

TypeScript

Gustavo Marino Botta

TypeScript

- JavaScript com definição de tipos
- Permite que os desenvolvedores adicionem tipos
- Usa verificação de tipo em tempo de compilação, mas não durante a execução
- É transpilado para JavaScript usando um compilador
- Roda em qualquer lugar onde o JavaScript roda

Comando para instalação do typescript

- Baixar e instalar o nodejs: <https://nodejs.org>
- `node -v`
- `npm install -g typescript`
- `tsc -v`
- `tsc --init`
 - Cria o arquivo inicial `tsconfig.json`
- `tsconfig.json` → acrescente:
 - `"sourceMap": true, // usado pelo depurador do VSC`
 - `"outDir": "./js"`

Erro na instalação:


- Abrir PowerShell como administrador
- Rodar este comando:
`Set-ExecutionPolicy RemoteSigned`

Ou usar o prompt do DOS que não tem este bloqueio de script.

-g: Significa que a instalação será global, ou seja, o pacote ficará disponível para todos os projetos no seu sistema, não apenas para o projeto atual.

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es2016",  
    "module": "commonjs",  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "skipLibCheck": true,  
    "sourceMap": true, // usado pelo depurador do VSC  
    "outDir": "./js"  ←  
  }  
}
```



Para compilar os arquivos .ts para .js na pasta ./js
use o comando no terminal:
tsc

Depurador no Visual Studio Code

- No VSC, selecione o arquivo .ts
- Menu Executar - Adicionar Configuração - Node.js
- Será criado o arquivo launch.json na pasta .vscode

No arquivo launch.json dos códigos fontes das demonstrações, comente esta linha:

```
// "program": "${workspaceFolder}\\index.ts"
```

E descomente a linha do arquivo que deseja depurar, exemplo:

```
"program": "${workspaceFolder}\\DemoTS01.ts"
```

.vscode/launch.json

Comente esta linha

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Iniciar o Programa",
      "skipFiles": [
        "<node_internals>/**"
      ],
      "program": "${workspaceFolder}\\DemoTS01.ts",
      // "preLaunchTask": "tsc: build - tsconfig.json",
      "outFiles": [
        "${workspaceFolder}/js/**/*.js"
      ]
    }
  ]
}
```

No arquivo launch.json dos códigos fontes das demonstrações, comente esta linha:
// "program": "\${workspaceFolder}\\index.ts"

E descomente a linha do arquivo que deseja depurar, exemplo:
"program": "\${workspaceFolder}\\DemoTS01.ts"

Depurador no Visual Studio Code

EXECUTAR E DEPURAR

Iniciar o Programa

launch.json

TS Demo01.ts

TS Demo01.ts > ...

```
1 let nome: string = "Gustavo";
2 console.log(nome);
3 console.log("Fim");
4
5 function fatorial() {
6     let numerol: number = 0;
7     let id = document.getElementById("numerol");
8     let p1 = document.getElementById("p1");
9
10    if (id)
11        numerol = parseInt(id.innerText);
12    let fatorial = 1;
13    for (let x = numerol; x >= 1; x--) {
14        fatorial = fatorial * x;
15    }
16    alert(fatorial);
17    console.log(fatorial);
18    if (p1)
19        p1.innerHTML = String(fatorial);
20 }
```

VARIÁVEIS

Local

__dirname: 'D:\\Meus Documentos\\PortaArquivo\\TypeScri...

__filename: 'D:\\Meus Documentos\\PortaArquivo\\TypeScri...

> exports: {}

> module: Module {id: '.', path: 'D:\\Meus Documentos\\Por...

INSPEÇÃO

nome: 'Gustavo'

PILHA DE CHAMADAS

Iniciar o Programa: Demo01.js [6760]

<anonymous> Demo01.ts 3:1

Mostrar Mais 6: Ignorado por skipFiles

SCRIPTS CARREGADOS

PONTOS DE PARADA

PROBLEMAS SAÍDA TERMINAL PORTAS CONSOLE DE DEPURAÇÃO

C:\\Program Files\\nodejs\\node.exe ...

