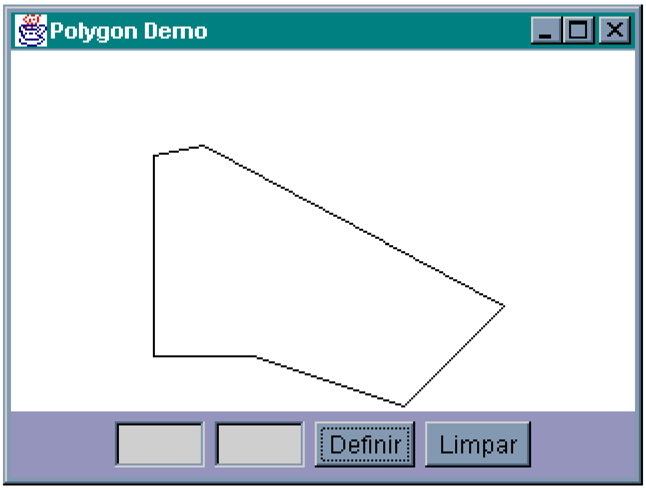


Figura 58Aplicação PolyDemo

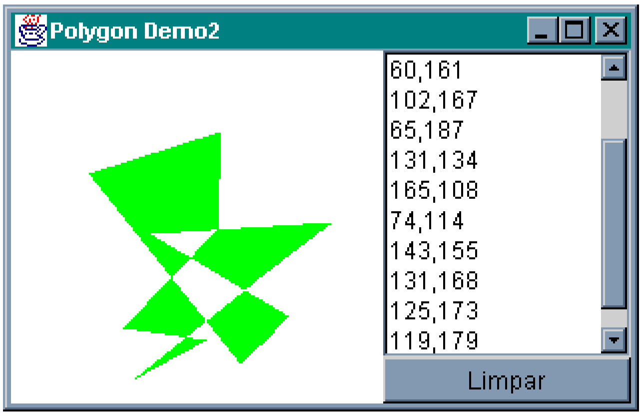
Algumas pequenas alterações no Exemplo 78 podem fazer que a definição do polígono utilize cliques do *mouse* ao invés da digitação das coordenadas, listando os pontos fornecidos numa caixa de lista. A figura desenhada agora é sólida pois utiliza o método **fillPolygon**.

```
// PolyDemo2.java

import java.awt.*;
import java.awt.event.*;

public class PolyDemo2 extends Frame
                       implements ActionListener, MouseListener{

  private List ltPoints;
  private Button bClear;
  private int x[], y[];
  private int points;

  public static final void main(String args[]) {
    PolyDemo2 f = new PolyDemo2();
    f.show();
  }

  public PolyDemo2() {
    super("Polygon Demo2");
    setSize(320, 240);
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    p.add("Center", ltPoints = new List());
    p.add("South", bClear = new Button("Limpar"));
    bClear.addActionListener(this);
    add("East", p);
    addMouseListener(this);
    addWindowListener(new CloseWindowAndExit());
    x = new int[20];
    y = new int[20];
    points = 0;
  }

  public void paint(Graphics g) {
    if (points==0)
      return;
    g.setColor(Color.green);
    g.fillPolygon(x, y, points);
  }

  // Interface ActionListener
  public void actionPerformed(ActionEvent e) {
    ltPoints.removeAll();
    x = new int[20];
    y = new int[20];
    points = 0;
    repaint();
  }

  // Interface MouseListener
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
  public void mouseClicked(MouseEvent e) {
    x[points] = e.getX();
```

```
    y[points] = e.getY();
    points++;
    ltPoints.add("" + e.getX() + "," + e.getY());
    repaint();
  }
}
```

Exemplo 79 Classe PolyDemo2

O resultado desta aplicação é ilustrado a seguir.



```
60,161
102,167
65,187
131,134
165,108
74,114
143,155
131,168
125,173
119,179
```

Figura 59 Aplicação PolyDemo2

7.3 Animação Simples

A construção de uma animação simples com o Java é uma tarefa pouco complexa. Uma estratégia bastante utilizada para animação é a seguinte:

(i) Uma sequência de imagens, usualmente arquivos ".gif", é carregada num vetor (*array*) no construtor da classe;
(ii) O método **paint** exibe uma imagem, avança o contador de imagens para a próxima imagem e entra em espera e
(iii) Após a espera é acionado o método **repaint**.

Um vetor de imagens pode ser declarado assim:

```
private Image imagens[];
```

A carga de imagens num vetor pode ser realizada como no trecho a seguir utilizando-se o método **getImage** do **Toolkit** associado a plataforma corrente. A variável **numImagens** contêm o quantidade de imagens a ser carregada. Supomos também que as imagens possuem um mesmo nome que inclui a numeração associada [1, **numImagens**]:

```
imagens = new Image[numImagens];
Toolkit t = Toolkit.getDefaultToolkit();
for (int i=1; i<=numImagens; i++)
  imagens[i] = t.getImage("T" + i + ".gif");
```

A exibição das imagens é realizada no método **paint** através do método **drawImage** associado ao contexto gráfico da aplicação:

```
g.drawImage(imagens[imagemAtual],x, y, this);
```

Note que a imagem de número **imagemAtual** é desenhada na posição x, y da janela (**Frame**) que é notificada desta ação (via referência **this**). A variável **imagemAtual** deve ser incrementada indicando que a exibição avançou para a próxima imagem mas garantindo-se que tal numeração se restrinja ao intervalo [0, **numImagens**–1]:

```
imagemAtual = (imagemAtual + 1) % numImagens;
```

Para que a exista um intervalo apropriado entre a exibição de cada imagem utiliza-se o método **sleep** que força uma pausa da aplicação, na verdade da *thread* (fluxo de execução) corrente:

```
Thread.sleep(tempoEspera);
```

Desta forma o código completo para produzir uma animação com uma série de imagens “gif” é dado no Exemplo 80.

```
// Movie.java

import java.awt.*;
import java.awt.event.*;

public class Movie extends Frame {
  private Image imagens[];
  private int tempoEspera = 150;
  private int numImagens = 10;
  private int imagemAtual;

  public static void main(String args[]) {
    Movie f = new Movie();
    f.show();
  }

  public Movie() {
    super("Movie");
    setSize(80, 110);
    imagens = new Image[numImagens];
    Toolkit t = Toolkit.getDefaultToolkit();
    for (int i=1; i<=numImagens; i++)
      imagens[i] = t.getImage("T" + i + ".gif");
    imagemAtual = 0;
    addWindowListener(new CloseWindowAndExit());
  }

  public void paint(Graphics g) {
    g.drawImage(imagens[imagemAtual],
                getInsets().left, getInsets().top, this);
    imagemAtual = (imagemAtual + 1) % numImagens;
    try {
      Thread.sleep(tempoEspera);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    repaint();
  }
}
```

Exemplo 80 Classe Movie

Resultados possíveis desta aplicação são ilustrados a seguir.

Figura 60 Aplicação Movie (Duke)

Através de pequenas modificações podemos fazer que a aplicação aceite como parâmetros a localização e prefixo do nome das imagens bem como o número de imagens que compõe a animação e o intervalo entre a exibição das imagens. Outras modificações interessantes seriam o ajusta automático do tamanho da janela para o tamanho das figuras e o controle de início e parada da animação propriamente dita.

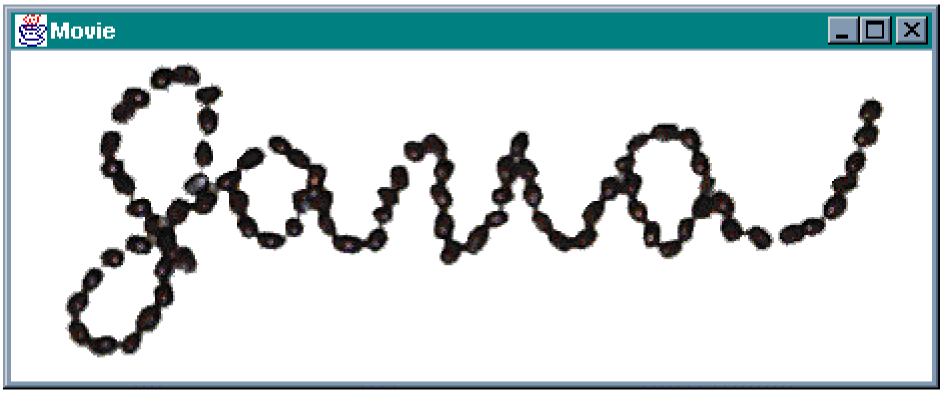


Figura 61 Aplicação Movie (Beans)

7.4 Melhorando as Animações

A aplicação desenvolvida do Exemplo 80 utiliza a estratégia mais simples possível para construção de animações. Se observarmos atentamente, quando utilizamos imagens maiores ocorrem dois problemas: (i) um pequeno efeito de cintilação (*flickering*) devido a renderização das imagens diretamente na janela através do método **drawImage** e (ii) algumas irregularidades na velocidade de exibição pela mesma razão.

Para resolvermos estes problemas, melhorando a qualidade da animação produzida utilizamos uma técnica denominada *double buffering*, onde o trabalho de renderização é feito em memória sendo que a imagem preparada é transferida diretamente para a janela acelerando a exibição e, portanto, suavizando a transição entre as imagens.

A técnica de *double buffering* consome substancialmente mais memória que a técnica de animação simples mas este é preço para obtenção de melhor performance e qualidade. Para utilização desta técnica devemos seguir os seguintes passos:

(i) cria-se uma imagem "em branco" (que possui seu próprio contexto gráfico);
(ii) desenha-se a imagem desejada na imagem "em branco", utilizando o contexto gráfico desta e
(iii) efetua-se a exibição da imagem já renderizada.

Desta forma precisamos declarar uma imagem e um contexto gráfico para suportar a implementação do *double buffering*:

```
private Image buffer;
private Graphics gContexto;
```

A criação do *buffer* e o preparo do contexto gráfico (definição da cor de fundo, limpeza da área a ser desenhada e desenho da primeira imagem) podem ser realizados como abaixo.

```
public void windowOpened(WindowEvent e) {
  // Cria buffer
  buffer = createImage(400, 160);
  // obtem e prepara contexto grafico associado
  gContexto = buffer.getGraphics();
  gContexto.setColor(Color.white);
  gContexto.fillRect(0, 0, 100, 100);
  // desenha primeira imagem no buffer
  gContexto.drawImage(imagens[0], 0, 0, this);
  imagemAtual = 1;
}
```

Note que tal rotina deve ser colocada dentro o método **windowOpened**, pois somente depois de criada e exibida a janela é que podemos obter seu contexto gráfico. Isso exige que a aplicação implemente a interface **java.awt.event.WindowListener** e portanto de seus outros métodos.

O método **paint** da aplicação também sofre uma pequena alteração, que é o uso do *buffer* para apresentação da imagem e preparo do *buffer* para apresentação da próxima imagem:

```
public void paint(Graphics g) {
  // renderiza buffer
  g.drawImage(buffer, getInsets().left, getInsets().top, this);
  // limpa buffer e desenha proxima imagem
  gContexto.fillRect(0, 0, 400, 160);
  gContexto.drawImage(imagens[imagemAtual], 0, 0, this);
  imagemAtual = (imagemAtual + 1) % numImagens;
  try {
    Thread.sleep(tempoEspera);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  repaint();
}
```

Segue a modificação completa do Exemplo 80 para a aplicação desta técnica para construção de animações.

```
// Movie2.java

import java.awt.*;
import java.awt.event.*;

public class Movie2 extends Frame implements WindowListener{
  private Image imagens[];
  private Image buffer;
  private Graphics gContexto;
  private int tempoEspera = 150;
  private int numImagens = 10;
  private int imagemAtual;

  public static void main(String args[]) {
    Movie2 f = new Movie2();
    f.show();
  }

  public Movie2() {
    super("Movie2");
    setSize(470, 180);
    imagens = new Image[numImagens];
    Toolkit t = Toolkit.getDefaultToolkit();
    // obtêm imagens da animacao
    for (int i=0; i<numImagens; i++)
      imagens[i] = t.getImage("Beans/T" + (i+1) + ".gif");
    addWindowListener(this);
  }

  public void paint(Graphics g) {
    // renderiza buffer
    g.drawImage(buffer, getInsets().left, getInsets().top,
this);
    // limpa buffer e desenha proxima imagem
    gContexto.fillRect(0, 0, 400, 160);
    gContexto.drawImage(imagens[imagemAtual], 0, 0, this);
    imagemAtual = (imagemAtual + 1) % numImagens;
    try {
      Thread.sleep(tempoEspera);
    } catch (InterruptedException e) {
      e.printStackTrace();
```

```
    }
    repaint();
  }

  // Interface WindowListener
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowOpened(WindowEvent e) {
    // Cria buffer
    buffer = createImage(400, 160);
    // obtem e prepara contexto grafico associado
    gContexto = buffer.getGraphics();
    gContexto.setColor(Color.white);
    gContexto.fillRect(0, 0, 400, 160);
    // desenha primeira imagem no buffer
    gContexto.drawImage(imagens[0], 0, 0, this);
    imagemAtual = 1;
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}
```

Exemplo 81 Classe Movie2

8 Applets

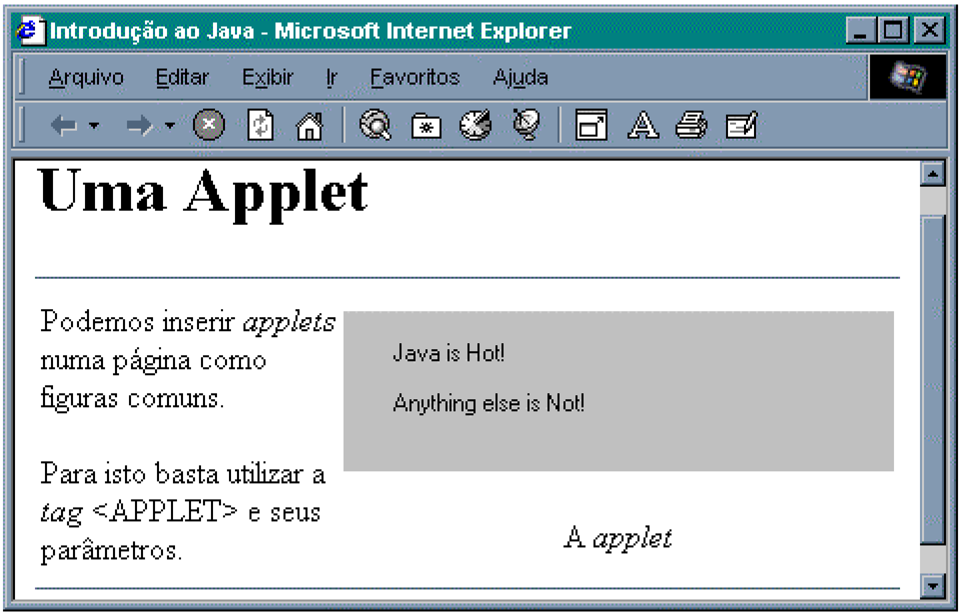
Uma *applet* é um programa Java que é executado por um *browser* (um navegador WWW) quando é carregada a página que contêm tal *applet*. Desta forma uma *applet* é um programa Java destinado a ser utilizado pelos browsers significando que será transportada pela Internet tal como documentos HTML, imagens GIF e JPEG e outros conteúdos típicos da rede.

Em função deste uso as *applets* usualmente são construídas para serem pequenos programas e, por questões de segurança, obedecem critérios rígidos para que sua execução seja possível pelos browsers.

