

Programação para Dispositivos Móveis

TypeScript



Prof. Davi Taveira Alencar Alarcão

davitaveira@yahoo.com.br

<http://sites.google.com/site/davitaveira/>

Conceitos iniciais

TypeScript

■ O que é TypeScript?

- ❑ **TypeScript** é um superset para a linguagem JavaScript
- ❑ Adiciona funções ao JavaScript como a declaração de tipos de variável
- ❑ Pode ser utilizado com frameworks/libs como Express, React e Ionic
- ❑ Precisa ser compilado em JavaScript, ou seja, não executamos TS
- ❑ Desenvolvido e mantido pela Microsoft

TypeScript

■ O que é TypeScript?

- ❑ Adiciona confiabilidade ao programa (tipos)
- ❑ Provê novas funcionalidades a JS (como interfaces)
- ❑ Com TS podemos verificar os erros antes da execução do código, ou seja, no desenvolvimento.
- ❑ Deixa o JS mais explícito, diminuindo a quantidade de bugs
- ❑ Por estes e outros motivos perdemos menos tempo com debug

TypeScript

■ O que é TypeScript?

- Os arquivos de código TypeScript usam a extensão .ts e são **transpilados** diretamente para o código JavaScript.
- **Transpilar** é semelhante a compilar, mas ao invés de gerar um código de mais baixo nível, é gerado um código de mesmo nível em outra linguagem, no caso do TypeScript (o código JavaScript é transpilado para o código TypeScript).

Preparando o ambiente

TypeScript

■ Instalações

- ❑ Node js
- ❑ Visual studio code
- ❑ Typescript

TypeScript



- Node js

- <https://nodejs.org/en/download/>
- Verificar instalação no cmd
 - Node -v
 - Npm -v

TypeScript

- Visual studio code

- <https://code.visualstudio.com/download>

Dicas

Os atalhos para indentar código no VS Code são:



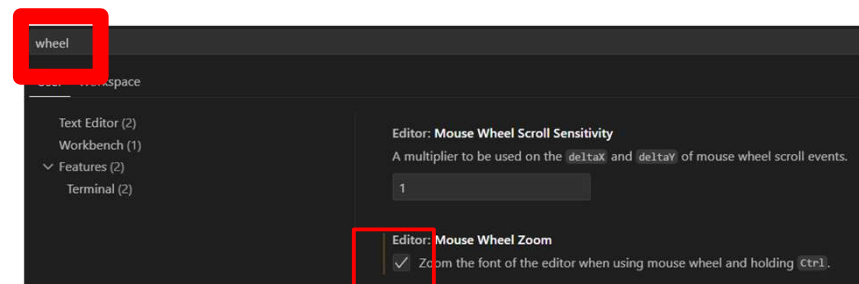
Windows: `shift + alt + f`



Ativar scroll para zoom



>



TypeScript

- Visual studio code

- <https://code.visualstudio.com/download>

Dicas

Ativar scroll para zoom no terminal

The terminal can now be zoomed in and out with the mouse wheel while holding **CTRL**.

To enable this feature, press down **CTRL** + **,** (comma) and search for "terminal mouse wheel zoom".

Tick the checkbox to enable the **Terminal > Integrated: Mouse Wheel Zoom** feature.

TypeScript

■ Instalando o TypeScript

- ❑ Para instalar o TS vamos utilizar o **npm**
- ❑ O nome do pacote é **typescript**
- ❑ Vamos adicionar de forma global com a flag **-g**
- ❑ A partir da instalação temos como executar/compilar TS em qualquer local da nossa máquina por meio do comando **tsc**

TypeScript

- Instalando o TypeScript

- Abrir um terminal na pasta e instalar o TYPESCRIPT

```
npm install -g typescript
```

