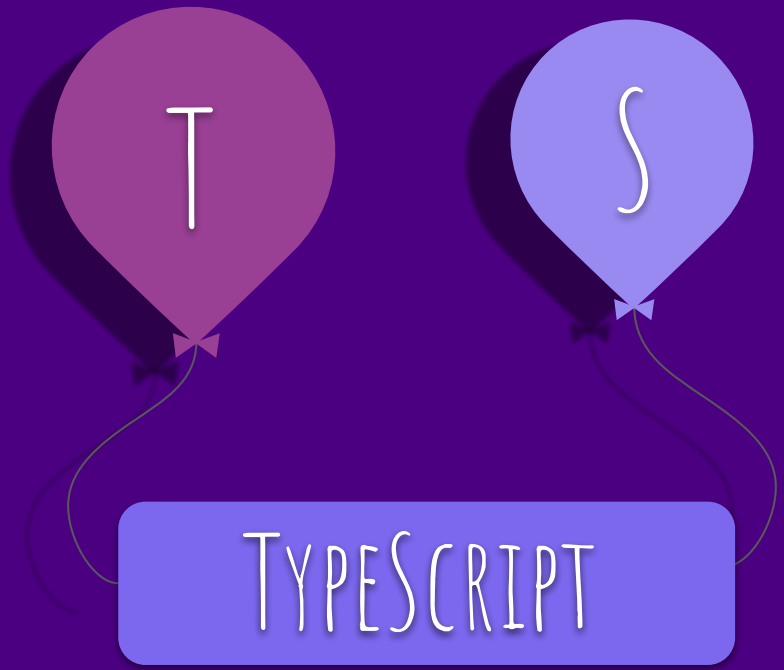
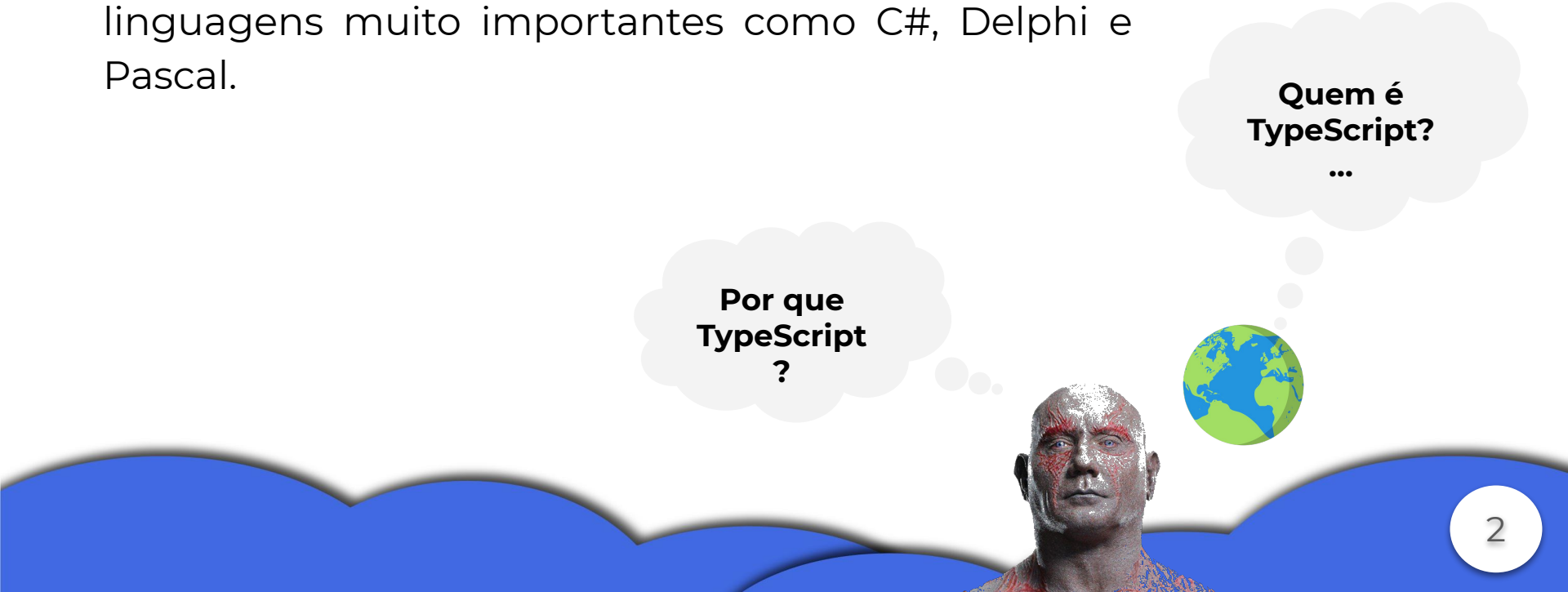


Gabriel Soares Xavier
Gabriel Ferrari Wagnitz
Iury Candeias Nogueira
Mikaella Ferreira Da Silva

Lançado pela Microsoft, o TypeScript foi criado por um Arquiteto de software chamado Anders Heljsberg, que participou da criação de outras linguagens muito importantes como C#, Delphi e Pascal.



- **Quem é TypeScript?**

  Podemos chamar TypeScript como um super conjunto da linguagem Javascript, com objetivo de elevar o nível de javascript, que antes era usado em códigos relativamente pequenos e simples.

- **Por que TypeScript?**

  O TypeScript trouxe consigo além da tipagem estática, o paradigma Orientação a Objeto, que facilita e viabiliza o desenvolvimento de softwares de longa “carga”, o que era requisitado na época já que o JavaScript não dava suporte.

Lançada em 2012, resultado de dois anos de trabalho interno da microsoft.

Em 2016, introduzindo novos recursos.

“ser null ou não ser?”.