Gustavo

Filtrar (por exemplo, text, lexclude, \\escape)

Demo01.ts:2

Acompanhando os resultados. Use F9 para marcar pontos de paradas e F10 para depurar linha a linha.

Tipos Simples

```
let nome1: string = "Gustavo";  
let nome2 = "Gustavo";  
let idade: number = 0;  
let brasileiro: boolean = true;
```

Definindo o tipo ou deixando para o TS definir automaticamente

```
nome1 = 33;  
nome2 = 33;  
idade = "abc";  
brasileiro = 0;
```

O TS irá acusar o erro em tempo de compilação

```
let teste: any = true;  
teste = "string";  
teste = 123;  
teste = false;
```

Evite o uso do tipo Any

Quando o TS não consegue inferir o tipo de uma variável automaticamente, ele definirá o tipo para any. Any desabilita a verificação de tipo e permite que todos os tipos sejam usados.

Tipos Simples

```
let nome1: string = "Gustavo";  
let nome2 = "Gustavo";  
let idade: number = 0;  
let brasileiro: boolean = true;
```

```
nome1 = 33;  
nome2 = 33;  
idade = "abc";  
brasileiro = 0;
```

```
let teste: any = true;  
teste = "string";  
teste = 123;  
teste = false;
```

Use o comando tsc para gerar os arquivos js (javascript) de todos os arquivos ts (typescript).

Antes de depurar sempre usar o comando tsc para garantir que js está com o mapeamento atualizado com o ts. Caso contrário a depuração pode mostrar inconsistência, pois ele se baseia no arquivo js, mas com visualização no ts.

Demonstrar falha na depuração:
nome1 = "Teste de Depuração";
console.log(nome1);

Matrizes

```
const nomes: string[] = ["Gustavo", "Fernando"];  
nomes.push("Gustavo");  
nomes.push(3);
```

TS irá acusar erro: tipo número tentando em tipo string.

```
const vetorNumeros = [1, 2, 3];  
vetorNumeros.push(4);  
vetorNumeros.push("abc");  
let variavel1: number = vetorNumeros[0];
```

TS definirá o tipo do vetor automaticamente para number.

TS irá acusar erro: tipo string tentando em tipo número.

Tuplas

- Matriz com comprimento e tipos predefinidos para cada índice
- A ordem deve ser seguida

```
let tupla1: [number, boolean, string];  
tupla1 = [5, false, 'abc'];  
tupla1 = [false, 'abc', 5];
```

Como os tipos não estão na ordem definidas na tupla o TS acusará erro.

Tuplas nomeadas

- Tuplas nomeadas fornecem mais contexto sobre o que nossos valores de índice representam

```
const grafico1: [x: number, y: number] = [10, 20];  
grafico1[0] = 5;  
grafico1[1] = 15;
```

Na codificação trás
lembrete que facilita
associar o contexto

```
const grafico1: [x: number, y: number] = [10, 20];  
grafico1[0] = 5;  
  
const grafico1: [x: number, y: number]  
grafico1[1] = 15;
```

Desestruturando tuplas

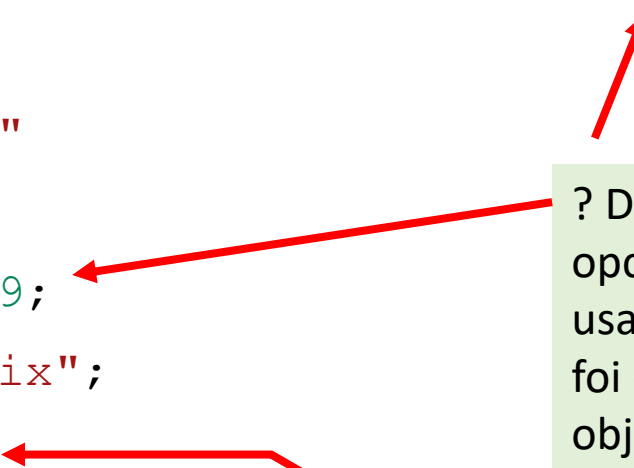
Aqui ocorre a desestruturação.

```
const grafico2: [number, number] = [20, 10];  
let [x, y] = grafico2;  
console.log("X da desestruturação: " + x);  
console.log("Y da desestruturação: " + y);
```