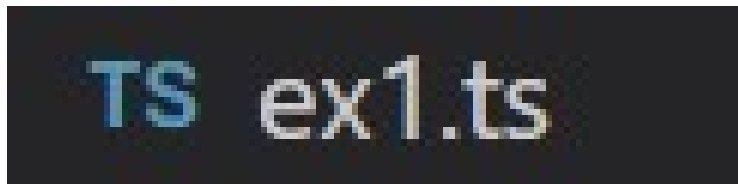
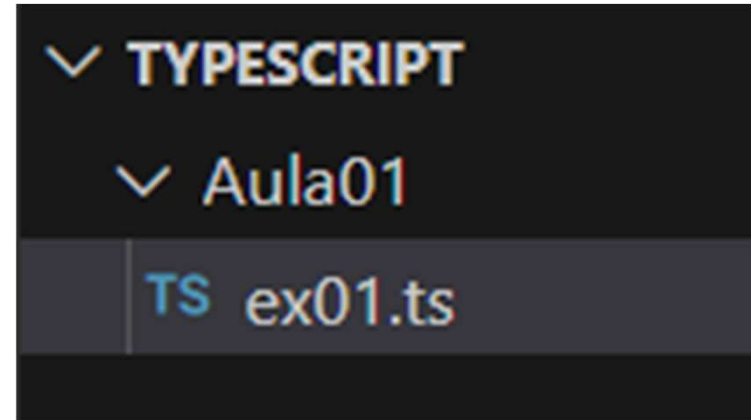
- Verificar a versão instalada

```
tsc --v
```

Criando arquivo TS

TypeScript

- Criando primeiro arquivo TS
 - ❑ Criar a pasta **TYPESCRIPT**
 - ❑ Abrir a pasta criada no VSCode
 - ❑ Criar o primeiro arquivo do tipo .TS



TypeScript

- Etapas para executar um arquivo TS

- Transpilar arquivo TS para JS

```
tsc ex1.ts
```

- Arquivo JS deverá ser criado na pasta

```
JS ex1.js
```

- Executar arquivo JS

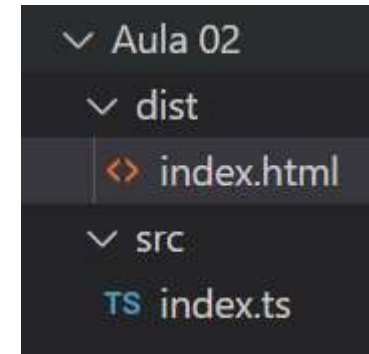
```
node ex1.js
```

Organizando melhor o projeto

TypeScript

■ Criando estrutura de pastas

- ❑ Criar a pasta **TYPESCRIPT**
- ❑ Abrir a pasta criada no VSCode
- ❑ Criar a seguinte estrutura de subpastas/arquivos
 - **dist**: pasta de deploy/build final/distribuição (para hospedagem)
 - ❑ index.html: será aberto no navegador
 - **src**: pasta de códigos de desenvolvimento
 - ❑ index.ts: arquivo com código TS



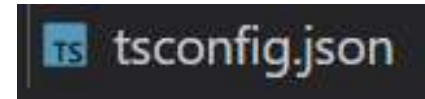
TypeScript

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Curso de TypeScript</title>
  <script src="js/index.js" defer></script>
</head>
<body>
  <h1>Hello TS</h1>
</body>
</html>
```



Carrega primeiro o html e depois o JS

TypeScript



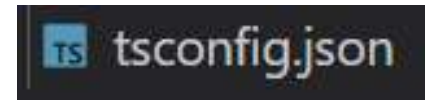
- Gerar arquivo de configuração do TS

- Dentro da pasta do projeto, digitar o comando

```
tsc --init
```

- Arquivo com as configurações de compilação do TS para JS

TypeScript



■ Ajustando o arquivo de configuração do TS

- ❑ Alterar o diretório onde se encontra no arquivo TS

```
"rootDir": "./src/",
```

- ❑ Alterar o diretório onde o arquivo JS será criado

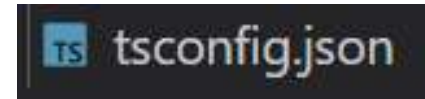
```
"outDir": "./dist/js/",
```

- ❑ Após os ajustes, compilar o arquivo de configuração utilizando o comando abaixo

```
tsc
```

- ❑ O arquivo **index.js** será gerado dentro da pasta **/dist/js/**

TypeScript



■ Compilação automática do arquivo TS

- Dentro da pasta do projeto, digitar o comando