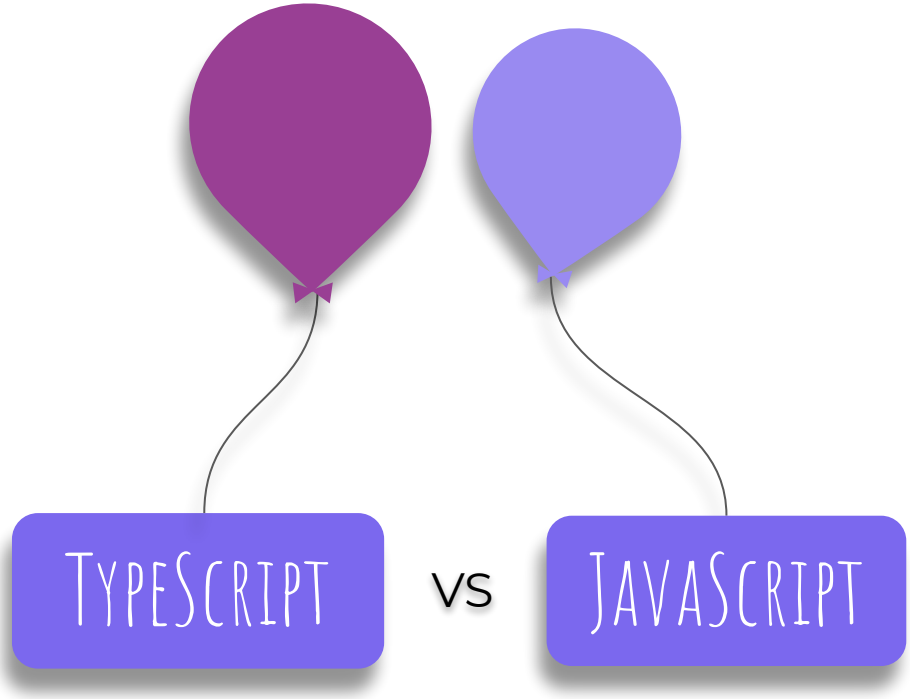
Versão atual!

A linguagem foi se aprimorando cada vez mais, com pequenas atualizações.

Escada de versionamentos.

- **Por que eu deveria usar TypeScript?**
    - TypeScript está em constante aprimoramento.
    - possui uma comunidade ativa, foi introduzido e é mantido pela microsoft (as pessoas que querem colaborar tem total brecha para tal, OpenSource).
    - possui compatibilidade com JavaScript.
    - A linguagem é muito usada para desenvolvimentos web de grande porte;
    - Sua documentação é completa e sofre atualização com as novas mudanças.

- O TypeScript praticamente pode ser executado em todos navegadores em qualquer ambiente, SO e dispositivos.
- Qualquer coisa que rode JavaScript irá rodar TypeScript.
- Não é necessário uma máquina virtual pessoal ou um ambiente de tempo de execução para poder executar os códigos.


TypeScript
vs
JavaScript


JS
JavaScript
Who are you?
TS
TypeScript
I AM you, but BETTER!

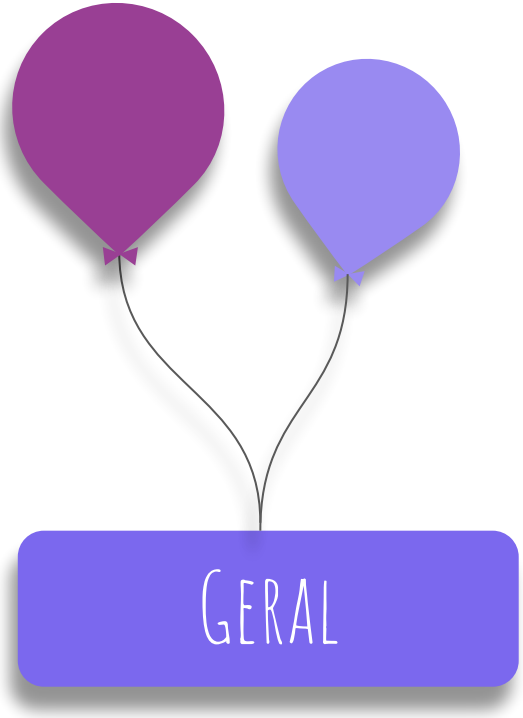
- TypeScript é considerada pela comunidade como uma linguagem fortemente tipada, diferente do JavaScript, TypeScript possui tanto tipagem dinâmica quanto estática.

  + Legibilidade, muito mais fácil de entender um código tipado estaticamente.

  + Confiabilidade, ao se usar tipagem estática, o compilador verifica as operação em cima dos tipos.

  - Redigibilidade, identificar sempre o tipo nas declarações e definições (o que não é muito a se perder!).

- Comunidade do JavaScript ainda é maior que a do TypeScript.
- É necessário compilar códigos TypeScript, já em JavaScript não é necessário.
- TypeScript possui a verdadeira Orientação a Objetos, enquanto JavaScript usa de recursos para simular.



- Fortemente tipada
- Orientada a Objetos e Funcional
- Suporta Tipos Genéricos
- O código é compilado para JavaScript
- TypeScript extends JavaScript
- TypeScript segue a especificação da ECMAScript 2015 (ou seja padrão ECMA-262)
- Suporte Unicode
- CaseSensitive

- **Espaços em branco e quebra de linha.**
    - TypeScript ignora os espaços em branco e quebras de linha, ou seja o programador é livre para formatar o código da maneira que achar melhor.
- **Ponto e vírgula é opcional.**

```
console.log("quero ;");
console.log("não quero ;")
console.log("aqui"); console.log("é obrigado ;");
```

- **Comentários**
  - // para comentários de uma linha
  - /* */ para comentários de várias linhas

- **Identificadores**
  - Podem conter caracteres e dígitos. No entanto, não podem começar com um dígito.
  - Não podem conter símbolos especiais, com exceção de “_” e “$”, e pode conter acentuação.
  - Não podem ser palavras chaves. (break, type, private, for, finally...)
  - Não há limite de tamanho mencionado na documentação.
  - CaseSensitive.
  - Não podem conter espaços.

- **Variáveis**
  - Para declarar variáveis no TypeScript podemos usar de três formas, let, var e const.
  - Var, let e const possuem a mesma sintaxe:

    let <NomeDaVariável>:Tipo = Valor;
    const <NomeDaVariável>:Tipo = Valor;
    var <NomeDaVariável>:Tipo = Valor;

  - Podemos inferir ou não valores às variáveis durante sua declaração.
  - Podemos ou não dizer o tipo de dado que ela agrega.