X da desestruturação: 20
Y da desestruturação: 10

Tipos em objetos

```
const carro: { modelo: string, avaliacao?: number } =  
{  
  modelo: "Corolla"  
};  
carro.avaliacao = 9;  
carro.modelo = "Onix";  
carro.modelo = 2;
```



? Define como propriedade opcional, por isto pode ser usado posteriormente e não foi obrigatório na criação do objeto.

TS irá acusar erro: tipo número tentando em tipo string.


Assinaturas de índice

O mesmo que
`Record<string, number>.`



```
const mapaNomeIdade: { [index: string]: number } =  
{  
  Gustavo : 51  
};
```

```
mapaNomeIdade.Fernando = 53;  
mapaNomeIdade.Aline = "Dezenove";  
console.log(mapaNomeIdade);  
console.log(mapaNomeIdade.Gustavo);  
console.log(mapaNomeIdade.Fernando);
```



TS irá acusar erro: tipo
string tentando em tipo
número.

```
> {Gustavo: 51, Fernando: 53}  
51  
53
```

Enumeração numérica

```
enum pontosCardeais {  
    Norte = 1,  
    Leste,  
    Sul,  
    Oeste  
}  
  
let currentDirection = pontosCardeais.Norte;  
console.log(currentDirection);  
currentDirection = 2;  
console.log(currentDirection);  
currentDirection = "Norte";
```


Por padrão inicia com 0, mas pode ser inicializado manualmente. Os demais são acrescidos de 1.

TS irá acusar erro: tipo string tentando em tipo pontosCardeais. Neste caso enumeração numérica.

Enumeração numérica - Totalmente Inicializados

```
enum codigoEstado {  
    naoEncontrado = 404,  
    Sucesso = 200,  
}
```

Numeração definida em todos.



```
console.log(codigoEstado.naoEncontrado);  
console.log(codigoEstado.Sucesso);
```

Enumeração de strings

```
enum pontosCardeais2 {  
    Norte = "Norte",  
    Leste = "Leste",  
    Sul = "Sul",  
    Oeste = "Oeste"  
};  
  
console.log(pontosCardeais2.Norte);  
console.log(pontosCardeais2.Oeste);
```

Alias (apelido) para tipos

```
type AnoCarro = number  
type ModeloCarro = string  
type Carro = {  
  ano: AnoCarro,  
  modelo: ModeloCarro  
}
```

Definindo apelidos para tipos primitivos.

Definindo apelido para objeto.

Utilizando os apelidos dos tipos primitivos.

```
const anoCarro: AnoCarro = 2024  
const modeloCarro: ModeloCarro = "Corolla"  
const carro1: Carro = {  
  ano: anoCarro,  
  modelo: modeloCarro  
};  
console.log(carro1);
```

Utilizando o apelido do objeto.

Utilizando os valores das variáveis definidas acima.

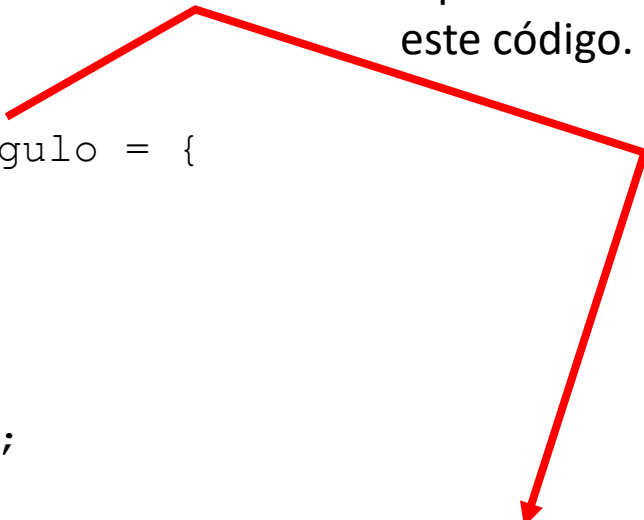
```
> {ano: 2024, modelo: 'Corolla'}
```