Outro aspectos é que as *applets* são programas executados nos ambientes gráficos das diversas plataformas que utilizam a WWW, assim sua construção é bastante semelhante a criação de programas que utilizem componentes ou recursos gráficos disponíveis na AWT. Abaixo um pequeno exemplo de *applet* e seu resultado quando na execução num *browser*.

```
import java.applet.Applet;
import java.awt.Graphics;

public class PMPWelcome extends Applet {
  public void paint( Graphics g){
    g.drawString("Java is Hot!", 25, 25);
    g.drawString("Anything else is Not!", 25, 50);
  }
}
```

Exemplo 82 Classe PMPWelcome



Uma Applet

Podemos inserir *applets* numa página como figuras comuns.

Para isto basta utilizar a *tag* <APPLET> e seus parâmetros.

Java is Hot!

Anything else is Not!

A applet

Figura 62 *Applet* PMPWelcome (*browser* MS Internet Explorer)

O código HTML necessário para produzir-se a página exibida na Figura 62 é reproduzido abaixo:

```
<html>
<head>
  <title>Introdução ao Java</title>
</head>
<body>
  <h1>Uma Applet</h1>
  <hr>
  <table border=0 width="100%">
    <tr>
      <td>
        <p>Podemos inserir <i>applets</i> numa página
           como figuras comuns.</p>
        <p>Para isto basta utilizar a <i>tag</i> &ltAPPLET&gt
           e seus parâmetros.</p>
      </td>
      <td>
        <applet code="PMPWelcome.class" width=275 height=80>
        </applet>
        <p align=center>A <i>applet</i></p>
      </td>
    </tr>
  </table>
  <hr>
</body>
```

Exemplo 83 Arquivo PMPWelcome.html

Notamos tratar-se de um arquivo HTML ordinário onde a inclusão da *applet* é realizada através da *tag* <APPLET>:

```
<applet code="PMPWelcome.class" width=275 height=80>
</applet>
```

Como podemos notar, a *tag* <APPLET> possui vários campos os quais indicam:

Campo	Descrição
ARCHIVE	Nome do arquivo compactado da *applet* (Java *Archive* - arquivo ".jar")
ALT	Nome alternativo.
ALIGN	Alinhamento da *applet* (valores possíveis são: *top*, *middle*, *bottom*, *left* e *right*).
CODE	Nome do arquivo de classe Java incluindo extensão ".class".
CODEBASE	URI base para os arquivos de classe Java.
HEIGHT	Altura da *applet* (em pixels).
HSPACE	Margem horizontal (em pixels).
NAME	Nome da *applet* para comunicação inter-*applet*.
OBJECT	Nome do arquivo da *applet* serializada (arquivo ".ser").
VSPACE	Margem vertical (em pixels).
WIDTH	Largura da *applet* (em pixels).

Tabela 54 Campos da *Tag* <APPLET>

É necessário utilizar-se o campo CODE ou OBJECT para especificar-se a *applet* que deve ser executada. O campo ARCHIVE é usado caso a *applet* esteja inserida num arquivo compactado tipo JAR. O campo CODEBASE indica o local onde os arquivos ".class", ".jar' ou ".ser" estão localizados. Os demais campos referem-se a disposição da *applet* na página HTML. A forma mais frequente de uso desta *tag* é como exemplificado, ou seja, especificando-se a *applet* através do campo CODE e seu tamanho através do campos HEIGHT e WIDTH.

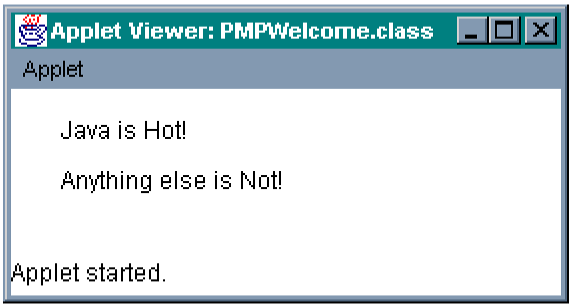
A *tag* <APPLET> é uma *tag* dupla portanto finalizada por </APPLET>. Em seu corpo, isto é, no interior da construção <APPLET> </APPLET> podemos inserir outra *tag*, <PARAM> destinada a parametrização da *applet*, como será visto na seção 8.4.

A programação e compilação de *applets* é uma tarefa idêntica a criação de programas em Java: utiliza-se um editor de textos comum e depois o compilador javac para realizar a compilação do programa nos *bytecodes* correspondentes. Novamente o nome do arquivo deve ser idêntico ao da classe pública nele contido.

No entanto o teste das *applets* produzidas não podem ser testadas através do interpretador java ou do ambiente de execução jre, mas por browser compatíveis com o Java ou através do utilitário *appletviewer* fornecido juntamente com as demais ferramentas do *Sun* JDK. O *appletviewer* executa *applets* que estejam inseridas em arquivos HTML, por exemplo:

```
appletviewer PMPWelcome.html
```

Isto produziria o seguinte resultado para a *applet* PMPWelcome:



```
Java is Hot!

Anything else is Not!
```

Applet started.

Figura 63 *Applet* PMPWelcome (appletviewer)

Não existe necessidade do arquivo HTML possuir o mesmo nome que a *applet* ou sua classe. Outra facilidade é que o arquivo HTML pode conter apenas a *tag* <APPLET> tal como ilustrado anteriormente embora consiga tratar arquivos HTML completos. Assim uso do appletviewer é mais conveniente durante a fase de desenvolvimento e teste das *applets*. Se existirem várias *applets* numa página Web, o appletviewer inicia várias janelas, uma para cada *applet* identificada.

8.1 Funcionamento das Applets

Embora a construção de *applets* seja semelhante a criação de programas gráficos em Java, a compreensão de seu ciclo de vida auxilia bastante no entendimento de sua estrutura. As *applets* são aplicações especiais que possuem um modo particular de funcionamento:

(i) instanciação (*create*)
(ii) inicialização (*init*)
(iii) início (*start*)
(iv) execução e renderização (*paint* e outros métodos)
(v) parada (*stop*)
(vi) finalização ou destruição (*destroy*)

Este modo de funcionamento define o ciclo de vida das *applets*, que é esquematizado na Figura 64.

Implementações mínimas de *applets* utilizam geralmente o método **init** ou o método **paint**.

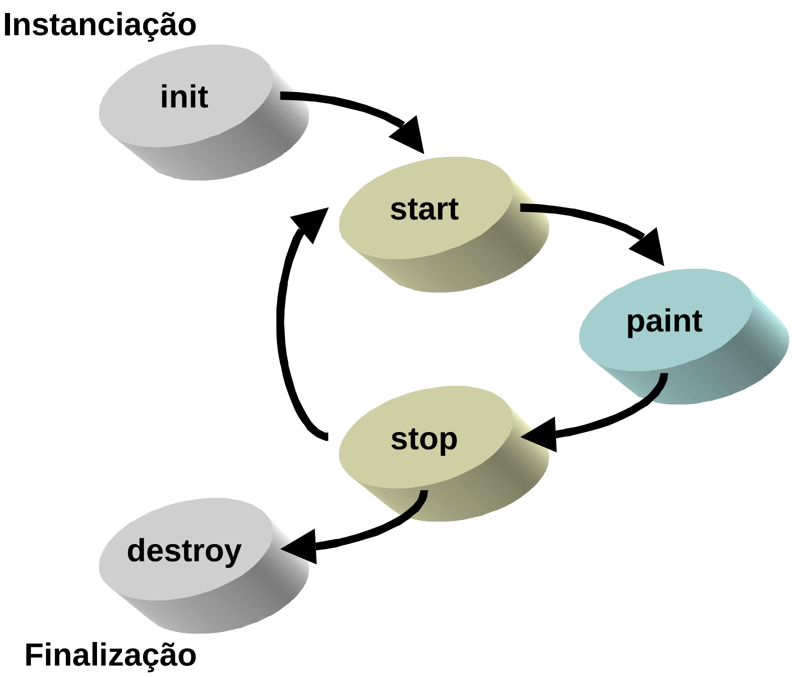


Figura 64 Ciclo de Vida das *Applets*

8.1.1 O Método *init*

Tal como programas, as *applets* são objetos definidos em classes. Enquanto os programas necessitam de um método **main**, implementado em uma de suas classes, para definir o início do programa (que provocará a instanciação e exibição de um **Frame** ou a execução de uma aplicação de console), as *applets* são instanciadas pelos próprios navegadores, ou melhor, pelas JVM encarregadas da execução das *applets*. Como qualquer classe Java, os construtores não necessitam ser explicitamente implementados dado a existência dos construtores default disponibilizados pelo Java.

Adicionalmente existem um método denominado **init**, invocado após a criação da *applet* (instanciação), utilizado pelos navegadores para iniciar a execução das *applets*. Este método é normalmente usado para adição de componentes, recebimento de parâmetros de execução e outras atividades de preparo. Este método é executado somente uma única vez.

8.1.2 O Método *start*

O método **start** é acionado pelo *browser* toda vez que *applet* torna-se visível, portanto deve ser utilizado quando é necessário assegurar alguma condição especial para a apresentação da *applet*. Geralmente este método é utilizado em conjunto com o método **stop**. O método **start** é acionado pela primeira vez logo após a execução do método **init**, sendo novamente acionado tantas vezes quantas a *applet* tornar-se visível.

8.1.3 O Método *paint*

Este método é chamado toda vez que a *applet* necessita atualizar sua exibição o que ocorre quando é exibida pela primeira vez (depois da execução do método **start**), quando a janela do browser é movimentada ou redimensionada como consequência do acionamento do método **update**.

O método **paint** pode ser utilizado para a realização de operações de desenho especiais na *applet*, ou seja, quando deseja-se tratar diretamente a renderização do conteúdo exibido pela *applet*, tal como no caso da construção de animações ou outras aplicações gráficas.

A maneira mais simples de construir-se uma *applet* é através da utilização do método **paint**, pois através deste temos acesso ao contexto gráfico da *applet* que permite a utilização das primitivas gráficas da AWT conforme visto na seção 7 Aplicações e Primitivas Gráficas.

8.1.4 O Método *stop*

Como complemento funcional do método **start**, dispomos do método **stop**. Este método é ativado toda a vez que a *applet* deixa de ser visível, ou seja, quando ocorre um rolamento (*scrolling*) da página exibida pelo *browser* ou quando a *applet* se torna encoberta por outra janela do sistema.

Isto significa que a *applet*, durante seu ciclo de vida, pode ter seus métodos **start** e **stop** acionados inúmeras vezes, conforme a sua exibição no sistema. Geralmente são utilizados para parar animações ou outras rotinas que consumam recursos e só devam ser executadas durante a exibição da *applet*.

O método **stop** também é chamado imediatamente antes do método **destroy**, descrito a seguir.

8.1.5 O Método *destroy*

Este método é acionado quando a *applet* está sendo descarregada da página para que seja realizada a liberação final de todos os recursos utilizados durante sua execução. É acionado pelo *browser* quando da troca de páginas. Raramente necessita ser utilizado de forma explícita quando da utilização de conexões em rede ou comunicação com outras *applets*.

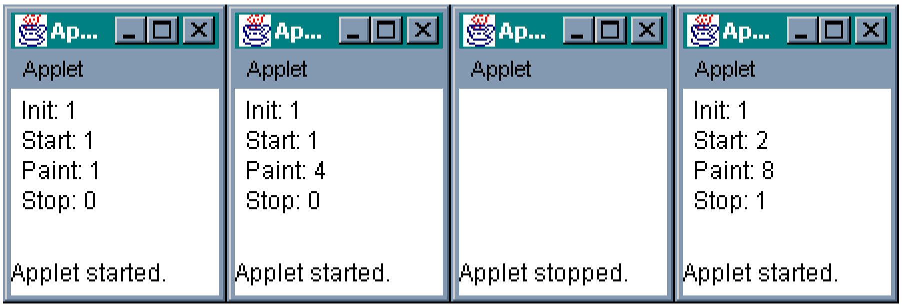
Segue um outro exemplo simples de *applet* que demonstra a ocorrência destes métodos especiais. No exemplo são declaradas e inicializadas quatro variáveis inteiras que serão utilizadas como contadores do acionamento dos métodos **init**, **start**, **paint** e **stop** (o método **destroy** não necessita ser contabilizado pois só ocorre uma única vez). Em cada um dos métodos coloca-se a respectiva variável de controle sendo modificada por uma operação de incremento, ou seja, cada vez que tal método é acionado a variável de controle "conta" quantas vezes isto ocorreu. O método **paint** é o único com código extra pois nele se exibe o valor dos contadores.

```
// AppletMethods.java

import java.applet.Applet;
import java.awt.Graphics;

public class AppletMethods extends Applet {
  private int inits = 0;
  private int starts = 0;
  private int paints = 0;
  private int stops = 0;

  public void init() {
    inits++;
  }
  public void start() {
    starts++;
  }
  public void paint( Graphics g){
    paints++;
    g.drawString("Init: "+inits, 5, 15);
    g.drawString("Start: "+starts, 5, 30);
    g.drawString("Paint: "+paints, 5, 45);
    g.drawString("Stop: "+stops, 5, 60);
  }
```

```
  public void stop() {
    stops++;
  }
}
```

Exemplo 84 Classe AppletMethods

Executando a aplicação através do appletviewer obtemos o seguinte resultado em diferentes situações:



```
Init: 1
Start: 1
Paint: 1
Stop: 0

Applet started.
```

```
Init: 1
Start: 1
Paint: 4
Stop: 0

Applet started.
```

```
Applet stopped.
```

```
Init: 1
Start: 2
Paint: 8
Stop: 1

Applet started.
```

Figura 65 *Applet* AppletMethods

Usando as opções do menu "Applet" do appletviewer é possível comandar operações de: *start*, *stop*, *restart*, *reload*, *clone* (duplicação da *applet*) e *save* (serialização da *applet*) permitindo testar adequadamente as *applets* em execução.

8.2 A Classe java.applet.Applet

Como colocado, uma *applet* é tipicamente um pequeno programa que é executado de forma embutida (*embedded*) em outra aplicação, usualmente o navegadores para Web. Portanto as *applets* não são programas independentes.

Pode-se notar nos exemplos dados que o pacote **java.applet** é sempre importado. Isto ocorre porque a classe **java.applet.Applet** deve sempre ser a superclasse de qualquer aplicação que se pretenda embutir numa página Web ou executar através do appletviewer. A classe **Applet**, única existente neste pacote, provê um interface padronizada entre as *applets* e o ambiente em que são executadas.

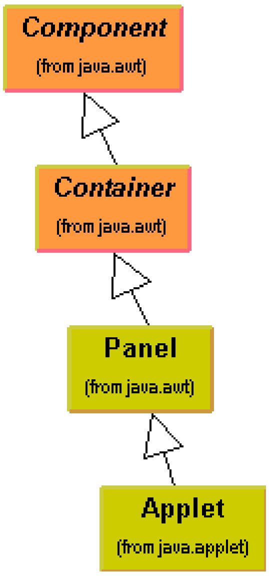


Figura 66 Hierarquia da Classe java.applet.Applet

Embora a classe **java.applet.Applet** não faça parte do pacote AWT, as *applets* são componentes da AWT pois esta classe é uma extensão da classe **java.awt.Panel**, como pode ser visto na hierarquia de classes ilustrada na Figura 66.

Sendo um componente derivado diretamente de **java.awt.Panel** seu tratamento em termos de adição de componentes, *layout* e controle de eventos é idêntico ao de aplicações comuns construídas através da utilização da AWT, portanto é relativamente simples construir *applets* com a aparência de aplicações destinadas a GUI do sistema.