Exemplos :

```
const UmaConstante: number = 2;
let UmaVariavel;
var UmaOutraVariavel = "Xananam";
let SouUmaVariávelComAcentuação = "xixiriri";
var ಠ_ಠ = "????"; (SIM ISSO É VÁLIDO)
```

- Podem receber funções como valores.

```
let x = function () {
    return "Ferias!!!";
}
console.log(x());//Ferias!!!
```

```
var x = function () {
    return ":(";
}
console.log(x());//:(
```

- **Let vs Var**
  - Escopo (Var)

    Diferente do costume, declarações var possuem escopo de função.

```
function m1(): void {
    var x: boolean = true;
    if (x) {
        var loucura = "Xoxo"
    }
    console.log(loucura);//Xoxo
}
```

Ainda sobre var, existe o conceito de elevação, onde você pode declarar variáveis globais no final do código e em tempo de execução ela ser elevada ao topo. O mesmo ocorre para subprogramas.

```
function m1(): void {
    console.log(opa);
}
m1();//undefined
var opa = "lelê";
m1();//lelê
```

```
function m3(): void {
    console.log(x);
    var x;
}
m3();//undefined
```

- Escopo (Let)

  Let declarações já se assemelham com a maioria das linguagens e possui escopo de bloco.

```
function m2(): void {
    let x: boolean = true;
    if (x) {
        let b;
    }
    console.log(b);//error TS2304: Cannot find name 'b'
}
```

O conceito de elevação só se aplica para let quando se trata de variáveis globais.

```
function m3(): void {
    console.log(x);
    let x;
}
m3();//error escopo da
variável 'x' usada antes da
declaração
```

```
function m4(): void {
    console.log(eleve);
}
let eleve = "fui elevado";
m4();//fui elevado
```

- Re-declaração e sombreamento

  Utilizando var, múltiplas declarações de uma mesma variável é válida!

```
function multDeclaracao(){
    var x = 1;
    var x = 2;
    var x; // erro, x só pode ser do tipo number
    var x = "Errei denovo";//erro
}
```

O sombreamento em declarações var são limitadas de duas formas.

```
var q = 1;
function f() {
    var q = 2;
    console.log(q);
}
f();//2
console.log(q);//1
```

```
function f1() {
    var x = 1
    function f2() {
        var x = 2
     console.log(x);//2
    }
    f2();
    console.log(x);//1
}
```

Nesse caso não ocorre sombreamento!

```
function f3() {
    var x = "ola";
    if (true) {
        var x = "mudei";
    }
    console.log(x);//mudei
}
```

Com let o sombreamento é totalmente possível, não há múltiplas declarações.

```
function f3() {
    let sombra = "ainda sou eu";
    if (true) {
        let sombra = "não sou mais eu";
        console.log(sombra);//não sou mais eu
    }
    console.log(sombra);//ainda sou eu
}
```

- **Const x Readonly**
    - const é para variáveis enquanto readonly são para propriedades de objetos.
    - consts são imutáveis, readonly podem ser alteradas dependendo da situação.

```
var obj: { readonly name: string
} = {
    name: "",
};
function mudando(obj: { name:
string }) {
    obj.name = "SOU ALGUÉM!";
}
mudando(obj);
console.log(obj.name);//SOU
ALGUÉM!
```

```
const nome = "nada";
function mudando(nome:
string) {
    nome = "Alguém";

console.log(nome);//Alguém
}
mudando(nome);
console.log(nome);//nada
```

- **Escopo**

TypeScript possui escopo estático e aninhado (Closures).

```
function f1() {
    let x = 1;
    function f2() {
        return x *= 10;
    }
    console.log(f2());
}
f1();//10
```

Tipos

Compostos Vs Primitivos

- **Primitivos**
  - number: Representa valores de ponto flutuante IEEE 754 de precisão dupla de 64 bits.
  - boolean: Valores true e false.
  - string: Sequência de caracteres armazenados como unidades de código Unicode UTF-16
  - symbol: Representa tokens exclusivos que podem ser usados como chaves para propriedades de objetos.

- void: Representa a ausência de valor e é usado como o tipo de retorno de funções sem valor para retorno. Valores possíveis: undefined ou null.
- null: Faz referência a o primeiro e único valor do tipo Null.
- undefined: Denota o valor dado a todas variáveis não inicializadas.
- enum: São subtipos definidos pelo usuário do tipo primitivo number.
- literais: São tipos de valores exatos.

Exemplos

```
var aux: number = 100;
var aux1: boolean = true;
var aux2: string = "Yooh";

var aux5: void = null;
var aux6: null = null;
var aux7: undefined = undefined;
enum aux8 { Red, Blue, Green,};
```

```
var nome:symbol = Symbol();
var obj = {};
obj[nome] = "Alguém";
console.log(obj[nome]);//Alguém

type newType = "Norte" | "Sul"
| 10 | true;
var aux9: newTipe = 10;//Okay
aux9 = "Norte";//Okay
aux9 = "X"; //ERROR
```

- **Compostos**
  - Array: Lista de elementos do mesmo tipo.
  - Tupla: Tuplas são listas fixas de elementos heterogêneos.
  - Objeto: É qualquer coisa que não seja um tipo primitivo, (pode ser anônimo).
  - União: Pode assumir valores do conjunto de tipos adotados.
  - Interseção: Conjunto dos valores em comum.
  - Mapeado: Derivar um tipo de objeto em outro.