Interfaces (apelido) para os tipos do objeto

```
interface Retangulo {  
  altura: number,  
  largura: number  
}
```

```
const retangulo: Retangulo = {  
  altura: 20,  
  largura: 10  
};  
console.log(retangulo);
```

Equivale a
este código.



```
const retangulo: {altura: number, largura: number} =  
{  
  altura: 20,  
  largura: 10  
};
```

Estendendo Interfaces

Estende a interface Retangulo com mais atributos, neste caso a cor.

```
interface Retangulo {  
  altura: number,  
  largura: number  
}
```

```
interface RetanguloColorido extends Retangulo {  
  cor: string  
}
```

```
const retanguloColorido: RetanguloColorido = {  
  altura: 20,  
  largura: 10,  
  cor: "vermelho"  
};
```

```
console.log(retanguloColorido);
```

```
> {altura: 20, largura: 10, cor: 'vermelho'}
```

União | (OU)

Pode ser do tipo string ou number.

- União - sinal | (pipe) - equivale a OU

```
function imprimeCodigo(code: string | number) {  
    console.log(`Código ${code}.`)  
    console.log(`Código ${code.toUpperCase()}.`)  
}  
  
imprimeCodigo(404);  
imprimeCodigo('404');
```

TS irá acusar erro: tipo number não possui esta propriedade.

2 Código 404.

Funções – retorno e parâmetros

```
function pegaHora(): number {  
    return new Date().getTime();  
}
```

Definindo o tipo do retorno.

```
function imprimeOi(): void {  
    console.log('Oi!');  
}
```

```
function multiplica(a: number, b: number) {  
    return a * b;  
}
```

Definindo os tipos dos parâmetros. Aqui não foi especificado o tipo de retorno, então será definido pelo TS de forma implícita.

```
function multiplica(a: number, b: number): number
```

```
console.log(pegaHora());  
imprimeOi();  
console.log(multiplica(2, 5))
```

```
1718198811158  
Oi!  
10
```

Funções – parâmetro opcional, padrão e resto

Definindo o valor padrão para b.
Se não for informado será usado
b = 10.

```
function soma(a: number, b: number = 10, c?: number, ...rest: number[])  
{  
    return a + b + (c || 0) + rest.reduce((p, c) => p + c, 0);  
}  
  
console.log(soma(2, 5))  
console.log(soma(2))  
console.log(soma(2, 5, 3))  
console.log(soma(2, 5, 3, 1, 1, 1, 1, 1))
```

? Torna o parâmetro opcional,
então ele pode ser informado ou
não.
Não sendo informado será usado
o valor = 0.

7
12
10
15

Funções – parâmetro opcional, padrão e resto

```
function soma(a: number, b: number = 10, c?: number, ...rest: number[])  
{  
    return a + b + (c || 0) + rest.reduce((p, c) => p + c, 0);  
}
```

console.log(soma(2, 5))

console.log(soma(2))

console.log(soma(2, 5, 3))

console.log(soma(2, 5, 3, 1, 1, 1, 1, 1))