As *applets* compartilham das operações disponíveis nas classes **java.awt.Panel** (Tabela 26), **java.awt.Container** (Tabela 27) e **java.awt.Component** (Tabela 18 e Tabela 19).

Temos na Tabela 55 os principais métodos da classe **Applet**.

Método	Descrição
destroy()	Método acionado pelo *browser* ou appletviewer para indicar que a *applet* deve liberar todos os recursos usados.
getAppletContext()	Retorna o contexto de execução da *applet* permitindo interação com o *browser*.
getAppletInfo()	Retorna informação associada a *applet*.
getCodeBase()	Retorna a URL de origem da *applet*.
getDocumentBase()	Retorna o documento de origem da *applet*.
getImage(URL)	Obtêm uma imagem na URL especificada.
getParameter(String)	Obtêm o valor do parâmetro especificado.
getParameterInfo()	Retorna informação associada aos parâmetros da *applet*.
init()	Método acionado pelo *browser* ou appletviewer para indicar que a *applet* foi carregada.
resize(int, int)	Redimensiona a *applet* para os valores dados de largura e altura.
showStatus(String)	Exibe a mensagem dada para a janela ou linha de *status*.
start()	Método acionado pelo *browser* ou appletviewer para indicar que a *applet* deve iniciar sua execução.
stop()	Método acionado pelo *browser* ou appletviewer para indicar que a *applet* deve parar sua execução.

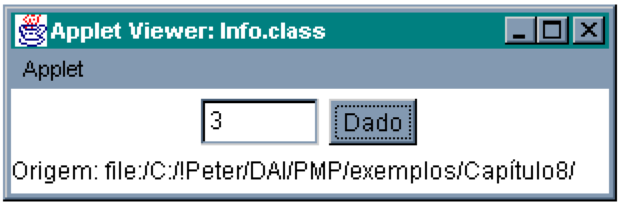
Tabela 55 Métodos da Classe java.applet.Applet

Segue um exemplo de uma *applet* que sorteia um número de 1 a 6 quando acionado o botão “Ok”. Na linha de *status* do appletviewer ou browser é exibida a origem do documento cada vez que o botão é acionado.

```
// Info.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Info extends Applet
                          implements ActionListener {
  private TextField tfSaida;
  private Button bOk;

  public void init() {
    // instanciacao e adicao de componentes e listener
    add(tfSaida = new TextField(5));
    tfSaida.setEditable(false);
    add(bOk = new Button("Dado"));
    bOk.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    showStatus("Origem: " + getCodeBase().toString());
    tfSaida.setText("" + (int)(1 + Math.random()*6));
  }
}
```

Exemplo 85 Classe Info

Na Figura 67 temos o resultado desta *applet* simples sendo executada através do appletviewer.



Origem: file:/C:/!Peter/DAI/PMP/exemplos/Capítulo8/

Figura 67 *Applet* Info

Segue outra *applet*, mais sofisticada, que utiliza diversos componentes da AWT e ainda usa sua área central para efetuar o desenho de uma *string* fornecida pelo usuário utilizando os fontes padrão do Java e efeitos especiais que podem ser facilmente obtidos através de alguns truques. No método **init** temos a adição dos componentes num painel superior e outro lateral, organizados diferentemente. São adicionados uma caixa para entrada do texto a ser exibido e quatro caixas de seleção que permitirão a escolha do fonte, seu tamanho e estilo além da aplicação de efeitos especiais implementados diretamente no código.

Ainda no método **init** são populados os vários componentes caixa de seleção (*combos*) destacando-se a forma com que obtemos os fontes disponíveis para o Java através do método **getFontList** do objeto **Toolkit**:

```
chFontes = new Choice();
  ⋮
// preparando combo fontes
String fonts[] = getToolkit().getFontList();
for (int i=0; i<fonts.length; i++)
  chFontes.add(fonts[i]);
```

No método **paint**, conforme as opções correntes, é desenhado o texto fornecido com as opções especificadas nas caixas de seleção. Sem exceção, os truques usados para obtenção dos efeitos especiais são o desenho das sombras, entalhes e contornos com cores diferenciadas e em posições ligeiramente diferentes do texto fornecido o qual é sempre desenhado sempre por último.

A *applet* também implementa as interfaces **ActionListener** e **ItemListener** para que a entrada de um novo texto na caixa de texto ou modificações das opções sejam refletidas imediatamente na exibição. Segue abaixo o código integral desta *applet*:

```
/* FontDemo.java
<applet code="FontDemo.class" width=500 height=130>
</applet>
*/

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class FontDemo extends Applet
                      implements ActionListener,
                                 ItemListener {
  private TextField tfTexto;
  private Choice chFontes;
  private Choice chEstilos;
  private Choice chTamanhos;
  private Choice chEfeitos;

  public void init() {
    setLayout(new BorderLayout());
    setBackground(SystemColor.control);
    // painel superior
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
```

```
    p.setBackground(SystemColor.control);
    p.add("West", new Label("Texto"));
    p.add("Center", tfTexto = new TextField("FontDemo Applet"));
    tfTexto.addActionListener(this);
    add("North", p);
    // painel lateral
    p = new Panel();
    p.setBackground(SystemColor.control);
    Panel paux = new Panel();
    paux.setBackground(SystemColor.control);
    paux.setLayout(new GridLayout(4, 2, 3, 0));
    paux.add(new Label("Fonte"));
    paux.add(new Label("Estilo"));
    paux.add(chFontes = new Choice());
    chFontes.addItemListener(this);
    paux.add(chEstilos = new Choice());
    chEstilos.addItemListener(this);
    paux.add(new Label("Tamanho"));
    paux.add(new Label("Efeito"));
    paux.add(chTamanhos = new Choice());
    chTamanhos.addItemListener(this);
    paux.add(chEfeitos = new Choice());
    chEfeitos.addItemListener(this);
    p.add(paux);
    add("East", p);
    // preparando combo fontes
    String fonts[] = getToolkit().getFontList();
    for (int i=0; i<fonts.length; i++)
      chFontes.add(fonts[i]);
    // preparando combo estilos
    chEstilos.add("Normal");
    chEstilos.add("Itálico");
    chEstilos.add("Negrito");
    chEstilos.add("Negrito Itálico");
    // preparando combo tamanhos
    for (int i=8; i<42; i+=2)
      chTamanhos.add(i+"");
    chTamanhos.select(3);
    // preparando combo efeitos
    chEfeitos.add("Nenhum");
    chEfeitos.add("Sombra");
    chEfeitos.add("Gravado");
    chEfeitos.add("Contorno");
    chEfeitos.add("Transparente");
  }

  public void paint(Graphics g) {
    String texto = tfTexto.getText();
    if (texto.equals(""))
      return;
    int estilo = Font.PLAIN;
    int tamanho =Integer.parseInt(chTamanhos.getSelectedItem());
    int hpos = 10;
    int vpos = getSize().height + getInsets().top +
               tfTexto.getSize().height + tamanho/2;
    switch(chEstilos.getSelectedIndex()) {
      case 1:
        estilo = Font.ITALIC;
        break;
      case 2:
        estilo = Font.BOLD;
```

```
        break;
      case 3:
        estilo = Font.ITALIC + Font.BOLD;
        break;
    }
    g.setFont(new Font(chFontes.getSelectedItem(),
                       estilo, tamanho));
    switch(chEfeitos.getSelectedIndex()) {
      case 0: // Nenhum
        g.setColor(SystemColor.controlText);
        break;
      case 1: // Sombra
        g.setColor(SystemColor.controlShadow);
        g.drawString(texto, hpos+2, vpos+2);
        g.setColor(SystemColor.controlLtHighlight);
        break;
      case 2: // Gravado
        g.setColor(SystemColor.controlLtHighlight);
        g.drawString(texto, hpos+1, vpos+1);
        g.setColor(SystemColor.controlShadow);
        break;
      case 3: // Contorno
        g.setColor(SystemColor.controlLtHighlight);
        g.drawString(texto, hpos+1, vpos+1);
        g.drawString(texto, hpos+1, vpos-1);
        g.drawString(texto, hpos-1, vpos+1);
        g.drawString(texto, hpos-1, vpos-1);
        g.setColor(SystemColor.controlText);
        break;
      case 4: // Transparente
        g.setColor(SystemColor.controlText);
        g.drawString(texto, hpos+1, vpos+1);
        g.drawString(texto, hpos+1, vpos-1);
        g.drawString(texto, hpos-1, vpos+1);
        g.drawString(texto, hpos-1, vpos-1);
        g.setColor(SystemColor.control);
        break;
    }
    g.drawString(texto, hpos, vpos);
  }

  public void actionPerformed(ActionEvent e) {
    repaint();
  }

  public void itemStateChanged(ItemEvent e) {
    repaint();
  }
}
```

Exemplo 86 Classe FontDemo

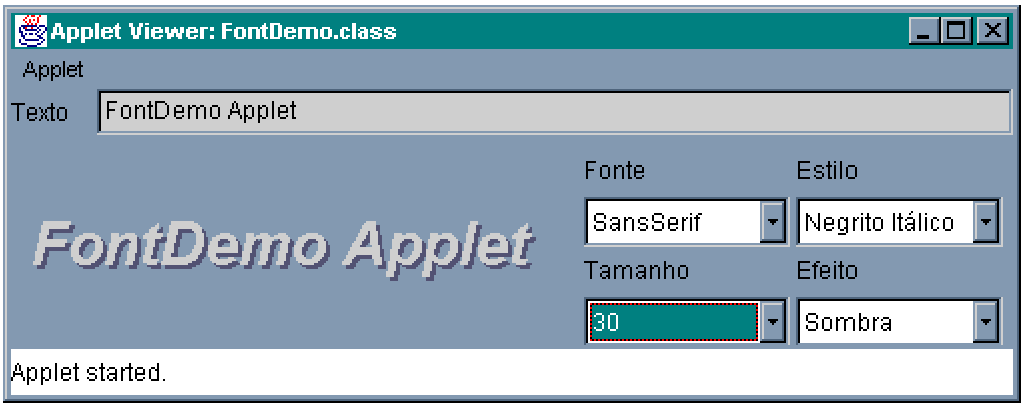
Após a compilação pode ser utilizado o appletviewer para testarmos a *applet*. Para evitarmos a monótona criação de arquivos HTML contendo apenas a *tag* <APPLET> para teste da *applet* via appletviewer ou browsers podemos adicionar o seguinte comentário no início do programa fonte:

```
/* FontDemo.java
<applet code="FontDemo.class" width=500 height=130>
</applet>
*/
```

O comentário não influi na compilação mas é entendido pelo appletviewer, permitindo usar o próprio fonte como argumento desta ferramenta:

```
appletviewer FontDemo.java
```

O resultado desta *applet* é ilustrado na .



Texto: FontDemo Applet
Fonte: SansSerif
Estilo: Negrito Itálico
Tamanho: 30
Efeito: Sombra
Applet started.

Figura 68 *Applet* FontDemo

Estes efeitos especiais de exibição de texto podem ser facilmente empregados em outras *applets*, não apenas para exibição de texto mas para exibição de outros objetos. Aplicando a mesma estratégia utilizada podemos criar outros efeitos como exibição em 3D, segmentação, movimento etc.

8.3 Restrições das Applets

As *applets* foram planejadas para serem carregadas de um *site* remoto e então executadas localmente. Desta forma, para evitar que *applets* estranhas prejudiquem o sistema local onde estão sendo executadas, isto é, alterem, apaguem ou acessem informações armazenadas neste sistema, foram impostas restrições bastante severas ao funcionamento das *applets*.

As *applets* são executadas pelos browser e monitoradas por um gerenciador de segurança (denominado de *applet security manager*) que lança uma interrupção do tipo **SecurityException** caso a *applet* viole alguma das regras de segurança impostas. Assim sendo é comum dizer que as *applets* operam dentro de uma caixa de areia, tal qual crianças brincando sob vigilância de um adulto (*the sandbox model*).

As *applets* podem realizar as seguintes tarefas: (i) podem exibir imagens, (ii) podem executar sons, (iii) podem processar o acionamento do teclado e *mouse* e (iv) podem se comunicar com o *host* de onde foram carregadas. Embora pareça excessivamente restritivo, este modelo admite funcionalidade suficiente para que uma *applet* exiba conteúdos diversos, auxilie ou dirija a navegação, realize operações de consistência de dados e envie ou receba informações necessárias para realização de uma consulta a um banco de dados, um pedido de compra ou cadastro de informações.

Por outro lado as *applets* estão sujeitas as seguintes restrições: (i) não podem executar programas localmente instalados, (ii) não podem se comunicar com outros *hosts* exceto o de origem (*originating host*), (iii) não podem ler e escrever no sistema de arquivos local e (iv) não podem obter informações do sistema onde operam exceto sobre a JVM sobre a qual operam.

Vale observar também que a capacidade de criação de janelas comuns e janelas de diálogo por parte das *applets* é limitada, pois o sistema automaticamente ajusta seus títulos para "*Untrusted Applet*" como forma de avisar os usuários de que tais janelas fazem parte de uma *applet* e não de um programa comum.

Todas estas restrições só são possíveis devido ao fato do código Java ser interpretado pela JVM que pode verificar se tais regras estão sendo de fato obedecidas. Na Tabela 56 temos relacionadas as restrições do código Java quando executado através do interpretador de aplicações, do appletviewer e de *browsers*.

Capacidades do Código Java	**Aplicação e java/jre**	***Applet* e appletviewer**	***Applet* (local) e browser**	***Applet* (remota) e browser**
Leitura de arquivo local	✓	✓		
Escrita de arquivo local	✓	✓		
Obtenção de informação sobre arquivo	✓	✓		
Eliminação de arquivo	✓			
Execução de outro programa	✓	✓		
Acesso a propriedade **user.name**	✓	✓	✓	
Conexão a porta do servidor de origem	✓	✓	✓	✓
Conexão a porta de outro servidor	✓	✓	✓	
Carregamento da biblioteca Java	✓	✓	✓	
Criação de janelas	✓	✓	✓	✓
Acionamento da rotina **exit**	✓	✓		

Tabela 56 Restrições de Funcionamento de Código Java

É bastante nítido o aumento de restrições de *applets* executadas através *download* quando comparadas com a execução de programas comuns escritos em Java. Em algumas situações tal nível de restrição se mostra demasiadamente severo para uma *applet*, assim é possível “assinar” o código, isto é, anexar um certificado de segurança que ateste a origem da *applet* tornando-a uma *signed applet* e permitindo sua execução num nível de restrição semelhante ao de programas comuns. Apesar de bem planejado, o esquema de segurança empregado para as *applet* Java não é perfeito e contêm alguns problemas que no momento estão sendo solucionados pela própria *Sun*.

8.4 Parametrizando Applets

Programas comuns podem receber argumentos oriundos da linha de comando, permitindo assim a modificação ou ajuste do programa através de tais parâmetros. As *applet* podem ser igualmente parametrizadas aumentando a versatilidade de sua utilização e permitindo a criação de código mais genérico. Como sua execução não é iniciada através de uma linha de comando mas sim através de um arquivo HTML lido pelo *browser* ou do appletviewer, a passagem de parâmetros pode se realizar com o uso de uma *tag* especial destinadas a esta finalidade, utilizada em conjunto com a *tag* <APPLET>. Esta *tag* é <PARAM> que possui vários campos dos quais dois são aplicáveis a utilização com a *tag* <APPLET> conforme descrito a seguir:

Campo	**Descrição**
NAME	*String* que indica o nome do parâmetro.
VALUE	*String* que indica o valor do parâmetro denominado por NAME.