Exemplos

```
let list: number[] = [1, 2, 3];
let tupla: [number, number] = [1.0, 2.0];
var objeto = {
    nome: "nome",
    idade: 10,
}
var x: string | number;
x = 10;//okay
x = "poh"//okay
x = true;//error
```

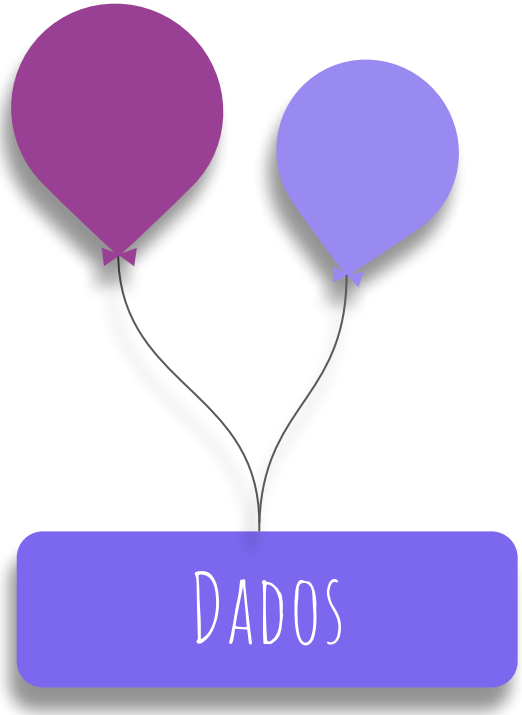
```
type tipo1 = string | number | boolean;
type tipo2 = string | boolean;
var t: tipo1 & tipo2;
t = "oi";//okay
t = true;//okay
t = 1;//error
```

```
type x = 'a' | 'b' | 'c';
type map = { [keys in x]:
string };
let v: map = { a: "a", b:
"b", c: "c" };
console.log(v.a);//a
```

- **Tipos especiais**
  - any: Assumem qualquer valor de tipos possíveis e desativa a checagem estática.
  - never: Representa os valores que nunca ocorrem, normalmente usado para funções que sempre retornam erro.

```
let aux10: any;
aux10 = 20;//okay!
aux10 = "eeeeba";//okay!
aux10 = true;//okay!
aux10 = {
nome: "Vitor",
idade: 30
};//okay!
```

```
let aux11: never;
aux11 = 1;//erro
aux11 = "s";//erro
aux11 = true;//erro
aux11 = null;//erro
aux11 = undefined; //erro
function f(): never {
    throw new Error("Útil?");
}//okay!
```



- **Persistência de dados**
  - Necessário importar como módulo a biblioteca fs do node.js
    - *readFile (path,options,callback) e writeFile(path,contents,options,callback)*
    - *readFileSync(path,options) e writeFileSync(path,contents,options)*

- **readFile e writeFile**

```
import * as fs from 'fs';
fs.writeFile("cof.txt", "GRRRRRRRRR", { 'flag': 'w' }, function
(err) {
    if (err) throw err
    console.log("success");
});
fs.readFile("cof.txt", { 'flag': 'r' }, function (err, data) {
    if (err) throw err;
    console.log(data.toString());//GRRRRRRRRR
})
```

- **readFileSync writeFileSync**

```
fs.writeFileSync("cof.txt",{ 'flag': 'w' },"au au au");
var file = fs.readFileSync("cof.txt");
console.log(file.toString());//au au au
fs.writeFileSync("cof.txt", "\nmiau miau ", { 'flag': 'a+' });
file = fs.readFileSync("cof.txt");
console.log(file.toString());//au au au\nmiau miau
```

- **Coletor de Lixo**
  - Marcar e varrer
  - Contagem de referência
- **Serialização**
  - JSON

    O TypeScript herda do JavaScript as serializações realizadas pelo JSON, existem plugins da comunidade que facilitam esse processo.

```
class Cachorro {
    name: string = "Dog-da-ufes";
    idade: number = 100;
    getName() {
        return this.name;
    }
    static fromJson(d: Object): Cachorro {
        return Object.assign(new Cachorro, d);
    }
}
```

```
var obj = new Cachorro;
var serializado = JSON.stringify(obj);
var obj_dnv = Cachorro.fromJson(JSON.parse(serializado));
console.log(serializado);//{"name":"Dog-da-ufes","idade":100}
console.log(obj_dnv.getName());//Dog-da-ufes
```

- **Operadores**
  - Operadores unários
  - Operadores binários
  - Operadores ternários
  - O operador this
  - O operador Spread

- Operadores unário

Operadores Aritméticos	
++	Incremental, prefixo ou posfixo, semelhante a C.
--	Decremental, prefixo ou posfixo, semelhante a C.

Operadores Bit a Bit	
~	Inverte a cadeia de bits.

Operadores especiais	
+	Converte qualquer tipo em number, positivo.
-	Converte qualquer tipo em number, negativo.
!	Transforma qualquer valor em booleano, pode ser combinado consigo mesmo várias vezes. Nega valores lógicos.
typeof	Pega um operando qualquer e gera o tipo do operando em forma de string.
delete	Deleta um elemento de um array ou propriedade de um objeto, se bem sucedido retorna true, senão, false.

Exemplos

```
let aux = 10;
console.log(aux++)//10
console.log(++aux)//12
console.log(aux--)//12
console.log(--aux)//10
```

```
let x = 10;
let v = '100';
let b = true;
console.log(+x);//10
console.log(+v);//100
console.log(+b);//1
```

```
let x = 10;
let v = '100';
let b = true;
console.log(-x);//-10
console.log(-v);//-100
console.log(-b);//-1
```

```
let x = 10;
let v = '100';
let b = true
console.log(~x);//-11
console.log(~v);//-101
console.log(~b);//-2
```

```
let x = 10;
let v = '100';
let b = true
console.log(!x);//false
console.log(!!v);//true
console.log(!!!b);//false
```

```
let x = 10;
console.log(typeof x);//number
let y: typeof x = "";//erro

var array = [1, 2, 3, 4]
console.log(delete
array[1]);//true
console.log(array);//[1,undefined,
3,4]
```

- Operadores Binários

Operadores Aritméticos	
*	Multiplicação
/	Divisão
%	Módulo
-	Subtração
**	Exponenciação
+	Soma

Operadores Bit a Bit	
^	Retorna 1 para cada posição de bit que um dos operandos possui 1 e o outro 0.
&	Retorna 1 em cada posição de bit para qual ambos operandos tem 1.
|	Retorna 1 para cada posição de bit quando ao menos um dos operandos tem 1.
<<	Desloca x quantidade de bits à esquerda, mandando 0s à direita.
>>	Descola x quantidade de bits à direita, descartando os bits que se tornam off.

>>>	Descola x quantidade de bits à direita, descartando os bits que se tornam off, e mandando os 0s a esquerda.

Operadores de Comparação	
>	Retorna true se o operando à esquerda for maior que o da direita, senão, retorna false.
<	Retorna true se o operando à esquerda for menor que o da direita, senão, retorna false.
<=	Retorna true se o operando à esquerda for menor ou igual ao da direita, senão, retorna false.