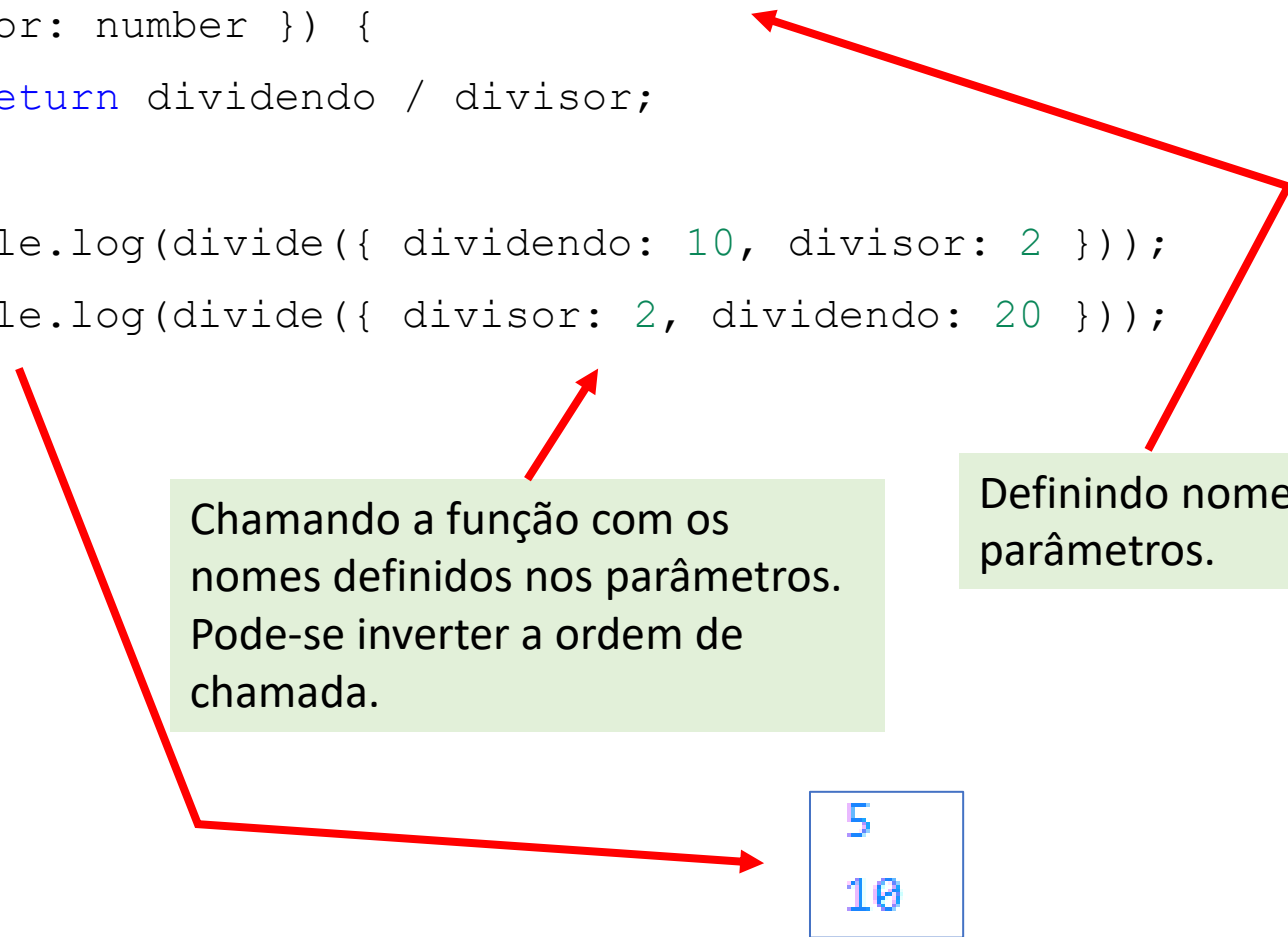
Argumentos restantes são sempre uma matriz.

O opcional da variável c poderia ser retirado visto que o restante iria tratá-lo. Foi deixado no slide somente para exemplificar num único código.

7
12
10
15

Função - parâmetros nomeados

```
function divide({ dividendo, divisor }: { dividendo: number,  
divisor: number }) {  
    return dividendo / divisor;  
}  
  
console.log(divide({ dividendo: 10, divisor: 2 }));  
console.log(divide({ divisor: 2, dividendo: 20 }));
```



Chamando a função com os nomes definidos nos parâmetros. Pode-se inverter a ordem de chamada.

Definindo nomes nos parâmetros.