Tabela 57 Campos da *Tag* <PARAM>

Desta forma com esta *tag* podemos especificar para a *applet* que um certo parâmetro, cujo nome está definido no campo NAME e cujo valor está definido no campo

VALUE, está disponível para utilização. Pode ser fornecidos tantos parâmetros quanto necessários para uma *applet*, isto é, uma ou mais *tags* <PARAM>, desde que incluídas entre as *tags* <APPLET> e </APPLET>, de forma que possam ser lidas pela *applet* identificada como exemplificado:

```
<applet code="Parametros.class" height=100 width=100>
<param name="corFundo" value="7C7C7C">
<param name="valorInicial" value="6">
</applet>
```

Para que as *applets* recebam os valores passados através da *tag* <PARAM> é necessário o uso do método **getParameter**. Este método permite recuperar o valor do parâmetro cujo nome é especificado como argumento, por exemplo:

```
// processamento dos parametros
    String aux = getParameter("corFundo");
    if (aux!=null) {
      setBackground(new Color(Integer.parseInt(aux, 16)));
    }
    aux = getParameter("valorInicial");
    if (aux!=null) {
      tfEntrada.setText(aux);
    }
```

Se o nome especificado como parâmetro não for encontrado ou não possuir valor definido, o método **getParameter** retorna o valor null, o qual pode ser utilizado para verificar se o valor do parâmetro desejado foi corretamente obtido. É importante notar que todos os valores recuperados são retornados como objetos tipo **String**, exigindo a conversão para o tipo desejado, no caso o valor **corFundo** foi convertido de **String** para um inteiro em hexadecimal (base 16) e o **valorInicial** utilizado diretamente como **String**.

Segue um exemplo de uma *applet* simples cujo único propósito é ler e utilizar os valores dos dois parâmetros oriundos da *tag* <PARAM> mostrados acima.

```
// Parametros.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class Parametros extends Applet
                        implements ActionListener {
  private TextField tfEntrada;
  private TextField tfSaida;
  private Button bOk;

  public void init() {
    // instanciacao e adicao de componentes e listener
    add(tfEntrada = new TextField(15));
    add(tfSaida = new TextField(15));
    tfSaida.setEditable(false);
    add(bOk = new Button("Ok"));
    bOk.addActionListener(this);
    // processamento dos parametros
    String aux = getParameter("corFundo");
    if (aux!=null) {
      setBackground(new Color(Integer.parseInt(aux, 16)));
    }
    aux = getParameter("valorInicial");
    if (aux!=null) {
      tfEntrada.setText(aux);
    }
  }

```

```
  public void actionPerformed(ActionEvent e) {
    int fatorial = 1;
    int valor = Integer.parseInt(tfEntrada.getText());
    for (int i=valor; i>0; i--)
      fatorial *= i;
    tfSaida.setText(valor + "! = " + fatorial);
  }
}
```

Exemplo 87 Classe Parametros

Executando-se esta *applet* temos que quando é acionado o botão “Ok”, o fatorial do inteiro fornecido na primeira caixa de texto é exibido na segunda caixa de entrada. A cor de fundo da *applet* e o valor inteiro inicial são obtidos do arquivo HTML. Teste esta *applet* com usando um arquivos HTML sem parâmetros e depois com outro contendo os parâmetros exemplificados anteriormente.

Testando a *applet* através do appletviewer dispõe-se de outras informações sobre a *applet*, seu ambiente atual e uso. Através da opção Tag pode-se verificar como estão sendo interpretadas as *tag* <APPLET> e <PARAM>. Também é possível documentar, através da própria *applet*, quais são os parâmetros aceitos e qual o seu uso bem como o autor e o propósito da *applet*. Tais informações podem ser codificadas nos métodos **getAppletInfo** e **getParameterInfo**, cuja implementação é exibida no XXX. Estes métodos são acionados através de sua opção Info do appletviewer, exibindo o conteúdo associado. A própria *applet* pode exibir tal resultado numa janela de diálogo separada quando acionado algum botão ou fornecido um parâmetro especial.

8.5 Applets que são Aplicações

É possível escrevermos programas Java que se comportem como aplicativos comuns ou como *applets* dependendo de como sejam carregados. Constatamos que isto é possível observando que as *applets* fazem parte da hierarquia de componentes da AWT (vide Figura 66) sendo um tipo especial de **Panel**, então a adição da *applet* a um **Frame** equivale a adição de um painel comum.

Portanto para que uma *applet* possa ser executada como uma aplicação comum basta adicionarmos um método **main** a classe que constitui esta *applet* realizando neste método o tratamento adequando para criação de uma aplicação:

(i) Adiciona-se um método **main** a classe da *applet* idêntico ao de aplicações comuns.

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

(ii) Cria-se uma instância de um **Frame** especificando também o título da janela.

```
Frame frame = new Frame("Título da Janela");
```

(iii) Cria-se uma instância da *applet*, não deixando de executar os métodos **init** e **start** para sua adequada inicialização, tal como faria um *browser*.

```
ClasseDaApplet applet = new ClasseDaApplet();
applet.init();
applet.start();
```

(iv) Adiciona-se a *applet* a região central do **Frame**.

```
frame.add("Center", applet);
```

(v) Ajusta-se o tamanho do **Frame**, adiciona-se um **WindowListener** e exibe-se o **Frame**.

```
frame.setSize(largura, altura);
frame.addWindowListener(new CloseWindowAndExit());
frame.show();
```

Outra alternativa seria a criação de uma classe a parte efetuando-se o mesmo processo mas não teríamos assim o comportamento duplo *applet*/aplicação de uma mesma classe. Considerando a *applet* **FontDemo** do Exemplo 86, poderíamos adicionar o seguinte método **main** a mesma, adicionando a esta *applet* esta característica dupla de comportamento:

```
public static void main(String args[]) {
    Frame f = new Frame("FontDemo");
    FontDemo fd = new FontDemo();
    fd.init();
    fd.start();
    f.add("Center", fd);
    f.setSize(500, 160);
    f.addWindowListener(new CloseWindowAndExit());
    f.show();
  }
```

Desta forma poderíamos executar o programa de qualquer uma das formas abaixo:

```
java FontDemo
```

```
appletviewer FontDemo.java
```

A adaptação de uma *applet* para que possa ser executada como um programa é útil em situações onde a *applet* deva ser executada tanto localmente quanto através de uma rede (uma *intranet* ou da própria Internet). No entanto, a adaptação no sentido inverso é tecnicamente possível, embora os programas transformados em *applet* devam se submeter as restrições de funcionamento destas, tal como listado na Tabela 56.

8.6 O Contexto das Applets

Uma das características mais importantes de uma *applet* é a possibilidade de interação desta com o navegador onde está sendo executada. As *applets* podem obter informações sobre o ambiente em que estão sendo executadas e atuar neste ambiente através da interface **java.applet.AppletContext**., cujos principais métodos estão relacionados na Tabela 58. Como a classe **java.applet.Applet** implementa esta interface, então toda *applet* tem disponíveis tais métodos.

Método	Descrição
getApplet(String)	Encontra a *applet* identificada pelo campo **name** da *tag* <APPLET> dada no documento atual retornando uma referência para a mesma.
getApplets()	Retorna uma lista das *applets* contidas no documento em exibição.
getAudioClip(URL)	Retorna um *clip* de áudio da URL absoluta dada.
getImage(URL)	Retorna uma imagem da URL absoluta dada.
showDocument(URL)	Requisita ao browser que substitua a página atual pela contida na URL dada.
showDocument(URL, String)	Requisita ao browser que substitua a página atual pela contida na URL dada na janela especificada como alvo.
showStatus(String)	Exibe a *string* fornecida na janela ou linha de *status*.

Tabela 58 Métodos da Interface java.applet.AppletContext

Notamos que esta interface oferece meios para as *applets* descubram quais as outras *applets* existentes numa mesma página (chamada de documento) obtendo até mesmo uma referência para uma *applet* específica. Além disso é possível através destes métodos obter imagens ou *clips* de áudio existentes em outras URLs (*Uniform Resource Locator*).

Dos métodos relacionados um é particularmente importante: **showDocument**. Através deste método uma *applet* pode comandar um navegador para exibir um novo documento em uma de suas janelas ou mesmo criar novas janelas. Esta capacidade das *applets* abre um novo e grande horizonte para sua utilização.

Uma *applet* simples que demostra esta capacidade é dada a seguir. Nela temos apenas uma caixa de entrada onde pode ser digitada uma URL qualquer e uma caixa de seleção onde pode-se escolher onde o browser deve exibir o documento solicitado. O método **init** contêm apenas a instanciação e adição dos componentes a interface da *applet*. É no método **actionPerfomed** que solicitamos a exibição de um novo documento quando o usuário pressiona a tecla “Enter” dentro da caixa de entrada. Analise o código abaixo:

```
public void actionPerformed(ActionEvent e) {
  // obtem o contexto da applet (janela do browser)
  AppletContext context = getAppletContext();
  try {
    URL u = new URL(tfURL.getText()); // cria objeto URL
    // aciona o browser
    context.showDocument(u, chModo.getSelectedItem());
    showStatus(tfURL.getText());
  } catch (MalformedURLException exp) {
    showStatus("URL inválida"); // URL inválida
  }
}
```

Nele obtêm-se o contexto da *applet* para que seja possível comandar o browser. Com o texto fornecido como endereço de um determinado *site* da Internet cria-se um objeto URL (o que exige a utilização do pacote **java.net**), caso não seja possível ocorre um exceção sinalizando o fato. Dispondo do contexto e da URL aciona-se o método **showDocument** indicando a forma de exibição da URL dada. A *applet* em execução num navegador comum é ilustrada na Figura 69.

http://www.uol.com.br

name

Figura 69 *Applet* ShowURL

As opções exibidas pela *applet* correspondem a seguintes situações:

Valor	Descrição
_self	Solicita a exibição na janela e *frame* que contêm a *applet*.
_parent	Solicita a exibição no *frame* pai da janela que contêm a *applet*. Caso não exista age como _self.
_top	Solicita a exibição no *frame* superior. Caso contrário age como _self.
_blank	Solicita a exibição numa nova janela.
name	Solicita a exibição na janela denominada. Se não existir cria tal janela.

Tabela 59 Opções para Argumento Target do Método showDocument

Através do uso destas opções a *applet* passa a controlar a exibição do novo documento em *frames* HTML de uma página ou janelas distintas do *browser*.

O código completo desta *applet* é dado a seguir.

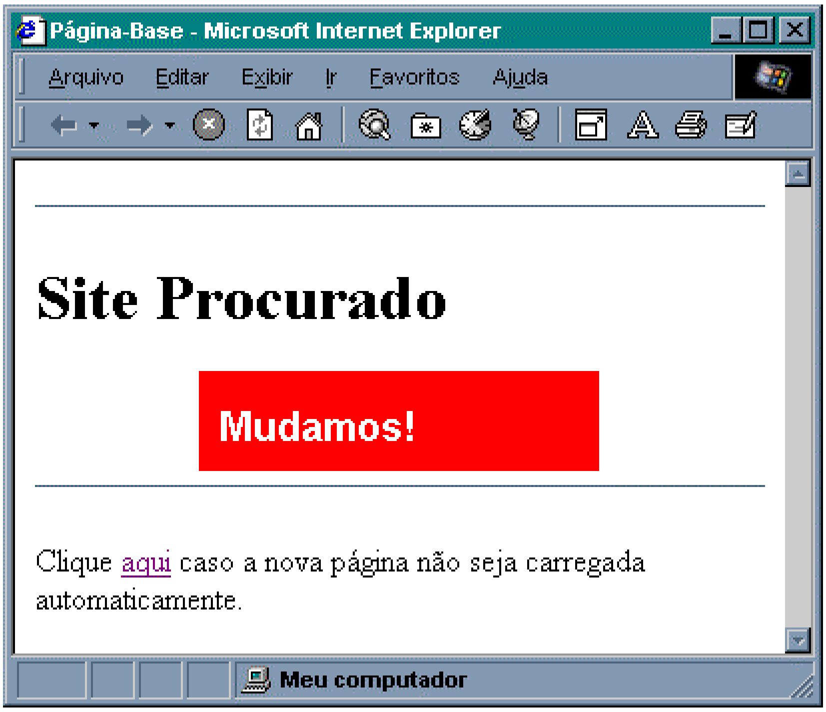
```
/* ShowURL.java
<applet code="ShowURL.class" width=300 height=40>
</applet>
*/

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class ShowURL extends Applet
                     implements ActionListener {
  private TextField tfURL;
  private Choice chModo;

  public void init() {
    tfURL = new TextField(30);
    tfURL.addActionListener(this);
    chModo = new Choice();
    chModo.add("_self");
    chModo.add("_parent");
    chModo.add("_top");
    chModo.add("_blank");
    chModo.add("name");
    add(tfURL);
    add(chModo);
  }
  public void actionPerformed(ActionEvent e) {
    // obtem o contexto da applet (janela do browser)
    AppletContext context = getAppletContext();
    try {
      // cria um objeto URL
      URL u = new URL(tfURL.getText());
      // aciona o browser
      context.showDocument(u, chModo.getSelectedItem());
      showStatus(tfURL.getText());
    } catch (MalformedURLException exp) {
      // Se URL inválida exibe mensagem
      showStatus("URL inválida");
    }
  }
}
```

Exemplo 88 Classe ShowURL

A *applet* apresentada demonstra a capacidade de um programa Java em comandar a navegação através da Internet.

8.6.1 Mudança Automática de Páginas com uma *Applet*

Consideremos agora que um determinado *site* tenha sido mudado de servidor. Ao invés de uma mensagem de aviso e um *link* na antiga *home page* seria conveniente adicionar uma *applet* capaz de transferir automaticamente o usuário para a nova página, agilizando a navegação. Desta forma uma página como a ilustrada na Figura 70 poderia ser exibida por alguns instantes, isto é, até a *applet* seja carregada e iniciada.



Site Procurado

Mudamos!

Clique aqui caso a nova página não seja carregada automaticamente.

Figura 70 *Applet* Go

A *applet* proposta utiliza o mesmo princípio para o comando do *browser*, exceto que tal código é colocado no método **start**, garantindo sua execução automática após o carregamento da *applet* por parte do browser. Através de parâmetros podemos determinar a mensagem a ser exibida, a cor do texto e a cor do fundo. No método **init** são obtidos os quatro parâmetros utilizados pela *applet*: a URL de destino (parâmetro obrigatório), a cor de fundo da área da *applet*, a cor do texto e a mensagem a ser exibida. O método **paint** é usado para exibir a mensagem especificada nos parâmetros da *applet*. O código sugerido para esta *applet* é dado no Exemplo 89.

```
/* Go.java
<applet code="Go.class" width=200 height=50>
<param name="urlAlvo" value="/Alvo.html">
<param name="corFundo" value="FF0000">
<param name="corTexto" value="FFFFFF">
<param name="mensagem" value="Mudamos!">
</applet>
*/

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class Go extends Applet {
  private String urlAlvo;
  private Color corFundo;
  private Color corTexto;
  private String mensagem;

  public void init() {
    // obtem URL dos parametros da applet
    urlAlvo = getParameter("urlAlvo");
    System.out.println("Alvo: " + urlAlvo);
    // obtem cores da applet dos parametros
    corFundo = Color.white;
```

```
    String c = getParameter("corFundo");
    if (c!=null) {
      corFundo = new Color(Integer.parseInt(c, 16));
    }
    setBackground(corFundo);
    corTexto = Color.black;
    c = getParameter("corTexto");
    if (c!=null) {
      corTexto = new Color(Integer.parseInt(c, 16));
    }
    // obtem mensagem da applet
    mensagem = getParameter("mensagem");
  }

  public void start() {
    // obtem o contexto da applet (janela do browser)
    AppletContext context = getAppletContext();
    try {
      // cria um objeto URL
      URL u = new URL(urlAlvo);
      // aciona o browser
      context.showDocument(u);
      showStatus(urlAlvo);
    } catch (MalformedURLException e) {
      // Se URL inválida exibe mensagem
      showStatus("URL inválida");
    }
  }

  public void paint(Graphics g) {
    // Ajusta fonte e cor
    g.setFont(new Font("SansSerif", Font.BOLD, 20));
    g.setColor(corTexto);
    g.drawString(mensagem, 10, 35);
  }
}
```

Exemplo 89 Classe Go

8.6.2 Uma *Applet* Menu

Construiremos agora uma *applet* capaz de exibir um menu de *sites* construído graficamente tendo sua configuração feita de parametrização. A configuração pretendida para esta *applet* é: (i) definição de uma lista de URL com os respectivos nomes de seus *sites*, (ii) definição do tamanho da *applet* e (iii) definição da cor do texto, cor de fundo e cor da decoração utilizada. Para isto será necessário declarar os seguintes atributos na classe desta *applet*:

```
private String urls[];
private String urlNames[];
private int width;
private int height;
private Color backColor;
private Color linesColor;
private Color textColor;
```

No método **init** são lidos os valores fornecidos para tais parâmetros. Observe que um parâmetro extra, **urlNumber** é utilizado para indicar a quantidade de *urls* pretendidas, facilitando seu processamento. Com exceção das *urls* e seus nomes, existem valores *default* para os demais parâmetros.