>=	Retorna true se o operando à esquerda for maior ou igual ao da direita, senão, retorna false.
==	Retorna true se o operando da esquerda for igual da direita, senão, retorna false.
!=	Retorna true se o operando da esquerda for diferente do da direita, senão, retorna false.
===	Retorna true se o operando da esquerda for igual em valor e em tipo ao da direita, senão, retorna false.
!==	Retorna true se o operando à esquerda for diferente em valor e/ou diferente em tipo, senão, retorna false.

Operadores Lógicos	
&&	Se os operandos forem expressões, retorna a expressão da esquerda se essa for possível converter para false, senão, a da direita. Caso os operandos sejam valores booleanos, retorna true se ambos forem true, senão, false.
||	Se os operandos forem expressões, retorna a expressão da esquerda se essa for possível converter para true, senão, retorna a direita. Caso os operandos sejam valores booleanos, retorna true se ao menos um for true, senão, false.

Operadores Relacionais	
in	Retorna true se a propriedade está presente no objeto.
instanceof	Retorna true se o objeto especificado a esquerda for do tipo do objeto a direita.

Operadores de Atribuição	
=	Atribui o valor a direita ao elemento da esquerda.
+=	Atribui o valor da direita somado com o valor do elemento a esquerda, ao elemento a esquerda.

-=	Atribui o valor da direita subtraído pelo valor do elemento a esquerda, ao elemento a esquerda.
***=**	Atribui o valor da direita multiplicado pelo valor do elemento a esquerda, ao elemento a esquerda.
/=	Atribui o valor do elemento a esquerda dividido pelo valor da direita, ao elemento a esquerda.
%=	Atribui o resto da divisão do valor do elemento a esquerda pelo da direita, ao elemento da esquerda.
****=**	Atribui o valor do elemento a esquerda elevado o valor da direita, ao elemento da esquerda.

<<=	Atribui o valor do elemento a esquerda deslocado o valor da direita bits para a esquerda, ao elemento da direita.
>>=	Atribui o valor do elemento a esquerda deslocado o valor da direita bits para a direita, ao elemento da direita.
>>>=	Atribui o valor do elemento a esquerda deslocado o valor da direita bits para a direita, ao elemento da direita. De forma não assinada.
&=	Atribui a operação & bit a bit do elemento a esquerda com o da direita.
^=	Atribui a operação ^ bit a bit do elemento a esquerda com o da direita.

|=	Atribui a operação | bit a bit do elemento a esquerda com o da direita.

Exemplos

```
var x = 10;
var y = 2;
x ^= y;
console.log(x);//8
```

```
x <<= y;
console.log(x);//32
x >>>= y;
console.log(x);//8
```

```
var z: { p1: number,
p2: number } = { p1: 1,
p2: 2 };
var w: { p1: number,
p2: number } = { p1: 1,
p2: 2 };
console.log(z ===
w);//false
```

```
var a = x && y;
var b = x || y;
console.log(a);//2
console.log(b);//8

console.log('p1' in
z);//true
console.log(w instanceof
Object);//true

 x = y = 9;
```

- Operador Ternário

Operadores Condicionais	
exp ? retorno 1 : retorno 2	Questiona se a exp pode ser atribuída como true, se sim, retorna o retorno 1, senão, o retorno 2.

```
var a = { name: "ciclano", idade: -10 };
var b = a.idade ? a : null;
console.log(b);//{name:'ciclano',idade:-10}
```

- O operador this

Em TypeScript o this continua sendo uma palavra chave de referência a um objeto, só que com algumas armadilhas de escopo.

```
var a = {
    name: "ciclano", idade: -10, imprime() {
        console.log(this.name, this.idade);
    }
};          a.imprime();//ciclano -10
            var p = a.imprime;
            p();//undefined undefined
```

```
var a = {
    name: "ciclano", idade: -10,
 imprime() {
        console.log(this.name, this.idade);
        function imprime2() {
            console.log(this.name, this.idade);//erro
        }
        imprime2();
    }
};
```

```
var a = {
    name: "ciclano", idade: -10, imprime() {
        console.log(this.name, this.idade);
        var _this = this;
        function imprime2() {
            console.log(_this.name, _this.idade);
        }
        imprime2();
    }
};
      a.imprime();// ciclano -10\n ciclano -10
```

```
var a = {
    name: "ciclano", idade: -10, imprime() {
        console.log(this.name, this.idade);
        var b = () => {
            console.log(this.name, this.idade);
        }
        b();
    }
};

a.imprime();// ciclano -10\n ciclano -10
```

- O operador Spread (...)

  Permite a expansão de uma expressão em locais onde se esperam vários argumentos.

```
let array1 = [1, 2, 3]
let array2 = [-1, -2, 0, ...array1, 4, 5, 6]
console.log(array2)//[-1,-2,0,1,2,3,4,5,6]
```

- **Tomada de Decisão**
  - if
  - else
  - else if
  - switch

```
if (true) {
    if (10) {
        console.log("apareci");//apareci
        if (0) {
            console.log("não apareci");
        }
        else if ("2") {
            console.log("apareci");//apareci
        }
    }
}
```

```
var x = 3;
switch (x) {
    case 0:
        console.log("passei por aqui");
        break;
    case 1:
        console.log("passei por aqui");
        break;
    default:
        console.log("não passei por nada");
}
```

- **Loops**
  - For
  - For of
  - For in
  - while
  - do ... while
- **Escapes**
  - Break
  - Continue
  - Return

- For of Vs For in

  For in serve para inspecionar propriedades do objeto, sempre fazendo um loop em cima dos nomes das propriedades, enquanto for of é para repetir dados, funcionando apenas em objetos iteráveis.