```
tsc -w
```

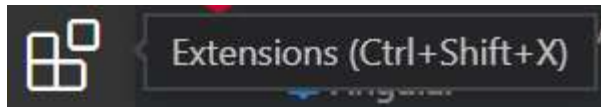
- Toda alteração no arquivo TS será refletida, automaticamente, no arquivo JS

TypeScript

<> index.html

■ Visualizando o index.html com o LIVE SERVER

- Para instalar a extensão LIVE SERVER, clique em extensions



- Procure por LIVE SERVER e instale

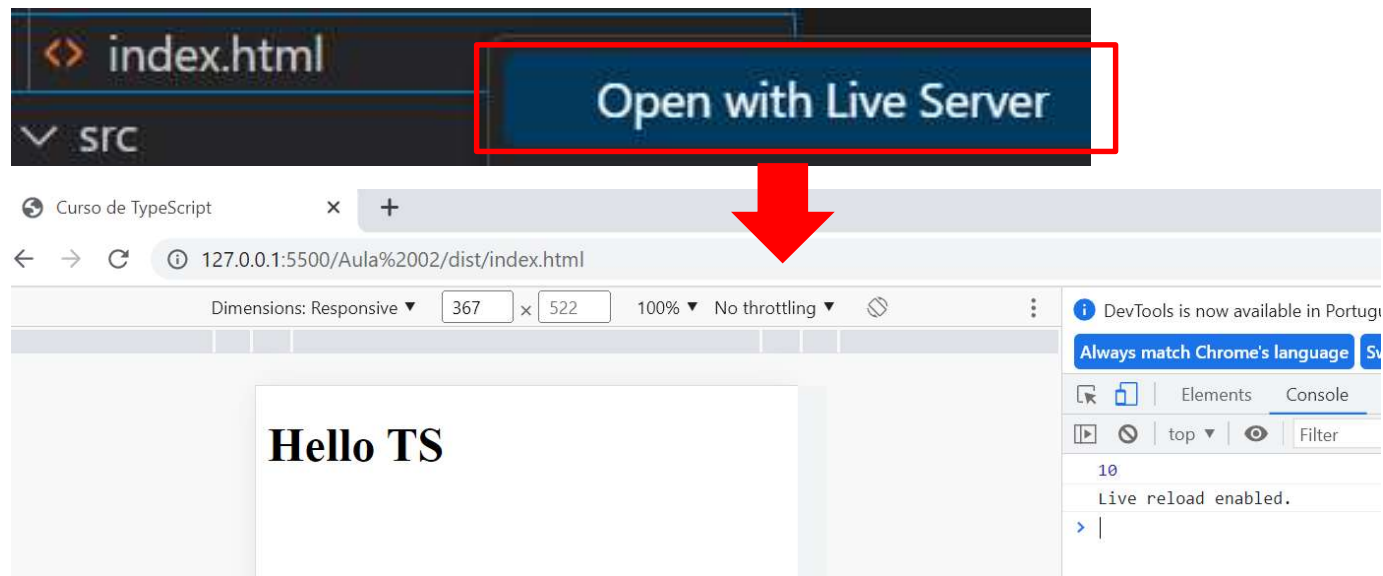


TypeScript

<> index.html

■ Utilizando o LIVE SERVER

- ❑ Clicar, com o botão direito do mouse, em cima do arquivo .html e selecionar “Open with Live Server”



Observações

TypeScript

- Depuração de erros

- TS auxiliando durante o desenvolvimento do código

```
let x:number= 25;  
  
x = "teste";  
  
console.log(x);
```

```
src/index.ts:3:1 - error TS2322: Type 'string' is not assignable to type 'number'.  
3 x = "teste";  
  ~  
  
[00:01:34] Found 1 error. Watching for file changes.
```

TypeScript

■ Inferência x annotation

□ Inferência

```
//1)inferência  
let y = 12  
y = 'teste'
```

Por inferência, a variável y é do tipo number

□ Annotation