5
10

Função – tipo alias

Definindo um tipo para a função, usa-se notação de funções de seta.

O retorno da função será do tipo number.

```
type TipoNumero = (value: number) => number;  
const funcaoTrocaSinal: TipoNumero = (value) => value * -1;  
console.log(funcaoTrocaSinal(10));  
console.log(funcaoTrocaSinal(-10));
```

Usando o tipo da função.

O parâmetro será do tipo number.

```
type TipoNumero = (value: number) => number
```

-10
10

Casting – substituindo um tipo

Convertendo o tipo com `as`.

```
let variavel2: unknown = 'teste';  
console.log((variavel2 as string).length);  
console.log(<string>variavel2.length);
```

Convertendo o tipo com `<>`.

```
let variavel3 = 'teste';  
console.log(((variavel3 as unknown) as number).length);
```

Para evitar erros,
primeiro converta para
`unknown` e depois para
o tipo de destino.

TS irá acusar erro:
A propriedade 'length' não
existe no tipo 'number'.

5
5

Classes

Public - (padrão) permite acesso de qualquer lugar

Private - permite acesso apenas dentro da classe

nome não pode ser alterado após sua definição inicial.

```
class Pessoa {  
  public constructor(private readonly nome: string) {  
  }  
  public pegaNome(): string {  
    return this.nome;  
  }  
  public mudaNome(nome: string): void {  
    this.nome = nome;  
  }  
}
```

```
const pessoa = new Pessoa("Gustavo");  
console.log(pessoa.pegaNome());  
console.log(pessoa.nome);
```

Equivale ao código:

```
private readonly nome: string;  
public constructor(nome: string){  
  this.nome = nome;  
}
```

TS irá acusar erro:
Não é possível atribuir a 'nome' porque é uma propriedade de somente leitura.

TS irá acusar erro:
A propriedade 'nome' é privada e somente é acessível na classe 'Pessoa'