Exemplos

```
for (var x = 0; x < 2; x++) {
    console.log("iteração n° ", x);
}
```

```
var vetor = ["Ai", "meu", "deus", "o",
"que", "eu", "to", "fazendo", "da",
"minha", "vida"];
for (var value of vetor) {
    console.log(value);
}

var obj = { nome: "cão brabo", atack: 50, defesa: 20,
habilidade: "morder" }
for (var propriedade in obj) {
    console.log(obj[propriedade]);
}
```

```
while ("s" && 0) {
    console.log("que bizarro");
}

do {
    console.log("normal por aqui");
} while (true);
```

- **Funções**
  - Funções nomeadas
  - Funções anônimas
  - Funções seta gorda ou lambda
  - Funções do gerador
- **Parâmetros**
  - Momento: avaliação normal.
  - Direção: tipos primitivos por cópia, enquanto objetos são por cópia de referência.
  - Mecanismo: posicional.

- Função nomeada

```
function MinhaFunção(x: number, y: number): any {
    return (x ** y);
}
```

- Função anônima

```
var anonimo = function () {
    console.log("Mr.Nobody");
}
```

- Função seta gorda

  Usada em funções anônimas, o uso da seta => implica em não utilizar mais a palavra chave function.

```
let foo = () => console.log("função lambda ou seta gorda");
let bar = (a: number, b: number) => a + b;
foo();//função lambda ou seta gorda
console.log(bar(2, 5));//7
```

- Funções geradoras

São funções iteráveis que acumulam valores antigos a partir da palavra chave yield. Os valores são obtidos pela chamada do next().

```
function* pares(init: number) {
    var a = init;
    while (true) {
        a == 0 ? yield a += 2 : yield a *= 2;
    }
}
```

```
var generator = pares(0);
var x = 0;
while (x < 3) {
    console.log(generator.next());
    x++;
}
```

Saídas

{ value: 2, done: false }
{ value: 4, done: false }
{ value: 8, done: false }

- Características

Funções podem sofrer elevação assim como aconteceu com declarações de variáveis.

```
m1();//subiiiiiiiiiindo!


function m1() {
    console.log("subiiiiiiiiiindo!");
}
```

Também podemos trabalhar com parâmetros opcionais, default, e funções que esperam argumentos variados.

```
function option(first = "Default", second: string, third?:
string) {
    console.log(first, second, third);
}
option(undefined, "au au");//Default au au undefined
option("mudei", "miau", "mordi");//mudei miau mordi
```

```
function option(first: string, second?: string, third = "me") {
    console.log(first, second, third);
}
option("help");//help undefined me
option("i need", "somebody", "help");//i need somebody help
```

```
function option(...args) {
    for (var value of args) {
        console.log(value);
    }
}
option("Eu", "quero", "cafe!!")//Eu \nquero\ncafe!!
```

- **Módulos**
  - Arquivo
  - Compilação
  - Importação

- Arquivo utils.ts

Cria-se o arquivo. Todos os elementos que tem que ser visíveis fora do módulo tem que ter a palavra chave export.

```
export function calcArea(a: number, b: number) {
    return a * b;
}
export function calcVolume(a: number, b: number, c: number) {
    return a * b * c;
}
```

- Compilação

Para poder usar o arquivo utils.ts como um módulo é necessário compilar o código de forma a gerar um arquivo módulo.

> “tsc --module amd utils\utils.ts”

isso vai gerar um arquivo utils.js.

- Importação

```
import * as util from "./utils/utils"
console.log(util.calcArea(1, 2));//2
console.log(util.calcVolume(1,2,3));//6

import {calcArea}  from "./utils/utils"
console.log(calcArea(1, 2));//2
console.log(util.calcVolume(1,2,3));//erro
```

- **Namespace**
    - Arquivo
    - Compilação
    - Importação

- Arquivo utils.ts

Cria-se o arquivo de interesse, envolva todos os elementos em um namespace, a regra do export continua valendo.

```
namespace util{
    export function calcArea(a: number, b: number) {
        return a * b;
    }
    export function calcVolume(a: number, b: number, c: number) {
        return a * b * c;
    }
}
```

- Compilação

  >tsc utils.ts --out utils.js

- Importação

```
///<reference path = "utils/utils.ts"/>
console.log(util.calcArea(1, 2));//2
console.log(util.calcVolume(1,2,3));//6
```

- **Namespace Vs Módulo**
  - Módulos são compilados como arquivos módulos e são importados diretamente, arquivos separados.
  - Namespace e qualquer outro arquivo que usam namespace são compilados juntos gerando tudo em um único arquivo.

- **Sistema de tipos**
  - Verificação de tipos
  - Inferência de tipos
  - Asserção de tipos

- Verificação de tipo
  - Verificação focada na “forma” dos dados, “duck typing” / “structural type”

```
class Cachorro{
    som: string = " Au au";
}
class Cobra{
    som: string = "Tssssss";
}
```

```
class Gato{
    som: string = "Miau";
    dominarOMundo() {
        //Ultra secreto
    }
}

let dog: Cachorro = new Cobra();
let cobra: Cobra = new Cachorro();
//let gato:Gato = new Cachorro(); <- Erro

console.log("o cachorro faz: ", dog.som);//Tssssss
console.log("a cobra faz: ", cobra.som);//Au au
```

- Inferência de tipos
  - Toda variável, constante, parâmetro e retorno de função quando não dito o tipo é inferido, não podendo ser alterado ao longo da execução.

```
let a = "booh";
//a = 10; <- Erro
let b;//Tipo any
b = "Hello";
b = true;
b = 10;
```

```
function RetornaAlgo(n: number) {
    if (n > 0) {
        return "Alguma coisa";
    }
    else return 2;
}

let a: number | string = RetornaAlgo(-1);//Okay
let g = RetornaAlgo(2);//Okay
let c: number = RetornaAlgo(-1);//Erro
```

- Asserção de tipos
  - O programador sabe mais que o compilador sobre o tipo trabalhado.
  - Conversão que não realiza verificação ou reestruturação.
  - Não tem impacto no tempo de execução, usado exclusivamente pelo compilador.

```
let str: any = "Uh Mu Bu Gai Fei Di Tal"
var tamanho: number = (<string>str).length;
tamanho = (str as string).length;
//tamanho = (tamanho as string).length; <- Erro
```

- **Polimorfismo**

  Suporta todos os tipos de polimorfismo

  - AD-HOC
    - Coerção
    - Sobrecarga
  - Universal
    - Paramétrico
    - Inclusão