```
//2)Annotation  
let z : number = 10  
z = 'teste'
```

Tipos de dados

TypeScript

■ Tipos de Dados

- O TypeScript suporta os mesmos tipos de dados que o JavaScript:
 - Booleano (**boolean**): é um tipo de dado que permite os valores verdadeiro (true) ou falso (false)
 - Número (**number**): é um tipo de dados que suporta qualquer número, seja ele natural, inteiro ou racional
 - **String**: é qualquer valor entre aspas simples, aspas duplas ou crases
 - **Array**: permite armazenar vários valores ou elementos em uma única variável.
 - **Any**: permite qualquer tipo de dado. É usado principalmente com bibliotecas de terceiros, quando não se sabe ao certo o tipo da variável. Seu uso deve ser evitado.

TypeScript

■ Tipos de Dados

- **Void**: é um tipo de dado vazio, ou seja, é na realidade, a ausência de tipo. Normalmente é utilizado para tipar uma função quando a mesma não possui nenhum retorno

TypeScript

- Sintaxe

```
//Tipos básicos
let firstName: string = "Matheus";
let age: number = 30;
const isAdmin: boolean = true;

console.log(firstName);
console.log(typeof firstName);
console.log(age);
console.log(isAdmin);
```

TypeScript

- Sintaxe any

- ❑ **Evite utilizar (somente em casos específicos)**

```
// any
let a: any = 0;

a = "teste";
a = true;
a = [];
```

Constantes

TypeScript

■ Constantes

- Para declarar uma constante, ou seja, um valor que não se modifica no decorrer da execução do programa, use a palavra-chave `const`:

```
const PI : number = 3.141592653589793;
```

Operadores

TypeScript

■ Operadores

- Um operador executa alguma operação em um ou vários operandos (valores) e produz um resultado.

TypeScript

■ Operadores

□ Segue os principais tipos de operadores utilizados em TypeScript:

- Operadores de Atribuição
- Operadores de Comparação
- Operadores Aritméticos
- Operadores Lógicos

TypeScript

■ Operadores

□ Atribuição

Operador Encurtado	Significado	Significado
<code>a = b</code>	<code>a = b</code>	Atribui o valor de b a a.
<code>a += b</code>	<code>a = a + b</code>	Atribui o resultado de a mais b, a a.
<code>a -= b</code>	<code>a = a - b</code>	Atribui o resultado de a menos b, a a.
<code>a *= b</code>	<code>a = a * b</code>	Atribui o resultado de a multiplicado por b, a a.
<code>a /= b</code>	<code>a = a / b</code>	Atribui o resultado de a dividido por b, a a.
<code>a %= b</code>	<code>a = a % b</code>	Atribui o resultado de a módulo b, a a.
<code>a <<= b</code>	<code>a = a << b</code>	Atribui o resultado de a deslocado para a esquerda por b, a a.
<code>a >>= b</code>	<code>a = a >> b</code>	Atribui o resultado de a deslocado para a direita (sinal preservado) por b, a a.
<code>a >>>= b</code>	<code>a = a >>> b</code>	Atribui o resultado de a deslocado para a direita por b para a.
<code>a &= b</code>	<code>a = a & b</code>	Atribui o resultado de a AND b, a a.
<code>a ^= b</code>	<code>a = a ^ b</code>	Atribui o resultado de a XOR b, a a.
<code>a = b</code>	<code>a = a b</code>	Atribui o resultado de a OR b, a a.

TypeScript

■ Operadores

□ Comparação

Operador	Significado	Exemplo	Descrição
=	Igual	a == b	Retorna true caso os operandos sejam iguais.
!=	Não igual	a != b	Retorna true caso os operandos não sejam iguais.
<	Menor que	a < b	Retorna true caso o operando da esquerda seja menor que o da direita.
>	Maior que	a > b	Retorna true caso o operando da esquerda seja maior que o da direita.
<=	Menor ou igual que	a <= b	Retorna true caso o operando da esquerda seja menor ou igual ao da direita.
>=	Maior ou igual que	a >= b	Retorna true caso o operando da esquerda seja maior ou igual ao da direita.
===	Estritamente Igual	a === b	Retorna true caso os operandos sejam iguais e do mesmo tipo.
!==	Estritamente não igual	a !== b	Retorna true caso os operandos não sejam iguais e/ou não sejam do mesmo tipo.

TypeScript

■ Operadores

□ Aritméticos

Operador	Significado	Exemplo	Descrição
+	Adição	a + b	Soma os números. Também concatena string se usar strings em algum operando.
-	Subtração	a - b	Subtrai o número da direita do número da esquerda.
*	Multiplicação	a * b	Multiplica os números.
/	Divisão	a / b	Divide o número da esquerda pelo número da direita.
%	Módulo	a % b	Retorna o resto da divisão do número da esquerda pelo número da direita.
**	Exponenciação	a ** b	Eleva o primeiro operando à potência do segundo operando.
++	Incremento	a++	Adiciona um (1) ao operando. <ul style="list-style-type: none">• Operador pósfixado (a++): retorna o valor de seu operando antes da adição.• Operador prefixado (++a), retorna o valor de seu operando após a adição.
--	Decremento	a--	Subtrai um (1) ao operando. <ul style="list-style-type: none">• Operador pósfixado (a--): retorna o valor de seu operando antes da subtração.• Operador prefixado (--a), retorna o valor de seu operando após a subtração.

TypeScript

■ Operadores

□ Lógicos

Operador	Significado	Exemplo	Descrição
&&	AND lógico	a && b	Quando aplicado a valores booleanos, retorna true se ambos os operandos são true e false caso contrário. Se um operando não for um booleano, retorna o primeiro valor false ou o último valor, se nenhum for encontrado.
	OR lógico	a b	Quando aplicado a valores booleanos, se algum de seus argumentos for true , retorna true ; caso contrário, ele retorna false . Se um operando não for um booleano, ele será convertido em um booleano para a avaliação. Se a pode ser convertido em true , retorna a; senão, retorna b.
!	NOT lógico	!a	Retorna true se o valor for false , e retorna false se o valor for true . Quando aplicado a um valor não booleano, primeiro converte o valor em um valor booleano e depois o nega.

Arrays

TypeScript

- Arrays