No método **paint** ocorre a renderização da *applet*. Nele são desenhadas as *string* correspondentes aos nomes. Para melhorar a aparência final são desenhadas separadores entre os nomes dos sites. Tais separadores, cujo propósito é simplesmente decorativo, são compostos de uma linha e de quadrados (a esquerda antes do nome do site e a direita de pois deste) como pode ser visto na Figura 71.

Universo OnLine
Universidade São Francisco
Núcleo de Educação a Distância
Unicamp
http://www.usf.com.br/distancia

Figura 71 *Applet* MenuApplet

Destacamos agora o código necessário para a método **paint**:

```
public void paint(Graphics g) {
  // Seleciona fonte
  Font f = new Font("SansSerif", Font.BOLD, 10);
  int i;
  // Desenha linhas, quadrados e strings
  for (i=0; i<urls.length ; i++) {
    g.setColor(linesColor);
    g.drawLine(      10, 20*i, width-12, 20*i);
    g.fillRect(       0, 20*i,       10,   10);
    g.fillRect(width-11, 11+20*i,    10,   10);
    g.setColor(textColor);
    g.drawString(urlNames[i], 15, 20*(1+i)-5);
  }
  g.setColor(linesColor);
  g.drawLine(10, 20*i, width-12, 20*i);
}
```

A *applet* necessita implementar as interfaces **MouseListener** para que os cliques do mouse sejam processados e **MouseMotionListener** para que o posicionamento do mouse seja acompanhado proporcionando a troca do cursor e a exibição da URL selecionada na linha de *status*.

O código completo desta *applet* é dado a seguir:

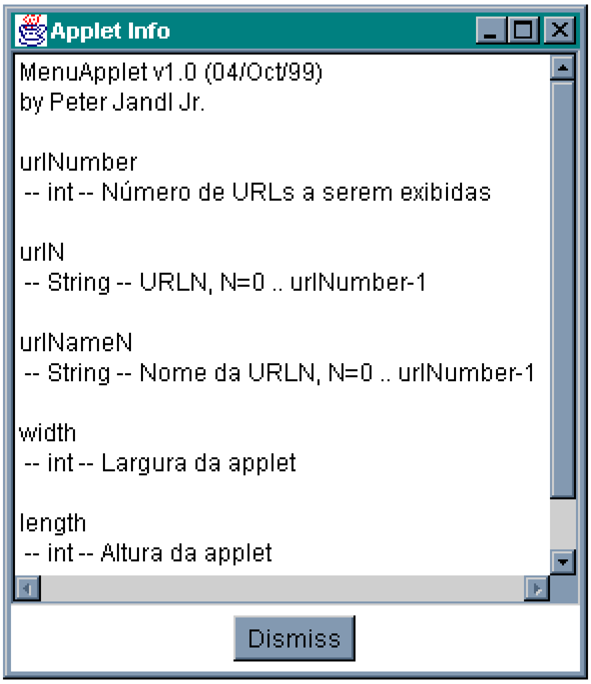
```
// MenuApplet.java

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.*;

public class MenuApplet extends Applet
             implements MouseListener, MouseMotionListener {
  private String urls[];
  private String urlNames[];
  private int width;
  private int height;
  private Color backColor;
  private Color linesColor;
  private Color textColor;

  public void init() {
    // Verifica número de URLs fornecidas
```

```
    String urlNumber = getParameter("urlNumber");
    System.out.println("urlNumber: " + urlNumber);
    if (urlNumber == null) {
      return;
    }
    // Obtêm as URLs dos parâmetros da applet
    int n = Integer.parseInt(urlNumber);
    urls = new String[n];
    urlNames = new String[n];
    for (int i=0; i<n; i++) {
      urls[i] = getParameter("url"+i);
      urlNames[i] = getParameter("urlName"+i);
    }
    // Adiciona os listeners a applet
    addMouseListener(this);
    addMouseMotionListener(this);
    // Obtêm tamanho da applet dos parâmetros
    width = 180;
    height = 180;
    String w = getParameter("width");
    String h = getParameter("height");
    if (w!=null && h!=null) {
      width = Integer.parseInt(w);
      height = Integer.parseInt(h);
    }
    setSize(width, height);
    // Obtêm cores da applet dos parâmetros
    backColor = Color.white;
    String c = getParameter("backColor");
    if (c!=null) {
      backColor = new Color(Integer.parseInt(c, 16));
    }
    setBackground(backColor);
    linesColor = Color.red;
    c = getParameter("linesColor");
    if (c!=null) {
      linesColor = new Color(Integer.parseInt(c, 16));
    }
    textColor = Color.black;
    c = getParameter("textColor");
    if (c!=null) {
      textColor = new Color(Integer.parseInt(c, 16));
    }
  }

  public void paint(Graphics g) {
    // Seleciona fonte
    Font f = new Font("SansSerif", Font.BOLD, 10);
    int i;
    // Desenha linhas, quadrados e strings
    for (i=0; i<urls.length ; i++) {
      g.setColor(linesColor);
      g.drawLine(      10, 20*i, width-12, 20*i);
      g.fillRect(       0, 20*i,       10,   10);
      g.fillRect(width-11, 11+20*i,    10,   10);
      g.setColor(textColor);
      g.drawString(urlNames[i], 15, 20*(1+i)-5);
    }
    g.setColor(linesColor);
    g.drawLine(10, 20*i, width-12, 20*i);
  }
```

```
  public void mouseClicked (MouseEvent e) {
    // Obtêm o contexto da Applet (janela do browser)
    AppletContext context = getAppletContext();
    int i = e.getY()/20;
    // Testa o posicionamento do mouse
    if (i>=0 && i<urls.length) {
      try {
        // Se posição válida cria um objeto URL
        URL u = new URL(urls[i]);
        // Aciona o browser
        context.showDocument(u);
        showStatus(urls[i]);
      } catch (MalformedURLException exp) {
        // Se URL inválida exibe mensagem
        showStatus("URL inválida");
      }
    } else
      // Se posição inválida limpa barra de status do browser
      showStatus("");
  }
  public void mousePressed (MouseEvent e) {}
  public void mouseReleased (MouseEvent e) {}
  public void mouseEntered (MouseEvent e) {}
  public void mouseExited (MouseEvent e) {}

  public void mouseMoved (MouseEvent e) {
    int i = e.getY()/20;
    // Testa o posicionamento do mouse
    if (i>=0 && i<urls.length) {
      // Se posição válida modifica o cursor e
      // exibe a URL na barra de status do browser
      setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
      showStatus(urls[i]);
    } else {
      // Se posição inválida exibe o cursor default
      // e limpa barra de status do browser
      setCursor(Cursor.getDefaultCursor());
      showStatus("");
    }
  }
  public void mouseDragged (MouseEvent e) {}

  public String getAppletInfo() {
    return "MenuApplet v1.0 (04/Oct/99)\nby Peter Jandl Jr.";
  }

  public String[][] getParameterInfo() {
    String[][] info = {
      {"urlNumber\n", "int", "No. de URLs a serem exibidas\n"},
      {"urlN\n",      "String", "URLN, N=0 .. urlNumber-1\n"},
      {"urlNameN\n",  "String", "Nome da URLN, N=0 .. urlNumber-
1\n"},
      {"width\n",        "int", "Largura da applet\n"},
      {"length\n",       "int", "Altura da applet\n"},
    };
    return info;
  }
}
```

Exemplo 90 Classe MenuApplet

Observe na parte final do código a implementação dos métodos **getAppletInfo** e **getParameterInfo** que são usados para obtenção de informação sobre a *applet*. O appletviewer exibe tais informações através da opção “Info” mostrando a janela abaixo:



```
MenuApplet v1.0 (04/Oct/99)
by Peter Jandl Jr.

urlNumber
 -- int -- Número de URLs a serem exibidas

urlN
 -- String -- URLN, N=0 .. urlNumber-1

urlNameN
 -- String -- Nome da URLN, N=0 .. urlNumber-1

width
 -- int -- Largura da applet

length
 -- int -- Altura da applet
```

Figura 72 Exibição de Informações de uma *Applet* (appletviewer)

Esta *applet* pode ser incorporada a uma página HTML, exibindo um menu URLs definido pelo usuário, como definido pelo trecho HTML dado a seguir que corresponde ao resultado ilustrado na Figura 71.

```
<applet code="MenuApplet.class" width="300" height="200">
  <param name="backColor"  value="FFFFFF">
  <param name="height" value="120">
  <param name="linesColor" value="800080">
  <param name="textColor"  value="000000">
  <param name="url0" value="http://uol.com.br">
  <param name="url1" value="http://www.usf.com.br">
  <param name="url2" value="http://www.usf.com.br/distancia">
  <param name="url3" value="http://www.unicamp.br">
  <param name="urlName0" value="Universo OnLine">
  <param name="urlName1" value="Universidade São Francisco">
  <param name="urlName2" value="Núcleo de Educação a Distância">
  <param name="urlName3" value="Unicamp">
  <param name="urlNumber" value="4">
  <param name="width" value="250">
</applet>
```

Exemplo 91 Parametrição da *Applet* MenuApplet

8.7 Empacotando Applets

Como visto, as *applets* são pequenos programas incorporados a páginas da Web, destinados a serem carregados pelos *browsers* através da Internet. Dado este cenário, sabemos que quanto menor a *applet* mais rapidamente esta será transmitida pela rede, minizando o tempo de espera o usuário. Outro aspecto importante é que uma *applet*, assim como qualquer aplicação Java, pode ser composta de várias classes, exigindo uma conexão com o servidor para a transmissão cada classe existente antes que *applet* possa ser corretamente iniciada e exibida.

Para otimizar o *download* de *applets*, reduzindo o seu tamanho e eliminando a necessidade de conexões adicionais para obtenção das demais classes exigidas, pode se utilizar os arquivos JAR (*Java Archive*).

Um arquivo JAR é o resultado produzido por uma ferramenta de mesmo nome, capaz de compactar vários arquivos diferentes em um único arquivo no formato ZIP. Sendo assim é possível compactar-se várias classes e outros recursos Java num único arquivo JAR, mantendo-se inclusive sua estrutura de diretórios. O arquivo JAR pode ser transmitido de uma só vez pela Internet, obtendo-se uma significativa melhora de performance na inicialização das *applets*. A ferramenta JAR que acompanha o *Sun* JDK, semelhante a ferramenta TAR de sistemas Unix, possui as seguintes opções:

Opção	Descrição
c	Cria um novo arquivo na saída padrão.
t	Lista o conteúdo do arquivo na saída padrão.
f	Especifica o nome do arquivo a ser criado, listado ou de onde arquivos serão extraídos.
x	Extrai os arquivos especificados do arquivo JAR.
O	Indica que não os arquivos não devem ser comprimidos.
m	Indica que um arquivo de manifesto externo deve ser incluído.
M	Indica que não deve ser criado um arquivo de manifesto.

Tabela 60 Opções da Ferramenta JAR

Um arquivo de manifesto é criado automaticamente para cada arquivo JAR existente e contêm uma relação dos arquivos incluídos no arquivo JAR. Este arquivo é criado num subdiretório “META-INF” e tem nome “MANIFEST.INF”.

O uso típico da ferramenta JAR é:

```
jar cf arquivoJAR *.class
```

Para criarmos um arquivo JAR com a aplicação **FontDemo** do Exemplo 86, deveríamos utilizar o seguinte comando:

```
jar cf FontDemo.jar FontDemo.class
```

Enquanto o arquivo class possui 4.174 bytes, o arquivo JAR correspondente possui apenas 2.657 bytes, ou seja, uma redução de aproximadamente 36% no tamanho.

Para indicarmos ao *browser* que a *applet* deve ser carregada a partir de um arquivo JAR devemos escrever a *tag* <APPLET> como segue:

```
<applet code="FontDemo.class" archive="FontDemo.jar"
        width="400" height="160">
</applet>
```

O mesmo procedimento pode ser adotado para aplicações e bibliotecas (pacotes) que podem empacotadas da mesma forma. No entanto o uso de pacotes requer que o arquivo JAR não seja compactado de forma que possa simplesmente ser colocado em qualquer diretório informado no CLASSPATH.

9 Arquivos

Grande parte das aplicações não pode se limitar a armazenar seus dados em memória, ou seja, não pode ter seus dados perdidos quando á aplicação é finalizada, exigindo uma forma mais perene de armazenamento que permita sua recuperação futura. Para isto devem ser utilizados arquivos, conjuntos de dados que podem ser armazenados na memória secundária de um sistema, usualmente unidades de disco rígido ou flexível, CD-ROM, DVD ou fitas magnéticas.

Para que as aplicações Java possam armazenar e recuperar seus dados através do uso de arquivos é necessário conhecermos o pacote **java.io**.

9.1 O pacote java.io

A hierarquia de classes oferecida pelo pacote **java.io** é bastante grande e relativamente complexa pois oferece 58 classes distintas para o processamento de entrada e saída em arquivos de acesso sequencial, aleatórios e compactados. Esta hierarquia pode ser parcialmente observada na Figura 73.

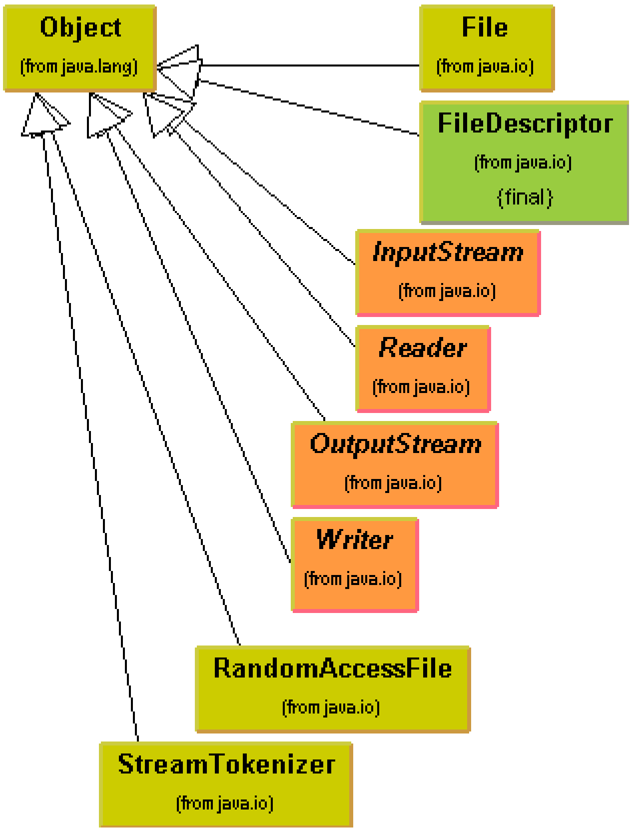


Figura 73 Hierarquia Parcial de Classes do Pacote java.io

Tal hierarquia é mais facilmente compreendida quando tratada em dois grupos básicos: um das classes voltada para entrada de dados (baseadas nas classes

java.io.InputStream e **java.io.Reader**) e outro das classes voltadas para saída de dados (baseadas nas classes **java.io.OutputStream** e **java.io.Writer)**. No momento as demais classes podem ser entendidas como auxiliares, pois oferecem serviços especializados sobre os dois grupos mencionados.