Classes - Heranças

Interface

Classe Abstrata

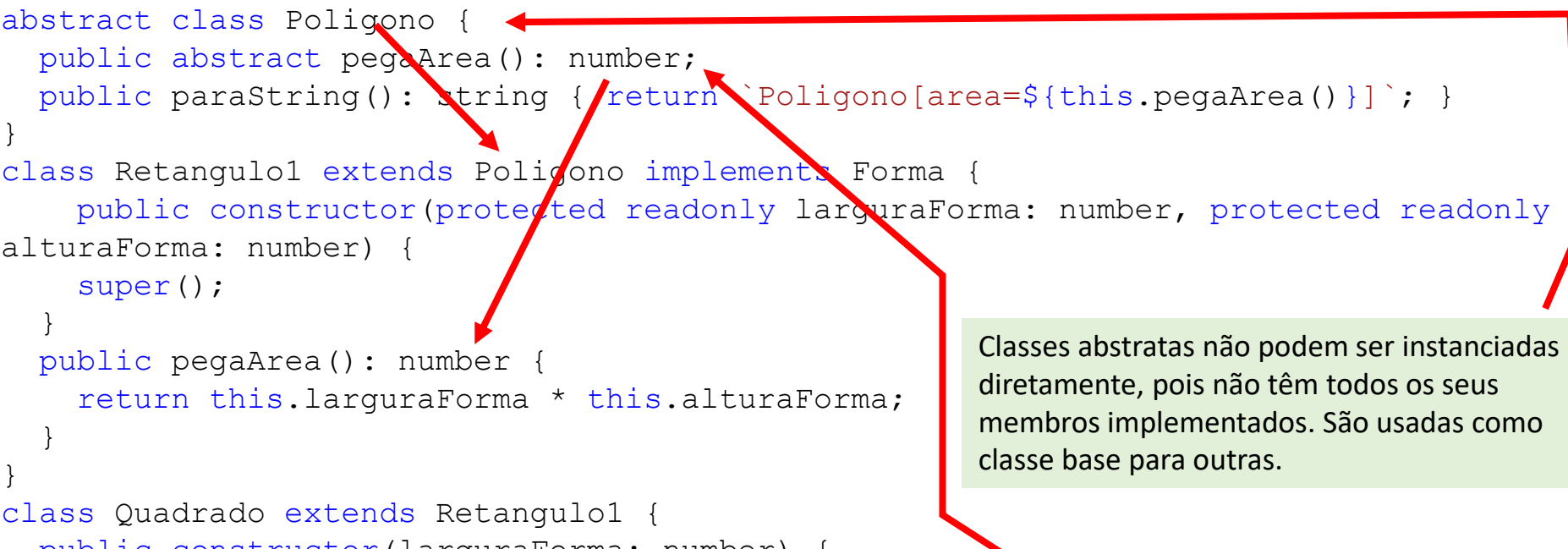
Estender

Sobrepor

```
interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());
```

Interfaces podem ser usadas para definir o tipo de uma classe. Neste caso está definindo que o retorno da função deve ser do tipo number.

```
interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());
```



Classes abstratas não podem ser instanciadas diretamente, pois não têm todos os seus membros implementados. São usadas como classe base para outras.

Esta função abstrata também está definindo que a função deve retorna number.

A interface Forma foi deixado no slide somente para efeitos didáticos, pois ela está fazendo a mesma definição desta função abstrata:
public abstract pegaArea(): number;


```

interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());

```

Protected – a propriedade é visível para a classe em si e seus filhos.

Extends - uma classe só pode estender uma outra classe.
A classe filha herda todos as propriedades da classe pai.

```

interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());

```

```
interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());
```

```

interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());

```

Override (sobrepôr) - Quando uma classe estende outra classe, ela pode substituir os membros da classe pai com o mesmo nome. Neste exemplo a função paraString substitui o paraString do Poligono

Como Retangulo1 não usou sobreposição, será usado paraString() do Poligono (classe pai).

Como Quadrado usou sobreposição, será usado paraString() do Quadrado e não da classe pai.

```

200
Poligono[area=200]
400
Quadrado[largura=20]

```

```

interface Forma { pegaArea: () => number; }
abstract class Poligono {
  public abstract pegaArea(): number;
  public paraString(): string { return `Poligono[area=${this.pegaArea()}]`; }
}
class Retangulo1 extends Poligono implements Forma {
  public constructor(protected readonly larguraForma: number, protected readonly
alturaForma: number) {
    super();
  }
  public pegaArea(): number {
    return this.larguraForma * this.alturaForma;
  }
}
class Quadrado extends Retangulo1 {
  public constructor(larguraForma: number) {
    super(larguraForma, larguraForma);
  }
  public override paraString(): string {
    return `Quadrado[largura=${this.larguraForma}]`;
  }
}
const meuRetangulo = new Retangulo1(10, 20);
console.log(meuRetangulo.pegaArea());
console.log(meuRetangulo.paraString());
const meuQuadrado = new Quadrado(20);
console.log(meuQuadrado.pegaArea());
console.log(meuQuadrado.paraString());

```

Tipos de utilitários

Partial altera todas as propriedades de um objeto para serem opcionais. Neste caso não exigiu RACA.

```
interface Animal {  
  nome: string;  
  raca: string;  
  idade?: number;  
}  
  
let meuAnimal1: Partial<Animal> = {};  
meuAnimal1.nome = "Meg";  
console.log(meuAnimal1);  
  
let meuAnimal2: Required<Animal> = {  
  nome: 'Pepe',  
  raca: 'Yorkshire',  
  idade: 7  
};  
console.log(meuAnimal2);
```

```
> {nome: 'Meg'}  
> {nome: 'Pepe', raca: 'Yorkshire', idade: 7}
```

```
const meuAnimal3: Omit<Animal, 'raca' | 'idade'> = {  
  nome: 'Duda',  
  idade: 7  
};  
console.log(meuAnimal3);  
  
const meuAnimal4: Pick<Animal, 'nome'> = {  
  nome: 'Bob',  
  raca: 'Yorkshire',  
  idade: 7  
};  
console.log(meuAnimal4);  
  
const meuAnimal5: Readonly<Animal> = {  
  nome: 'Thor',  
  raca: 'Rottweiler',  
};  
meuAnimal5.nome = 'Gigante';  
console.log(meuAnimal5);
```

Required altera todas as propriedades em um objeto para serem necessárias. Neste caso obriga IDADE que era opcional.