```
//arrays
const myNumbers: number[] = [1, 2, 3];

myNumbers.push(4);

console.log(myNumbers);
console.log(myNumbers.length)
```

TypeScript

- Arrays

- Métodos que podem auxiliar durante o uso de arrays

- **push**
 - **some**
 - **reduce**

Tuplas

TypeScript

■ Tuplas

- Define como um array será constituído

```
//tuplas  
let myTuple: [number, string, string[]];  
  
myTuple = [10, "string", ["a", "b", "c"]];
```

Object literals

TypeScript

- Object literals

```
// object literals
const user: { name: string; age: number } = {
  name: "Matheus",
  age: 30,
};

console.log(user);
console.log(user.name)
console.log(user.age)
```

TypeScript

■ Object literals

- ❑ Vamos criar um array de objetos no formato chave-valor

```
let alunos:{ nome : string; idade : number}[] = []
```

- ❑ Agora vamos criar instâncias do objeto aluno e adicioná-las ao array.

```
// Inserindo alunos no array  
alunos.push({ nome: "Alice", idade: 20 });  
alunos.push({ nome: "Bob", idade: 22 });  
alunos.push({ nome: "Carlos", idade: 21 });
```

- ❑ Apresentando os alunos

```
// Exibindo os alunos  
console.log(alunos);
```


TypeScript

■ Object literals

- Podemos criar os alunos separadamente e, em seguida, adicioná-los ao array:

```
const aluno1 = { nome: "Alice", idade: 20 };
const aluno2 = { nome: "Bob", idade: 22 };
const aluno3 = { nome: "Carlos", idade: 21 };

// Adicionando os alunos ao array
alunos.push(aluno1, aluno2, aluno3);

// Exibindo os alunos
console.log(alunos);
```

Union type

TypeScript

- Union type

- Proposta para se evitar utilizar tipo **any**

```
//union type  
let id: number | string = "10";  
id = 10;  
id = "200"
```

Enum

TypeScript

- Enum