Praticamente todos os construtores e métodos deste pacote lançam a exceção **java.io.IOException** que deve ser apanhada e tratada no contexto onde ocorreu ou relançada para o contexto superior.

Tratemos agora como realizar as operações básicas de entrada e saída utilizando as classe especializadas que são oferecidas neste pacote.

9.2 Entrada

Todas a operações de entrada são realizadas por classes derivadas das classes abstratas **java.io.InputStream** e **java.io.Reader** que oferecem alguns métodos básicos que são sobrepostos e especializados nas suas classes derivadas.

9.2.1 A Família java.io.InputStream

A classe **java.io.InputStream** é uma classe abstrata significando que não pode ser usada diretamente. Ao invés disso dá origem a várias diferentes classes especializadas cujo uso típico é em camadas associadas para prover a funcionalidade desejada, tal como proposto pelo *decorator pattern*, um padrão de projeto onde objetos são utilizados em camadas para adicionar novas características a objetos mais simples de forma dinâmica e transparente, sendo que todos os objetos possuem a mesma interface. Embora um pouco mais complexo, este esquema oferece extrema flexibilidade na programação.

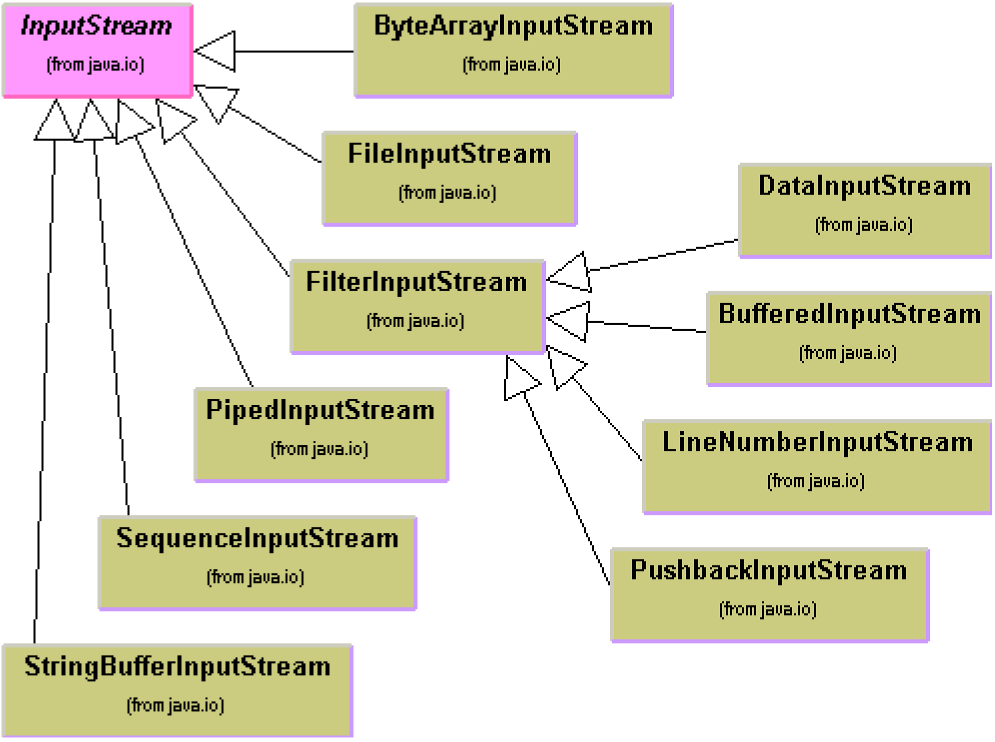


Figura 74 Hierarquia Parcial da Classe java.io.InputStream

Os tipos de **InputStream,** descritos na Tabela 61, permitem o acesso da diferentes estruturas de dados, onde cada classe especializada oferece tratamento para um destes tipos. Todas as classes derivadas de **InputStream** são apropriadas para tratamento de arquivos orientados a bytes e em formato ASCII comum.

A classe **java.io.ByteArrayInputStream** permite que um *buffer* existente em memória seja usado como uma fonte de dados. Seu construtor recebe o *buffer* e pode ser usado em conjunto com a classe **java.io.FilterInputStream** para prover uma interface útil. De forma semelhante a classe **java.io.StringBuffer** utiliza um objeto **StringBuffer** como argumento do construtor para transformar esta *string* numa fonte de dados cuja interface é dada pela associação com a classe **java.io.FilterInputStream**.

Para leitura de dados de arquivos utilizamos a classe **java.io.FileInputStream**, cujo construtor recebe um objeto **String**, **File** ou **FileDescriptor** contendo informações sobre o arquivo a ser aberto. Outra vez a associação com um objeto da classe **java.io.FilterInputStream**. permite o acesso a um interface útil para programação.

A classe **java.io.PipedInputStream** é utilizada em conjunto com a classe **java.io.PipedOutputStream** para a implementação de *pipes* (dutos de dados). Finalmente a classe **java.io.SequenceInputStream** fornece meios para concentrar a entrada de vários objetos **InputStream** numa única *stream*.

Tipo	Descrição
Vetor de Bytes	Dados são lidos de um vetor comum de bytes.
String	Dados são lidos de um objeto **StringBuffer**.
Arquivo	Dados são lidos de um arquivo existente no sistema de arquivos.
Pipe	Dados são obtidos de um *pipe*, facilidade oferecida pelo sistema operacional para redirecionamento de dados.
Sequência de Streams	Dados obtidos de diversas *streams* podem ser agrupados em uma única *stream*.
Outra fontes	Dados pode ser obtidos de outras fontes, tais como conexões com outros computadores através da rede.

Tabela 61 Tipos de InputStream

Através do uso de um **InputStream** associado a um **FilterInputStream** obtêm-se formas mais especializadas de leitura:

Tipo	Descrição
DataInputStream	Oferece suporte para leitura direta de int, char, long etc.
BufferedInputStream	Efetua a leitura de dados através de um *buffer*, aumentando a eficiência das operações de leitura.
LineNumberInputStream	Mantêm controle sobre o número da linha lida na entrada.
PushbackInputStream	Permite a devolução do último caractere lido da *stream*.

Tabela 62 Tipos de FilterInputStream

As classes **DataInputStream** e **BufferedInputStream** são as classes mais versáteis desta família pois respectivamente permitem a leitura direta de tipos primitivos da linguagem e a leitura “bufferizada” de dados.

Uma construção de classes em camadas típica é:

```
try {
  // constroi objeto stream para leitura de arquivo
  DataInputStream in = new DataInputStream (
                         new BufferedInputStream (
                           new InputStream("NomeArquivo.ext")));
  // string auxiliar para linha lida
  String linha;
  // string para armazenar todo texto lido
  String buffer = new String();
  // le primeira linha
```

```
  linha = in.readLine();
  // enquanto leitura não nula
  while (linha != null) {
    // adiciona linha ao buffer
    buffer += linha + "\n";
    // le proxima linha
    linha = in.readLine();
  }
  in.close(); // fecha stream de leitura
} catch (IOException exc) {
  System.out.println("Erro de IO");
  exc.printStackTrace();
}
```

A classe **java.io.FilterInputStream**. possui os seguintes construtores e métodos de interesse:

Método	Descrição
FilterInputStream(InputStream)	Constrói um objeto tipo **Filter** sobre o objeto **InputStream** especificado.
available()	Determina a quantidade de bytes disponíveis para leitura sem bloqueio.
close()	Fecha a *stream* e libera os recursos associados do sistema.
mark(int)	Marca a posição atual da *stream*. O valor fornecido indica a validade da marca para leituras adicionais.
markSupported()	Verifica se as operações **mark** e **reset** são suportadas.
read()	Lê o próximo byte disponível.
read(byte[])	Preenche o vetor de bytes fornecido como argumento.
reset()	Reposiciona a *stream* na última posição determinada por **mark**.
skip(long)	Descarta a quantidade de bytes especificada.

Tabela 63 Métodos da Classe java.io.FilterInputStream

O método **readLine** utilizado no exemplo pertence a classe **java.io.DataInputStream**.

9.2.2 A Família java.io.Reader

Tal qual a classe **java.io.InputStream**, a **java.io.Reader** é uma classe abstrata destinada a oferecer uma infra-estrutura comum para operações de leitura de *streams* orientadas a caractere (byte) embora suportando o padrão UNICODE. Todas as suas subclasses implementam os métodos **read** e **close**, no entanto o fazem de forma a prover maior eficiência ou funcionalidade adicional através de outros métodos. Na Figura 75 temos ilustrada a hierarquia de classes baseadas na classe **java.io.Reader**.

A classe **java.io.BufferedReader** oferece leitura eficiente de arquivos comuns quando implementada em associação a classe **java.io.FileReader**, pois evita que operações simples de leitura seja realizadas uma a uma, lendo uma quantidade maior de dados que a solicitada e armazenado tais dados num *buffer*, reduzindo o número de operações físicas. A classe **java.io.BufferedReader** serve de base para a classe **java.io.LineNumberReader**.

A classe **java.io.PipedReader** é utilizada em conjunto com **java.io.PipedWriter** para implementação de *pipes*. A classe **java.io.StringReader** é especializada na leitura de *strings* de arquivos de texto.

A classe **java.io.FilterReader**, abstrata, embora oferecendo funcionalidades adicionais, é concretamente implementada na classe **java.io.PushBackReader**, cujo uso é bastante restrito.

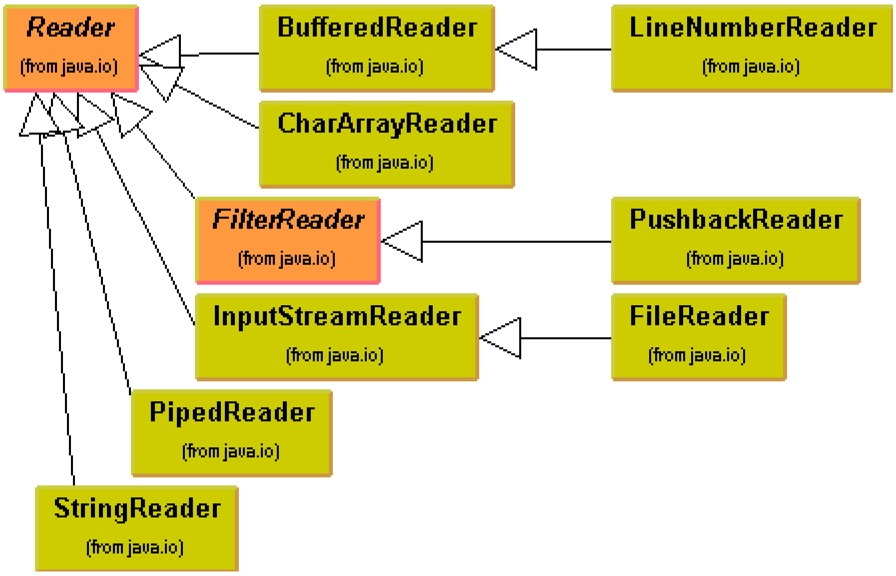


Figura 75 Hierarquia de Classes de java.io.Reader

Os construtores e métodos oferecidos pela classe **java.io.Reader** são:

Método	Descrição
Reader()	Constrói um objeto tipo **Reader** sobre o objeto **InputStream** especificado.
close()	Fecha a *stream* e libera os recursos associados do sistema.
mark(int)	Marca a posição atual da *stream*. O valor fornecido indica a validade da marca para leituras adicionais.
markSupported()	Verifica se as operações **mark** e **reset** são suportadas.
read()	Lê o próximo byte disponível.
read(byte[])	Preenche o vetor de bytes fornecido como argumento.
read(byte[], int, int)	Preenche o vetor de bytes fornecido como argumento entre as posições especificadas.
ready()	Verifica se a *stream* está pronta para leitura.
reset()	Reposiciona a *stream* na última posição determinada por **mark**.
skip(long)	Descarta a quantidade de bytes especificada.

Tabela 64 Métodos da Classe java.io.Reader

Segue um exemplo de uma aplicação simples capaz de ler o conteúdo de um arquivo texto transferindo-o para um componente **java.awt.TextArea**. O nome do arquivo é especificado num componente **java.awt.TextField** cujo evento **java.awt.event.ActionEvent** é processado na própria classe realizando a leitura do arquivo.

```
// FileRead.java

import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class FileRead extends Frame implements ActionListener {
  private TextField tfFileName;
```

```
  private TextArea taOutput;

  public static void main(String args[]) {
    FileRead f = new FileRead();
    f.show();
  }

  public FileRead() {
    super("File Read");
    setSize(320, 320);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(new Label("Arquivo"));
    p.add(tfFileName = new TextField(30));
    tfFileName.addActionListener(this);
    add("North", p);
    add("Center", taOutput = new TextArea());
    taOutput.setEditable(false);
    addWindowListener(new CloseWindowAndExit());
  }

  public void actionPerformed(ActionEvent evt) {
    try {
      BufferedReader in = new BufferedReader(
        new FileReader(tfFileName.getText()));
      taOutput.setText("");
      String line;
      String buffer = new String();
      while((line = in.readLine()) != null)
        buffer += line + "\n";
      in.close();
      taOutput.append(buffer);
    } catch (IOException exc) {
      taOutput.setText("IOException:\n"+exc.toString());
    }
  }
}
```

Exemplo 92 Classe FileRead

Abaixo uma ilustração desta aplicação.

Arquivo c:\windows\faq.txt

```
-----------------------------------------------
        LEIAME do Microsoft Windows 98
      sobre as perguntas mais freqüentes
                 Abril de 1998
-----------------------------------------------

(c) Copyright Microsoft Corporation, 1998


Este documento fornece informações complementares ou mais recentes
que completam a documentação do Microsoft Windows 98.
```

Figura 76 Aplicação FileRead

No método **actionPerformed** é criado um objeto **BufferedReader** que recebe como argumento um outro objeto **FileReader**. O primeiro providencia operações “buferizadas” enquanto segundo realiza o acesso ao arquivo propriamente dito. Através do método **readLine** obtêm-se dados do arquivo.

9.3 Saída

Todas as operações de saída são realizadas por classes derivadas das classes abstratas **java.io.OutputStream** e **java.io.Writer** as quais oferecem alguns métodos básicos que devem ser obrigatoriamente reimplementados em suas subclasses para oferecer operações mais especializadas.

9.3.1 A Família java.io.OutputStream

Analogamente a classe **java.io.InputStream**, a **java.io.OutputStream** é uma classe abstrata, ou seja, não pode ser usada diretamente, mas origina várias diferentes classes especializadas que podem ser usadas em conjunto para prover a funcionalidade desejada.