- **AD-HOC**
  - Coerção
    - Em TypeScript o polimorfismo de coerção ocorre somente com o operador “+”.

```
let dog = 1+ " cachorro";
console.log(dog)//1 cachorro
console.log(typeof dog)//string
```

	any	booleano	number	string	outros
any	any	any	any	string	any
booleano	any			string	
number	any		number	string	
string	string	string	string	string	string
outros	any			string	

```
let x :any = true;
let y = x+" doideira D++"
console.log(typeof y);//string
console.log(y);//true doideira D++

let x :any = true;
let y = x+10;
console.log(typeof y);//number
console.log(y);//11
```

```
let x :boolean = true;
let y = x+10;//erro
```

- Sobrecarga
  - Sobrecarga de assinatura de funções, mas somente uma implementação.
  - Pode ser facilmente substituído por any, pelos parâmetros opcionais ou por vários argumentos.
  - Diferenciam pelos tipos e quantidades de parâmetros.
  - Tipo de retorno igual para todas declarações.

```
//Calcula área do quadrado
function calculaArea(base:number);
//Calcula área do triangulo
function  calculaArea(base:number,altura:number);

//Implementação
function  calculaArea(base:number,altura?:number)  {
    if(altura == undefined)return base*base;
    else return base*altura/2;
}
```

- Também conseguimos sobrecarregar construtores

```
class Aluno {
    nome:string;
    matricula:string;
    disciplinasCursadas:Array<string>

    constructor(nome:string,matricula:string);

    constructor(nome:string,matricula:string,disciplin
    as:Array<string>);
```

```
constructor(nome:string,matricula:string,disciplinas?:Array<
string>){
        this.nome=nome;
        this.matricula = matricula;
        if(disciplinas!=undefined){
            this.disciplinasCursadas=disciplinas;
        }
        else this.disciplinasCursadas = [];
    }
}
```

- Não é possível sobrecarregar operadores, somente a LP realiza essas sobrecargas.

```
let somaNumber:number = 10 +30;
let somaString:string = 'D'+'O'+'I'+"DEIRA!";

console.log(somaNumber);//40
console.log(somaString);//DOIDEIRA!
```

- Paramétrico
  - Podemos utilizar o tipo any.

```
function validar(arg:any) {
    return arg;
}
```

  - Não se tem controle dos tipos.

  - TypeScript possui suporte para tipos genéricos.

- O controle agora é garantido...

```
function validar<T>(arg:T) {
    return arg;
}

//Utiliza-se assim
let n:number = validar<number>(123);
```

```
class ClasseGenerica<T>{
    atributo:T;
    constructor(at:T){
        this.atributo = at;
    }
}
```

- **Universal**
  - Inclusão
    - TypeScript é uma linguagem Orientada a Objetos, logo podemos herdar classes.
    - Há herança múltipla apenas em interfaces, pois não há como ter ambiguidade.

```
class Funcionario{
    private nome:string;
    private salario:number;
    constructor(nome:string,salario:number){
        this.nome = nome;
        this.salario = salario;
    }
    seApresentar (){
        return "Oi, meu nome é "+this.nome;
    }
}
```

```
class Estagiario extends Funcionario{
    constructor (nome:string,salario:number){
        super(nome,salario);
    }
    seApresentar():string{
        return super.seApresentar + " e eu sou estagiário.";
    }
}
```

- Amarração tardia??

```
class Produto{
    public price:number
    constructor (preco:number){
        this.price = preco;
    }
     Ehcaro(){
        this.price>1000?console.log("EhCaro")
        :console.log("Barato");
    }
}
```

```
class Livro extends Produto{
    constructor (price:number){
        super(price);
    }
   EhCaro(){
        console.log(this.price)
        this.price > 200?console.log("Ehcaro"):
        console.log("Barato");
    }
}
var b:Produto= new Livro(300);
b.EhCaro();//Ehcaro
```

- **Modificadores de acesso**

Acessível em	public	protected	private
classe	Sim	Sim	Sim
classes filhas	Sim	Sim	Não
classes instanciadas	Sim	Não	Não

- **Interfaces**
  - Dão forma aos dados ou agem como contrato

```
interface LaInterface{
    descricao:string;
}
function printValue(obj:LaInterface) {
    console.log(obj.descricao);
}
let obj = {size:10, descricao:"objeto tamanho 10"};
printValue(obj);//objeto tamanho 10
```

- Classes implementam interfaces

```
class Caixa implements LaInterface{
    descricao:string;
    size:number;
    constructor(descricao:string,size:number){
        this.descricao = descricao;
        this.size=size;
    }
    getDescricao(){
        console.log(this.descricao);
    }
}
```

- Interfaces herdam classes

```
class Caixa {
    size:number;
    constructor(size:number){
        this.size=size;
    }
}
interface LaInterface extends Caixa{
    descricao:string;
}
let b:LaInterface = {descricao :"Caixa pequena",size:1};
```

- **Classes Abstratas**
  - Pode conter detalhes de implementação, por exemplo modificador de acesso.
  - A implementação tem que ser feita pela classe derivada.
  - Para ser uma classe abstrata não precisa necessariamente possuir métodos abstratos, mas se tiver tem que ser uma classe abstrata.

```
abstract class  Animal{
    public abstract fazerSom():void;
    move():void{
        console.log("movendo");
    }
}
abstract class  Animal{
    move():void{
        console.log("movendo");
    }
}
```

- A classe Erro
- Bloco try / catch / finally
- Tipos de erro definidos pela LP

- **Classe Error**
  - Classe base usada para lançamento de exceções.
  - São lançados objetos do tipo Error quando ocorrem erros em tempo de execução.
  - Pode ser usada como objeto base para exceções definidas pelo usuário.

```
//Isso:
const erro1 = Error('Criado por uma chamada de função');
//É o mesmo que isso:
const erro2 = new Error('Criado com a palavra chave new');
//Lançando o erro
throw erro2;

class DivisaoPor0 extends Error{
    constructor(){
        super("Erro: Divisão por 0");
    }
}
```

- É possível lançar praticamente qualquer coisa
- Não é possível declarar quais erros uma função lança
- Não é obrigatório o tratamento quando um erro é lançado

```
function  f1() {
    throw new Error("Erro em f1");
}
function  f2() {
    throw "me lançaaaa";
}
function f3() {
    return 3
}
throw f3();//throw f3(); ^ 3
```