```
//enum
//Outra forma de organizar o código
//tamanho de roupa (size: medio, size: pequeno)

enum Size {
    P = "Pequeno",
    M = "Médio",
    L = "Grande",
}

const camisa = {
    name: "Camisa gola V",
    size: Size.M,
};

console.log(camisa);
```

Literal types

TypeScript

- Literal types

- Definir apenas determinados valores para aquela variável

```
//literal types  
let teste: "algunvalor" | null  
  
//teste = "outrovalor"  
teste = null  
teste = "algunvalor"
```

Funções

TypeScript

■ Funções

- ❑ Dominar um estilo de programação funcional é essencial para trabalhar com JavaScript moderno e TypeScript.
- ❑ Para declarar um função, basta fornecer: o nome da função, os parâmetros (se tiver), o tipo de retorno da função e o corpo da função (com os procedimentos e retorno da função):

```
function NOME_DA_FUNCAO(PARAMETROS): TIPO {  
    //corpo da função  
}
```

TypeScript

■ Funções

□ Exemplo:

```
function media(x: number, y: number): number {  
    return (x + y) / 2;  
}
```

□ Para "chamar" esta função, basta passar os argumentos desejados:

```
let resultado = media(6, 7);
```

TypeScript

■ Funções

- Outra forma de declarar a função é atribuindo a função a uma variável:

```
let media = function (x: number, y: number) : number {  
    return (x + y) / 2;  
}
```

TypeScript

■ Arrow Functions

- A partir do ECMAScript 6, foi introduzida uma sintaxe mais compacta de declarar funções, conhecida como Arrow Functions (funções de seta), por utilizarem o operador **=>**.

TypeScript

■ Arrow Functions

- A sintaxe básica é a seguinte:

```
// Sintaxe básica:  
variavel = (PARAMETROS) => {  
    //corpo da função  
}  
  
// Se há somente um único parâmetro, o parênteses é opcional  
variavel = unicoParametro => {  
    // corpo da função  
}  
  
// Se não há parâmetros, deve ser escrita com um par de parênteses  
variavel = () => {  
    // corpo da função  
}
```

TypeScript

■ Arrow Functions

- Segue o exemplo da função media escrita no formato de arrow function:

```
let media = (x: number, y: number) : number => {  
    return (x + y) / 2;  
};
```

TypeScript

■ Funções

- Parâmetros: Não é aceito definir tipo por inferência, apenas por annotation.

inferência

```
//funções
function sum(a, b) {
    return a + b;
}
```

annotation

```
//funções
function sum(a: number, b: number) {
    return a + b;
}
```

```
console.log(sum(2, 2));
```

TypeScript

■ Funções

- Retornando uma string

```
function sayHelloTo(name: string): string {  
    return `Hello ${name}!`;  
}  
  
console.log(sayHelloTo("Matheus"));
```


TypeScript

■ Funções


- Não retorna nenhum tipo

```
function logger(msg: string): void {  
    console.log(msg);  
}  
  
logger("Testando");
```

TypeScript

■ Funções

Parâmetro opcional



```
function greeting(name: string, greet?: string): void{  
    if (greet) {  
        console.log(`Hello ${greet} ${name}`);  
        return;  
    }  
    console.log(`Hello ${name}`);  
}  
  
greeting("João");  
greeting("Pedro", "Sir");
```

Interfaces

TypeScript

- Interfaces

```
//Interfaces
interface MathFunctionParams {
  n1: number;
  n2: number;
}

function sumNumbers(nums: MathFunctionParams) {
  return nums.n1 + nums.n2;
}

function multiplyNumbers(nums: MathFunctionParams) {
  return nums.n1 * nums.n2;
}

const someNumbers: MathFunctionParams = {
  n1: 10,
  n2: 12,
};

console.log(multiplyNumbers(someNumbers));
```

Narrowing

TypeScript

■ Narrowing

- ❑ Realizar checagem de tipos

```
//narrowing
function doSomething(info: number | boolean) {
    if (typeof info === "number") {
        console.log(`O número é ${info}`);
        return;
    }

    console.log("Não foi passado um número!");
}

doSomething(5);
doSomething(false);
```

Classes


TypeScript

■ Classes

- ❑ A versão ECMAScript2015 introduziu o conceito de classes em JavaScript.
- ❑ Em JavaScript, uma classe é um tipo de função, mas em vez de usar a palavra-chave function utiliza-se a palavra-chave class e as propriedades são atribuídas dentro de um método construtor – constructor() - que é chamado sempre que o objeto de classe é inicializado.
- ❑ No TypeScript usa-se da mesma forma.