Os tipos de **OutputStream,** idênticos aos descritos na Tabela 61, permitem o envio de dados para diferentes estruturas de dados, utilizando classes especializada oferece tratamento para um destes tipos. Como **InputStream**, as classes derivadas de **OutputStream** são apropriadas para tratamento de arquivos orientados a byte e que utilizem o formato ASCII comum. Para cada classe da família **OutputStream** encontramos um correspondente na família **InputStream**. A hierarquia de classes da família **java.io.OutputStream** pode ser vista na Figura 77.

A classe **java.io.ByteArrayOutputStream** permite a criação de um *buffer* em memória, onde seu construtor recebe o tamanho inicial *buffer*, podendo ser usado em conjunto com a classe **java.io.FilterOutputStream** para prover uma interface útil. Para envio de dados para arquivos utilizamos a classe **java.io.FileOutputStream**, cujo construtor recebe um objeto **String**, **File** ou **FileDescriptor** contendo informações sobre o arquivo a ser escrito. Outra vez a associação com um objeto da classe **java.io.FilterInputStream**. permite o acesso a um interface útil para programação.

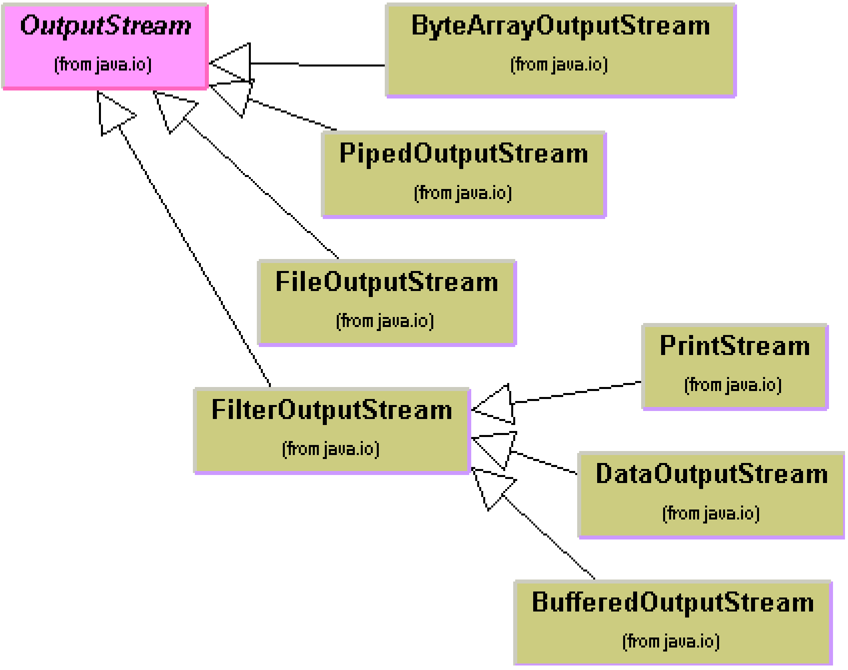


Figura 77 Hierarquia Parcial da Classe java.io.OutputStream

A classe **java.io.PipedOutputStream** é utilizada em conjunto com a classe **java.io.PipedInputStream** para a implementação de *pipes* (dutos de dados).

A classe **java.io.FilterOutputStream** serve como base para implementação de três outras classes especializadas, como mostra a Tabela 65.

Tipo	Descrição
DataOutputStream	Permite o armazenamento de dados formatados como tipos primitivos (char, int, float etc.)
PrintStream	Permite a produção de dados formatados como tipos primitivos orientado a exibição.
BufferedOutputStream	Permite otimizar as operações de escrita, realizadas em blocos.

Tabela 65 Tipos de FilterOutputStream

A classe **java.io.FilterOutputStream** possui os seguintes construtores e métodos:

Método	Descrição
FilterOutputStream(OutputStream)	Cria um objeto **FilterOutputStream** a partir da **OutputStream** especificada.
close()	Fecha a *stream* e libera os recursos associados.
flush()	Força a escrita de qualquer dado ainda presente no *buffer* da *stream*.
write(byte[])	Escreve o *array* de bytes na *stream*.
write(byte[], int, int)	Escreve as posições especificadas do *array* de bytes na *stream*.
write(int)	Escreve o bytes especificado na *stream*.

Tabela 66 Métodos da Classe java.io.FilterOutputStream

As classes derivadas de **java.io.FilterOutputStream** adicionam funcionalidade ou especializam ainda mais as operações contidas nesta classe, tal como a classe **java.io.PrintStream** que oferece os métodos **print** e **println** para obtenção de saída formatada de dados. A seguir um exemplo simples de uso destas classes na criação de um arquivo de saída:

```
try {
  DataOutputStream out = new DataOutputStream(
                         new BufferedOutputStream(
                         new FileOutputStream("Arquivo.ext")));
  out.writeBytes("Um frase simples");
  out.writeDouble(265.34554);
  out.close();
} catch (IOException exc) {
  System.out.println("Erro de IO");
  exc.printStackTrace();
}
```

Note que são enviados tipos de dados diferentes (uma *string* e um valor real) para a *stream* sendo convertidos para o formato texto pelo objeto **DataOutpuStream**.

Segue agora um exemplo de um método estático, que pode ser adicionado em qualquer classe, capaz de copia de forma eficiente um arquivo de qualquer tipo (binário ou texto). Este método recebe como argumentos o nome e diretório de origem (*source*) e de destino (*target*) utilizando as classes **BufferedInputStream**, **FileInputStream**, **BufferOutputStream** e **FileOutputStream**.

```
import java.io.*;

public static void copyFile(String source, String target) {
  try {
    // cria objeto stream para entrada
```

```
    BufferedInputStream src = new BufferedInputStream(
                                new FileInputStream(source));
    // cria objeto stream para saida
    BufferedOutputStream tgt = new BufferedOutputStream(
                                 new FileOutputStream(target));
    // verifica disponibilidade de dados na entrada
    while (src.available()>0) {
      // cria array para conter todos os dados disponiveis
      byte [] data = new byte[src.available()];
      // le todos os dados disponiveis
      src.read(data, 0, src.available());
      // grava todos os dados disponiveis
      tgt.write(data, 0, data.length);
    }
    // garante que dados serao gravados
    tgt.flush();
    // fecha streams de entrada e saida
    tgt.close();
    src.close();
  } catch (Exception e) {
    // em caso de erro exibe mensagens abaixo
    System.out.println("Copia não foi possivel:\n");
    System.out.println("de  > "+source);
    System.out.println("para> "+target);
    e.printStackTrace();
  }
}
```

Exemplo 93 Método Estático copyFile

Este método procura ler, se possível, o arquivo todo, minimizando operações físicas de IO, escrevendo de forma análoga todos os dados disponíveis Pode ser incluído em qualquer classe para prover facilidade de cópia de arquivo entre diretórios diferentes, como no exemplo abaixo que implementa uma aplicação de console que efetua a cópia um arquivo.

```
// JCopy.java

import java.io.*;

public class JCopy {
  public static final void main(String args[]) {
    if (args.length==2)
      copyFile(args[0], args[1]);
    else {
      System.out.println("Uso:");
      System.out.println("  JCopy arqOrigem arqDestino");
    }
  }

  public static void copyFile(String source, String target) {
    // utilize o método exemplificado anteriormente
    ⋮
  }
}
```

Exemplo 94 Classe JCopy

É claro que esta aplicação não pretende substituir os comandos fornecidos com os próprios sistemas operacionais mas sim demostrar a realização de cópia de arquivos de qualquer tipo.

9.3.2 A Família java.io.Writer

A classe **java.io.Writer** é uma classe abstrata que serve como superclasse para a criação de outras, mais especializadas na escrita de caracteres a *streams*. Todas as suas subclasses devem implementar os métodos **close**, **flush** e **write**, que são suas operações básicas. Como para as outras classe abstratas do pacote **java.io**, as subclasses geralmente implementam tais métodos de forma mais especializada ou adicionam novas funcionalidades. Outro aspecto importante é que a família de classe **java.io.Writer** oferece suporte para o UNICODE permitindo a internacionalização de aplicações.

Na Figura 78 temos a hierarquia de classes de **java.io.Writer**.

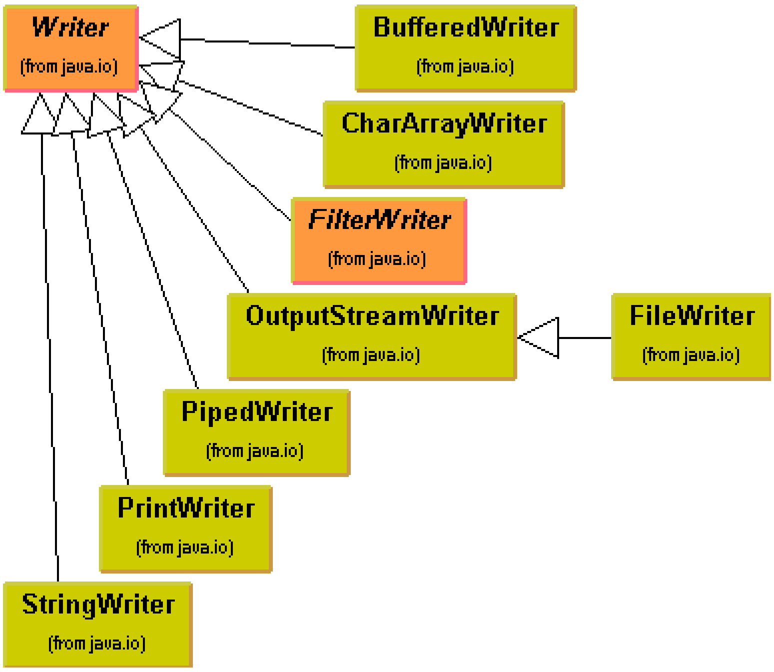


Figura 78 Hierarquia da Classe java.io.Writer

Nela podemos observar a existência de diversas classes especializadas, entre elas: **java.io.BufferedWriter** para realização de operações de escrita através de uma *stream* “buferizada”, **java.io.OutputStreamWriter** destinada a operações com arquivos, **java.io.PipedWriter** utilizada em conjunto com **java.io.PipedReader** para implementação de *pipes* e **java.io.PrintWriter** especializada em produção de saída formatada de dados.

Os principais métodos e construtores da classe **java.io.Writer** são:

Método	Descrição
Writer(Object)	Cria um objeto **Writer** sincronizado no objeto dado como argumento.
close()	Fecha a *stream*, gravando os dados existentes no *buffer* e liberando os recursos associados.
write(char[])	Escreve um *array* de caracteres na *stream*.
write(char[], int, int)	Escreve a porção especificada de um *array* de caracteres na *stream*.
write(int)	Escreve um caractere na *stream*.
write(String)	Escreve uma *string* na *stream*.
write(String, int, int)	Escreve a porção especificada de uma *string* de caracteres na *stream*.

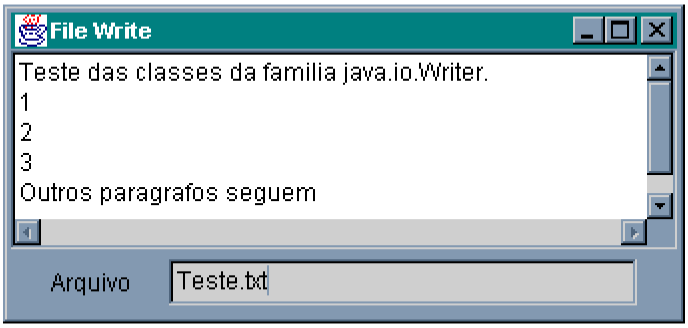
Tabela 67 Métodos da Classe java.io.Writer

Segue um outro exemplo simples de aplicação que cria um arquivo com o conteúdo de um componente **java.awt.TextArea**. O nome do arquivo é especificado num componente **java.awt.TextField** cujo evento **java.awt.event.ActionEvent** é processado na própria classe realizando a criação do arquivo.

```
// FileWrite.java

import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class FileWrite extends Frame implements ActionListener {
  private TextField tfFileName;
  private TextArea taOutput;

  public static void main(String args[]) {
    FileWrite f = new FileWrite();
    f.show();
  }

  public FileWrite() {
    super("File Write");
    setSize(320, 320);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(new Label("Arquivo"));
    p.add(tfFileName = new TextField(30));
    tfFileName.addActionListener(this);
    add("South", p);
    add("Center", taOutput = new TextArea());
    addWindowListener(new CloseWindowAndExit());
  }

  public void actionPerformed(ActionEvent evt) {
    try {
      PrintWriter out = new PrintWriter(
         new FileWriter(tfFileName.getText()));
      out.print(taOutput.getText());
      out.close();
    } catch (IOException exc) {
      taOutput.setText("IOException:\n"+exc.toString());
    }
  }
}
```

Exemplo 95 Classe FileWrite

Segue uma ilustração desta aplicação.



Teste das classes da familia java.io.Writer.
1
2
3
Outros paragrafos seguem

Teste.txt

Figura 79 Aplicação FileWrite

Observe que no método **actionPerformed** é criado um objeto **PrintStream**, que realiza as operações de saída formatada, associado a um objeto **FileWriter** que realiza efetivamente as operações sobre o arquivo. Uma única operação **print** é capaz de transferir todo o conteúdo do componente **java.awt.TextArea** para o arquivo, que é fechado imediatamente após a escrita dos dados.

Combinando-se as operações de leitura e escrita demonstradas no Exemplo 92 e no Exemplo 95 podemos construir aplicações mais sofisticadas envolvendo o processamento de texto ou escrita e leitura de dados em formato texto.

9.4 Acesso Aleatório

As classes para tratamento de *streams* vistas anteriormente permitiam apenas o acesso sequencial aos dados, isto é, a leitura de todos os dados do início do arquivo até o dado desejado. Em algumas situações o acesso sequencial aos dados existentes em arquivos se mostra inadequado pois pode ser necessária a leitura de muitos dados antes da informação requerida ser acessada, tornando tal acesso lento e consumindo recursos de forma desnecessária.. Com acesso aleatório podemos acessar diretamente uma determinada posição de um arquivo, possibilitando operações de leitura e escrita mais eficientes e adequadas para tratamento de grande volume de dados.

A classe **java.io.RandomAccess** oferece suporte tanto para leitura como para escrita em posições aleatórias de um arquivo de dados. Esta classe também oferece meios da aplicação limitar as operações que podem ser realizadas no arquivo, garantindo sua integridade.

Os principais construtores e métodos disponíveis nesta classe são:

Método	**Descrição**
RandomAcessFile(File, String)	Constrói um objeto desta classe recebendo um objeto **File** especificando o arquivo e uma *string* determinando o modo de acesso.
RandomAcessFile(String, String)	Constrói um objeto desta classe recebendo um objeto **String** especificando o arquivo e uma *string* determinando o modo de acesso.
close()	Fecha a *stream*, gravando dados pendentes e liberando os recursos associados.
getFilePointer()	Obtêm a posição corrente dentro da *stream*.
lenght()	Obtêm o tamanho total do arquivo.
read(byte[])	Lê o *array* de bytes fornecido.
readBoolean(), readByte(), readChar(), readFloat(), readInt(), readLong()	Lê um valor no formato especificado da *stream*.
readLine()	Lê uma linha do arquivo associado a *stream*.
seek(long)	Movimenta o cursor para a posição especificada.
skipBytes(long)	Avança o cursor da quantidade de bytes indicada.
write(byte[])	Escreve o *array* de bytes fornecido na *stream*.
writeBoolean(boolean), writeByte(byte), writeChar(chat), writeFloat(float), writeInt(int), writeLong(long)	Escreve o valor do tipo especificado na *stream*..