- **Bloco try / catch / finally**
  - Funciona como as linguagens C++ e Java.
  - Diferencial que o tipo lançado não é especificado.
  - Conseguimos capturar um tipo específico usando instanceof

```
function  getNomedoMes(mes:number) {
    enum meses{Janeiro = 1,Fevereiro,Marco,Abril,Maio,Junho,
        Julho,Agosto,Setembro,Outubro,Novembro,Dezembro};
        if (mes>=1 && mes<=12 ){
            return meses[mes];
        }
        else{
            throw Error("Mês invalido!");
        }
}
```

```
try{
    console.log(getNomedoMes(13));
}
catch(e){
    //if(e instanceof Error)
    console.log(e);
}
finally{
    console.log("Sempre serei notado.")
}
```

Saída

```
Error: Mês invalido!
    at getNomedoMes (C:\Users\gabri\OneDrive\Área de
Trabalho\ts\estudo.js:24:15)
    at Object.<anonymous> (C:\Users\gabri\OneDrive\Área de
Trabalho\ts\estudo.js:28:5)
    at Module._compile (internal/modules/cjs/loader.js:956:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:973:10)
    at Module.load (internal/modules/cjs/loader.js:812:32)
    at Function.Module._load (internal/modules/cjs/loader.js:724:14)
    at Function.Module.runMain (internal/modules/cjs/loader.js:1025:10)
    at internal/main/run_main_module.js:17:11
Sempre serei notado.
```

- **Tipos de erros definidos pela LP**
  - InternalError: Lançado quando um erro interno ocorre, como excesso de recursão “too much recursion”
  - RangeError: Quando uma variável ou parâmetro numérico está fora de intervalo válido.
  - ReferenceError: É lançado ao tentar referenciar uma variável que não foi declarada.
  - SyntaxError: Quando ocorre erro de sintaxe.

- TypeError: Lançado quando uma variável ou parâmetro não é de um tipo válido.

- Não há suporte da linguagem para criação de threads. Mas como implementar um paralelismo?
    - WorkerThreads
    - setTimeOut & setInterval
    - Funções Assíncronas
    - Promises

- **Worker Threads**
  - API experimental disponível a partir da versão 10.5.0 do Node.js para simular o comportamento de threads.
  - É orientada a eventos.
  - A partir da versão 11.7 não é necessário usar a flag "--experimental-worker".
  - O compartilhamento de dados entre thread principal e thread “filha” é feito pelo envio de mensagens.
  - Funções, variáveis e status não são compartilhados

```
import * as conc from 'worker_threads';
if (conc.isMainThread) {
    // Código executado pela thread principal
    // Cria um Worker
    const worker = new conc.Worker(__filename);
    //Recebe a mensagem de seu Worker e printa na tela
    worker.on('message', (msg) => { console.log(msg); });
} else {
    // Código executado pelo Worker criado
      // Manda mensagem para thread main
   conc.parentPort!=null?conc.parentPort.postMessage('Hello
world!'):Error("null object");
}
```

- **setTimeOut & setInterval**
  - Não existe paralelismo de fato.
  - Ações são inseridas em uma fila com tempo de espera.
  - Quando o tempo de espera é alcançado tudo é pausado para se esperar a execução da ação agendada.

- O que acontece aqui então?

```
setTimeout(() => console.log("1"), 2000);//1
setTimeout(() => console.log("2"), 0);//2
setInterval(() => console.log("3"), 200);//3
setInterval(() => console.log("4"), 10);//4
```

- **Funções assíncronas**
  - São funções que não seguem o fluxo normal síncrono. Ou seja em sua execução o programa pode optar por tomar outras ações.
  - Possibilidade de erros, como funções que precisam do valor de uma função assíncrona serem terminadas antes do valor ser retornado pela função assíncrona.
  - Funções assíncronas retornam promises.

- **Promises**
  - Possui 5 estados, Pendente, Resolvida, Rejeitada, Realizada e Estabelecida.
  - Como o nome diz, é uma promessa de uma ação, se essa ação for bem sucedida, teremos uma promise resolvida e poderemos usufruir de seu resultado.
  - Promises.all

```
async function corrida(x: number) {
      setTimeout(()=>console.log(x," está
correndo..."),Math.random()*x);
       return "terminou "+x;
    }
    var p = Promise.all([corrida(1), corrida(2),
corrida(3),corrida(4),corrida(5)]);
    console.log(p);
    setTimeout(()=>console.log(p),1000);
```

Saída

```
Promise {<pending>}
1 está correndo...
2 está correndo...
3 está correndo...
4 está correndo...
5 está correndo...

Promise {
 [
      'terminou 1',
      'terminou 2',
      'terminou 3',
      'terminou 4',
      'terminou 5'
 ]
}
```

- Comparação entre TypeScript, C, C++ e Java

Critérios gerais	TypeScript	C	C++	Java
Aplicabilidade	Parcial	Sim	Sim	Parcial
Confiabilidade	Não	Não	Não	Sim
Aprendizado	Não	Sim	Não	Sim
Eficiência	Não	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de projeto	OO e funcional	Estruturado	Estruturado e OO	OO

Critérios gerais	TypeScript	C	C++	Java
Evolutibilidade	Parcial	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Parcial	Sim	Sim	Parcial
Custo	Depende	Depende	Depende	Depende

Critérios Específicos	TypeScript	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e comandos	Sim	Sim	Sim	Sim
Tipos primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de memória	Sistema	Programador	Programador	Sistema

Critérios específicos	TypeScript	C	C++	Java
Persistência dos dados	Biblioteca de classes e funções, serialização.	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização.
Passagem de parâmetros	Lista variável, por valor e por cópia de referência.	Lista variável e por valor.	Lista variável, por valor e por referência.	Lista variável, por valor e por cópia de referência.

Critérios específicos	TypeScript	C	C++	Java
Encapsulamento e proteção	Sim	Parcial	Sim	Sim
Sistema de tipos	Sim	Não	Parcial	Sim
Polimorfismo	Todos	Coerção e sobrecarga	Todos	Todos
Exceções	Parcial	Não	Parcial	Sim
Concorrência	Não	Não	Não	Sim