```
class User {  
  name;  
  role;  
  isApproved;  
  
  constructor(name: string, role: string, isApproved: boolean) {  
    this.name = name;  
    this.role = role;  
    this.isApproved = isApproved;  
  }  
  
  showUserName() {  
    console.log(`O nome do usuário é: ${this.name}`);  
  }  
}  
  
const zeca = new User("Zéca", "Admin", true);  
  
zeca.showUserName();
```

Tipagem dos atributos feita no construtor



Interfaces em classes

Type

■ Interface

```
//interface em classes
interface IVehicle {
    brand: string;
    showBrand(): void;
}

class Car implements IVehicle {
    brand;
    wheels;

    constructor(brand: string, wheels: number) {
        this.brand = brand;
        this.wheels = wheels;
    }

    showBrand(): void {
        console.log(`A marca do veículo é: ${this.brand}`);
    }
}

const fusca = new Car("VW", 4);

fusca.showBrand();
```

Herança

TypeScript

- Herança

```
//heranca
class SuperCar extends Car {
    engine;

    constructor(brand: string, wheels: number, engine: number) {
        super(brand, wheels);
        this.engine = engine;
    }
}

const a4 = new SuperCar("Audi", 4, 2.0);
```

Conditionais

TypeScript

■ Condicionais

- ❑ Ao escrever um programa, pode haver uma situação em que precise adotar um caminho, dentro de conjunto de caminhos.
- ❑ Nesses casos, é preciso utilizar instruções condicionais que permitem que o programa tome decisões corretas:
 - Declaração if
 - Declaração else
 - Declaração else if
 - Declaração switch
 - Operador Condicional Ternário

TypeScript

■ Condicionais

□ Declaração if

```
if (condition) {  
    // Bloco de código a ser executado se a condição for verdadeira  
}
```

```
if (hour < 18) {  
    greeting = "Bom dia!";  
}
```


TypeScript

■ Condicionais

□ Declaração else

```
if (condition) {  
    // Bloco de código a ser executado se a condição for verdadeira  
} else {  
    // Bloco de código a ser executado se a condição for falsa  
}
```

```
if (hour < 18) {  
    greeting = "Bom dia";  
} else {  
    greeting = "Boa noite";  
}
```

TypeScript

■ Condicionais

□ Declaração else if

```
if (condicao1) {  
    // Bloco de código a ser executado se a condição for verdadeira  
} else if (condicao2) {  
    // Bloco de código a ser executado se a condicao1 for falsa e a condição2  
    for verdadeira  
} else {  
    // Bloco de código a ser executado se a condicao1 for falsa e a condicao2  
    for falsa  
}
```

```
if (horas < 12) {  
    greeting = "Bom dia!";  
} else if (horas < 18) {  
    greeting = "Boa tarde!";  
} else {  
    greeting = "Boa noite!";  
}
```

TypeScript

■ Condicionais

□ Declaração switch

```
switch(expressao) {  
  case x:  
    // Bloco de código  
    break;  
  case y:  
    // Bloco de código  
    break;  
  default:  
    // Bloco de código  
}
```

```
switch ( new Date().getDay() ) {  
  case 0:  
    dia = "Domingo";  
    break;  
  case 1:  
    dia = "Segunda";  
    break;  
  case 2:  
    dia = "Terça";  
    break;  
  case 3:  
    dia = "Quarta";  
    break;  
  case 4:  
    dia = "Quinta";  
    break;  
  case 5:  
    dia = "Sexta";  
    break;  
  case 6:  
    dia = "Sábado";  
}
```

TypeScript

■ Condicionais

□ Operador Condicional Ternário

```
condition ? expression1 : expression;  
  
//Também é possível atribuir o resultado da operação a uma variável  
variavel = condition ? expression1 : expression2;
```