Tipos de utilitários

```
interface Animal {
  nome: string;
  raca: string;
  idade?: number;
}

let meuAnimal1: Partial<Animal> = {};
meuAnimal1.nome = "Meg";
console.log(meuAnimal1);

let meuAnimal2: Required<Animal> = {
  nome: 'Pepe',
  raca: 'Yorkshire',
  idade: 7
};
console.log(meuAnimal2);
```

Pick remove todas, exceto as chaves especificadas, de um tipo de objeto. Neste caso ficará somente NOME.

Readonly é somente leitura, não podem ser modificadas depois de atribuído um valor.

Omit remove chaves de um tipo de objeto. Neste caso remove RACA e IDADE.

```
const meuAnimal3: Omit<Animal, 'raca' | 'idade'> = {
  nome: 'Duda',
  idade: 7
};
console.log(meuAnimal3);
```

TS irá acusar erro.

```
const meuAnimal4: Pick<Animal, 'nome'> = {
  nome: 'Bob',
  raca: 'Yorkshire',
  idade: 7
};
console.log(meuAnimal4);
```

TS irá acusar erro.

```
const meuAnimal5: Readonly<Animal> = {
  nome: 'Thor',
  raca: 'Rottweiler',
};
meuAnimal5.nome = 'Gigante';
console.log(meuAnimal5);
```

TS irá acusar erro.

Tipos de utilitários

Exclude remove tipos de uma união.
Neste caso ficarão NUMBER e BOOLEAN.

```
type Basico = string | number | boolean;  
let valor: Excluído<Basico, string> = true;  
valor = 8;  
valor = "abc";  
console.log(typeof valor);
```

TS irá acusar erro.

```
const mapa: Record<string, number> = {  
  'Alice': 21,  
  'Bob': 25  
};  
console.log(mapa);
```

Record define um tipo de objeto com um tipo de chave e tipo de valor.
Record<string, number> é equivalente a { [key: string]: number }

> {Alice: 21, Bob: 25}

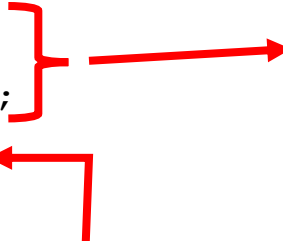
keyof

```
interface Crianca {  
  nome: string;  
  idade: number;  
}  
  
function imprimeCrianca(crianca: Crianca, campo: keyof Crianca) {  
  console.log(`Imprimindo criança.....: ${campo}: "${crianca[campo]}"`);  
}  
  
let crianca = {  
  nome: "Pedrinho",  
  idade: 10  
};  
  
imprimeCrianca(crianca, "nome");  
imprimeCrianca(crianca, "idade");  
imprimeCrianca(crianca, "pai");
```

keyof extrai o nome da chave de um objeto e cria uma união com essas chaves.
Neste caso será NOME | IDADE.



Imprimindo criança.....: nome: "Pedrinho"
Imprimindo criança.....: idade: "10"



TS irá acusar erro, pois não existe chave com este título.

Genéricos

Genéricos – São variáveis do tipo e facilitam a escrita de código reutilizável. Neste exemplo o S e T são os genéricos.

```
function criaPar<S, T>(v1: S, v2: T): [S, T] {  
  return [v1, v2];  
}  
  
console.log(criaPar<string, number>('Gustavo', 51));  
console.log(criaPar<string, string>('Gustavo', 'Celma'));
```

```
> (2) ['Gustavo', 51]  
> (2) ['Gustavo', 'Celma']
```

O S e T irão receber estes tipos e irão repassar para os próximos S e T respectivos.