Tabela 68 Métodos da Classe java.io.RandomAccessFile

Como pode ser observado, esta classe oferece inúmeros métodos para a escrita e leitura de tipos primitivos facilitando seu emprego. No entanto, para utilização de arquivos com acesso aleatório com dados de tipo diferente dos primitivos devemos trabalhar com uma classe que encapsule os dados escrevendo e lendo os mesmos através dos métodos **read** e **write**.

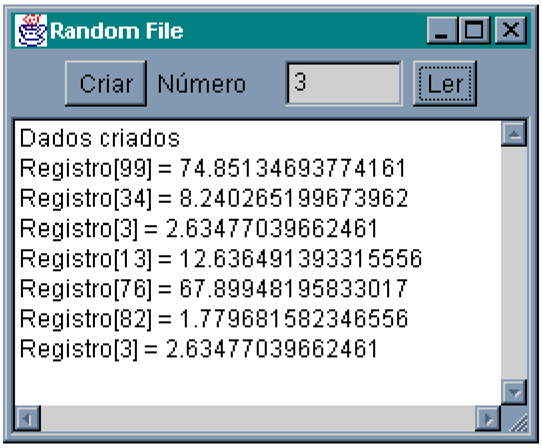
No Exemplo 96 temos um classe que demonstra a criação e leitura de dados através da classe **java.io.RandomAccessFile**. Um arquivo de dados de teste pode ser criado através da aplicação. Registros individualmente especificados pelo usuário podem ser lidos e apresentados na área central da aplicação.

```
// RandomFile.java

import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class RandomFile extends Frame
                        implements ActionListener {
  private Button bCreate, bRead;
  private TextField tfRecord;
  private TextArea taOutput;

  public static void main(String args[]) {
    RandomFile f = new RandomFile();
    f.show();
  }

  public RandomFile() {
    super("Random File");
    setSize(320, 320);
    Panel p = new Panel();
    p.setBackground(SystemColor.control);
    p.add(bCreate = new Button("Criar"));
    p.add(new Label("Número"));
    p.add(tfRecord = new TextField(5));
    p.add(bRead = new Button("Ler"));
    bCreate.addActionListener(this);
    bRead.addActionListener(this);
    add("North", p);
    add("Center", taOutput = new TextArea());
    addWindowListener(new CloseWindowAndExit());
  }

  public void actionPerformed(ActionEvent evt) {
    if (evt.getSource()==bCreate) {
      try {
        RandomAccessFile arq =
                    new RandomAccessFile("dados.dat", "rw");
        for (int i=0; i<100; i++)
          arq.writeDouble(i*Math.random());
        arq.close();
        taOutput.setText("Dados criados");
      } catch (IOException exc) {
        taOutput.setText("Erro na criação de arquivo:\n" +
                         exc.toString());
      }
    } else {
      try {
        RandomAccessFile arq =
                    new RandomAccessFile("dados.dat", "r");
        int num = Integer.parseInt(tfRecord.getText());
        arq.seek(num*8);
        double dado = arq.readDouble();
```

```
    taOutput.append("\nRegistro["+num+"] = " + dado);
    arq.close();
   } catch (IOException exc) {
    taOutput.setText("Erro na leitura de arquivo:\n" +
             exc.toString());
   }
  }
 }
}
```

Exemplo 96 Classe RandomFile

A aplicação do Exemplo 96 encontra-se ilustrada baixo.



```
Dados criados
Registro[99] = 74.85134693774161
Registro[34] = 8.240265199673962
Registro[3] = 2.63477039662461
Registro[13] = 12.636491393315556
Registro[76] = 67.89948195833017
Registro[82] = 1.779681582346556
Registro[3] = 2.63477039662461
```

Figura 80 Aplicação RandomFile

9.5 Obtendo Informações de Arquivos e Diretórios

A classe **java.io.File** permite a obtenção de informações sobre arquivos e diretórios, além de algumas operações elementares sobre estes, possibilitando seu tratamento automatizado por parte de aplicações Java.

Esta classe oferece meios para o tratamento de aspectos relacionados ao sistema de arquivos que são dependentes de cada plataforma tal como o caractere separador de diretórios. Os principais construtores e métodos desta classe são relacionados na Tabela 69.

Através desta classe é possível efetuar a distinção de arquivos e diretórios. Pode-se determinar se um arquivo específico existe, seu tamanho, se pode ser lido ou escrito, bem como renomeá-lo ou eliminá-lo.

Com relação aos diretórios é possível criá-los, renomeá-los ou obter uma relação contendo todos os arquivos e subdiretórios contidos neste que podem ser posteriormente processados. Segue um exemplo:

```
// Jdir.java

import java.io.*;

public class JDir {

 public static final void main(String args[]) {
  if (args.length != 1) {
   System.out.println("Numero invalido de argumentos");
  } else {
   File f = new File(args[0]);
   if (f.isDirectory()) {
    String arquivos[] = f.list();
    System.out.println("Diretorio de " + args[0]);
    for(int i=0; i<arquivos.length; i++)
```

```
          System.out.println(arquivos[i]);
      } else {
        System.out.println("Não e um diretorio");
      }
    }
  }
}
```

Exemplo 97 Classe JDir

Método	Descrição
File(File, String)	Cria um objeto **File** representando o arquivo especificado através do objeto **File** correspondente ao diretório e o nome do arquivo dados.
File(String)	Cria um objeto **File** representando o arquivo especificado através do caminho dado.
File(String, String)	Cria um objeto **File** representando o arquivo no diretório especificado através do caminho e do nome do arquivo.
canRead()	Verifica se a aplicação pode ler do arquivo representado.
canWrite()	Verifica se a aplicação pode escrever no arquivo representado.
delete()	Remove o arquivo representado por este objeto.
exists()	Determina se o arquivo existe.
getAbsolutePath()	Retorna o caminho absoluto representado por este objeto.
getCanonicalPath()	Retorna o caminho canônico representado por este objeto.
getName()	Retorna o nome do arquivo representado por este objeto.
getPath()	Retorna o caminho representado por este objeto.
isAbsolute()	Verifica se o caminho associado é absoluto.
isDirectory()	Verifica se o objeto associado é um diretório.
isFile()	Verifica se o objeto associado é um arquivo.
lastModified()	Retorna o horário da última modificação do arquivo associado.
length()	Retorna o tamanho do arquivo.
list()	Retorna uma lista de arquivos contidos neste diretório.
mkdir(0	Cria o diretório especificado pelo caminho deste objeto.
renameTo(File)	Renomeia o arquivo representado pelo objeto conforme o novo objeto especificado.

Tabela 69 Métodos da Classe java.io.File

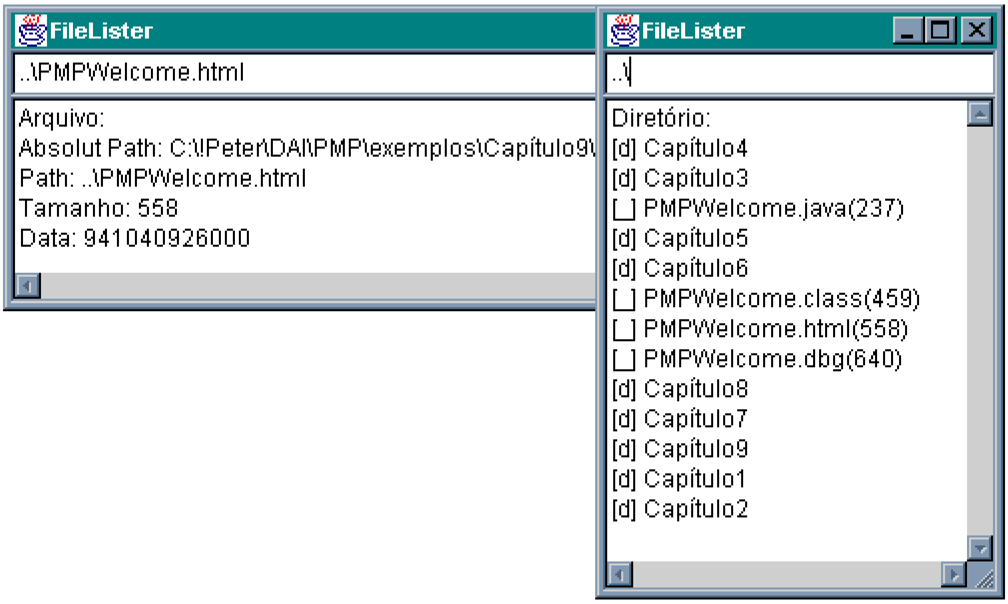
Segue o código de uma aplicação gráfica que permite a obtenção de dados de um diretório ou arquivo, exibidos num componente **java.awt.TextArea**. No método **actionPerformed** determina-se se o nome fornecido é válido ou não, correspondendo a um arquivo ou diretório. Conforme o caso são exibidas informações sobre o arquivo ou uma listagem dos arquivos e subdiretórios do diretório especificado.

```
// FileLister

import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class FileLister extends Frame
                      implements ActionListener {

  private TextField path;
  private TextArea output;
```

```
  public FileLister() {
    super ("FileLister");
    path = new TextField();
    path.addActionListener(this);
    add(path, BorderLayout.NORTH);
    output = new TextArea();
    add(output, BorderLayout.CENTER);
    setSize(320, 320);
    setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {
    File name = new File(e.getActionCommand());
    if (name.exists()) {
      if (name.isDirectory()) {
        String directory[] = name.list();
        output.setText("Diretório:\n");
        for (int i=0; i<directory.length; i++) {
          File file = new File(name.getPath()+directory[i]);
          if (file.isFile()) {
            output.append("[_] " + directory[i] +
                          " (" + file.length() + ")\n");
          } else {
            output.append("[d] " + directory[i] + "\n");
          }
        }
      } else {
        if (name.isFile()) {
          output.setText("Arquivo:\n");
          output.append("Absolut Path: " +
                         name.getAbsolutePath()+
                        "\nPath: " + name.getPath()+
                        "\nTamanho: " + name.length() +
                        "\nData: " + name.lastModified() +"\n");
        } else {
          output.setText("Arquivo inválido.\n");
        }
      }
    } else {
      output.setText("Caminho ou arquivo especificado não existe
ou inválido.\n");
    }
  }

  public static void main(String args[]) {
    FileLister f = new FileLister();
    f.addWindowListener(new CloseWindowAndExit());
  }
}
```

Exemplo 98 Classe FileLister

Segue uma ilustração da aplicação sugerida mostrando dados de um diretório e de um arquivo.



```
..\PMPWelcome.html

Arquivo:
Absolut Path: C:\!Peter\DAI\PMP\exemplos\Capítulo9\
Path: ..\PMPWelcome.html
Tamanho: 558
Data: 941040926000
```

```
..\

Diretório:
[d] Capítulo4
[d] Capítulo3
☐ PMPWelcome.java(237)
[d] Capítulo5
[d] Capítulo6
☐ PMPWelcome.class(459)
☐ PMPWelcome.html(558)
☐ PMPWelcome.dbg(640)
[d] Capítulo8
[d] Capítulo7
[d] Capítulo9
[d] Capítulo1
[d] Capítulo2
```

Figura 81 Aplicação FileLister

10 Bibliografia

ARNOLD, Ken e GOSLING, James. *The Java Programming Language*. 2nd Edition. Reading, MA: Addison-Wesley, 1998.

BELL, Doug. *Make Java Fast: Optimize!* IN JavaWorld, April/1997.
[Internet: http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize_p.html]

CESTA, André Augusto. *Tutorial: A linguagem de programação Java – Orientação à Objetos*. Apostila. Campinas, SP: Unicamp, 1996.
[Internet: http://www.dcc.unicamp.br/~aacesta]

CRAWFORD III, George. *A Practical Crash Course in Java 1.1+ Programming and Technology: Part I*. IN ACM Crossroads, April/1998.
[Internet: http://www.acm.org/crossroads/xrds4-2/ovp42.html]

_________. *A Practical Crash Course in Java 1.1+ Programming and Technology: Part II*. IN ACM Crossroads, April/1998.
[Internet: http://www.acm.org/crossroads/xrds4-3/ovp.html]

_________. *Java AWT Layout Management 101*. IN ACM Crossroads, May/1998.
[Internet: http://www.acm.org/crossroads/xrds5-1/ovp51.html]

CYMERMAN, Michael. *Smarter Java Development*. IN JavaWorld, August/1999.
[Internet: http://www.javaworld.com/javaworld/jw-08-1999/jw-08-interfaces_p.html]

DEITEL, Harvey M., DEITEL, Paul J. *Java: How to Program*. 2nd Edition. Upper Saddle River, NJ: Prentice-Hall, 1998.

ECKEL, Bruce. *Thinking in Java*. Upper Saddle River, NJ: Prentice-Hall, 1997.
[Internet: http://www.eckelobjects.com/javabook.html]

FIRSTPERSON., *Oak Language Specification*. Palo Alto, CA: FirstPerson Inc., 1994.

GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J.. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addisson-Wesley, 1994.

GEARY, David M.. *Graphic Java - Mastering the AWT*. Upper Saddle River, NJ: Prentice-Hall, 1997.

GOSLING, J. *A Brief History of the Green Project*.
[Internet: http://java.sun.com/people/jag/green/index.html]

HORSTMANN, Cay S. & CORNELL, Gary. *Core Java Volume I - Fundamentals*. Upper Saddle River, NJ: Prentice-Hall, 1997.

_________ *Core Java Volume II - Advanced Features*. Upper Saddle River, NJ: Prentice-Hall, 1998.

KRAMER, Douglas. *The Java Platform: A White Paper*. Montain View, CA: Sun Microsystems Inc., 1996.

LIU, Chunyen. *Jazz up the standard Java fonts*. IN JavaWorld, October/1999.
[Internet: http://www.javaworld.com/javaworld/javatips/jw-javatip81.html]

McCLUSKEY, Glen. *Remote Method Invocation: Creating Distributed Java-toJava Applications* IN Java Developer Connection, September/1998.
[Internet: http://developers.java.sun.com/developers/TechnicalArticles/Mccluskey/rmi.html]

McGRAW, Gary & FELTEN, Edward. *Twelve rules for developing more secure Java code*. IN JavaWorld, December/1998.
[Internet: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html]

MORISSON, Michael et al. *Java 1.1 Unleashed*. Indianapolis, IN: Sams, 1997.

ORFALI, R. & HARKEY, D. *Client/Server Programming with Java and CORBA*. 2nd Edition. New York, NY: J. Wiley, 1998.

RUMBAUBH, J., BLAHA, M., PREMERLANI, W., EDDY, F. e LORENSEN, W. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

SUN. *Java Platform API Specification Version 1.1.7*. Palo Alto, CA: Sun Microsystems Inc., 1998.

VENNERS, Bill. *Designing with Interfaces*. IN JavaWorld, December/1998.
[Internet: http://www.javaworld.com/javaworld/jw-12-1998/jw-12-techniques.html]

WALNUM, Clayton. *Java Em Exemplos*. São Paulo, SP: Makron Books, 1997.

WEB DESIGN GROUP, *HTML 4.0 Reference*. Recomendação do W3C – WWW Consortiun, 1997.
[Internet: http://www.w3c.org/]

WUTKA, Mark et. al. *Java Expert Solutions*. Indianapolis, IN: Que, 1997.
[Internet: http://www.kaposnet.hu/books/hackjava/index.htm]

ZUCHINI, M. H., SILVEIRA, L. G. Jr., JANDL, P. Jr., LIMA, T. C. B. & CASOTTI, F. A. G. *Orientação à Objetos: uma visão em C++*. Apostila. Itatiba, SP: Universidade São Francisco, 1997.