```
new Date().getHours() < 18 ? "Bom dia" : "Boa noite";
```

Estrutura de repetição

TypeScript

■ Estruturas de Repetição

- Estruturas de repetição são conhecidas como loops ou laços e permitem executar um bloco de código várias vezes:
 - Laço for
 - For ... of
 - Laço while
 - Laço do/while

TypeScript

■ Estruturas de Repetição

□ Laço for

```
for (inicialização; condição-saída; expressão-final) {  
    // Bloco de código a ser executado  
}
```

```
for (let counter = 1; counter <= 5; counter++) {  
    console.log('Loop: ' + counter);  
}
```

TypeScript

■ Estruturas de Repetição

□ Laço for ... of

```
let carros = [ 'fiesta', 'onix', 'fusca', 'saveiro' ];  
  
let vet : number[] = [1,2,3,4];  
  
for (let carro of carros) {  
    console.log(carro);  
}
```


TypeScript

■ Estruturas de Repetição

□ Laço while

```
while (condição) {  
    // Bloco de código a ser executado  
}
```

```
let count = 1;  
while (count < 10) {  
    console.log(count);  
    count += 2;  
}  
  
// Saída: 1 3 5 7 9
```

TypeScript

■ Estruturas de Repetição

□ Laço do/while

```
do {  
    // Bloco de código a ser executado  
}  
while (condição);
```

```
let count = 0;  
do {  
    count+=2;  
    console.log('count:' + count);  
} while (count < 10);  
  
// Saída: 1 3 5 7 9
```

Requisições Assíncronas

TypeScript

■ Promisses

- ❑ Promises (ou promessa) são uma das maneiras de lidar com operações assíncronas em JavaScript.
- ❑ Uma promise é semelhante a uma promessa da vida real: quando fazemos uma promessa na vida real, é uma garantia de que faremos algo no futuro.
- ❑ Uma promise é um objeto para processamento assíncrono, e representa um valor que pode estar disponível agora, no futuro, ou nunca.

TypeScript

■ Promisses

- Pode estar nos seguintes estados:
 - Pendente (pending): estado inicial, antes de receber algum resultado;
 - Resolvido (resolved): Promessa cumprida;
 - Rejeitado (rejected): Promessa fracassada.

- Por exemplo, quando solicitamos dados de um servidor Web usando uma promise. Esta ficará no estado **pending** até recebermos os dados. Se conseguirmos obter as informações do servidor, mudará para o estado de **resolved**. Mas se não obtivermos as informações, a promessa estará no estado **rejected**.

TypeScript

■ Promisses

- Para criar uma promise é preciso utilizar o construtor new Promise():

```
let promise = new Promise(function(resolve,reject){  
    setTimeout(  
        () => resolve("resolvido"),2000  
    );  
});
```

```
let promise = new Promise(function(resolve,reject){  
    setTimeout(  
        () => reject("rejeitado"),2000  
    );  
});
```

TypeScript

■ Promisses

- Um objeto promise serve de elo entre o executor e as funções de consumo, que receberão o resultado (com sucesso ou erro).
- As funções de consumo podem ser assinadas usando os métodos **.then**, **.catch** e **.finally**.

TypeScript

■ Promisses

- O **.then** é o mais importante e fundamental e sua sintaxe básica é:

```
promise.then(  
  function(result) { /* Trata um resultado de sucesso */ },  
  function(error) { /* Trata com um erro */ }  
);
```

TypeScript

■ Promisses

- ❑ O primeiro argumento de `.then` é uma função executada quando a promessa é resolvida com sucesso e recebe o resultado. O segundo argumento de `.then` é uma função executada quando a promessa é rejeitada e recebe o erro.
- ❑ O exemplo a seguir apresenta o construtor da promise e o seu consumo:

```
let promise = new Promise(function(resolve, reject){
  setTimeout(
    () => reject("rejeitado"), 2000
  );
})

promise.then(
  result => alert(result),
  error => alert("Erro: "+error)
);
```

Dúvidas ???



Prof. Davi Taveira Alencar Alarcão
davitaveira@yahoo.com.br
<http://sites.google.com/site/davitaveira>