



**SOFTEx**  
PERNAMBUCO

 **Softex**

MINISTÉRIO DA  
CIÊNCIA, TECNOLOGIA  
E INOVAÇÃO

GOVERNO FEDERAL  
**BRASIL**  
UNIÃO E RECONSTRUÇÃO



## Aula 19 | Módulo: Typescript e Orientação a Objetos



- Interação entre objetos e construção de exemplos práticos
- Sintaxe do TypeScript para orientação a objetos: class, constructor, this, public, private

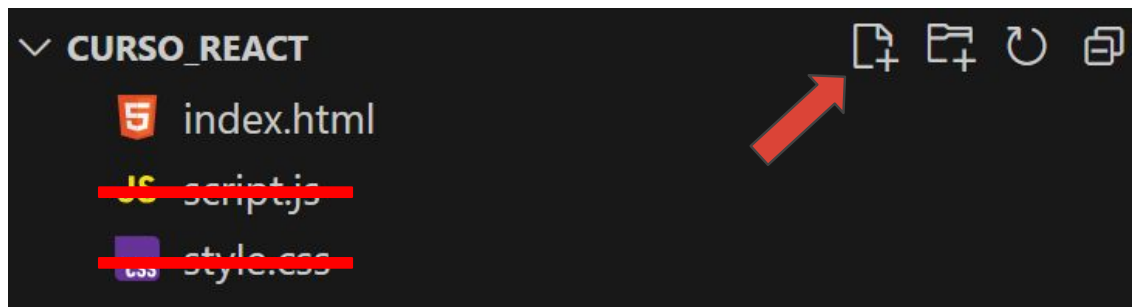


## Abrindo editor de código



### Vamos agora abrir o VSCode e criar os arquivos

- Com o programa aberto, clique em File > Open Folder... (Arquivo > Abrir Pasta...).
- Escolha um local para criar a sua pasta, crie uma nova pasta e dê o nome de **seunome\_aula\_19**. Depois dê dois clique nessa pasta criada e clique em **Selecionar pasta**. O VSCode reabrirá dentro dessa pasta que foi criada.
- Agora vamos criar o arquivo HTML:
- Dê o nome de **index.html**





# Typescript e Orientação a Objetos



## Configurações iniciais

- Certifique que o NodeJS está instalado:
  - **node -v**
  - **npm -v**
- Se você usa Windows, caso não esteja instalado, baixe e instale:
  - **<https://www.nodejs.tech/pt-br/download>**
- Se você usa Linux, use o gerenciador de pacotes ou nvm se souber usar:
  - **sudo apt install nodejs npm**



# Typescript e Orientação a Objetos



## Configurações iniciais

- No seu projeto, abra o terminal e vamos iniciar o projeto e instalar o pacote necessário:
  - **npm init -y** (criar o package.json, que controla as dependências do projeto.)
- Instale agora o compilador TypeScript como dev-dependência:
  - **npm i -D typescript**
- Gere agora o arquivo de configuração **tsconfig.json**:
  - **npx tsc --init**
- Crie uma pasta chamada **src** no diretório raiz do projeto (o mesmo diretório que está o index.html).
- Dentro da pasta **src**, crie o arquivo **main.ts**



# Typescript e Orientação a Objetos



## Configurações iniciais

- Quando o arquivo **tsconfig.json** for criado, vamos apagar todo o conteúdo e adicionar o que temos abaixo:

```
{
  "compilerOptions": {
    "target": "ES2017",
    "lib": ["DOM", "ES2017"],
    "module": "ES2015",
    "moduleResolution": "Node",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src"]
}
```

Copie esse

```
tsconfig.json > ...
1  {
2    "compilerOptions": {
3      "target": "ES2017",           // versão do JS de saída (mais moderno e compatível)
4      "lib": ["DOM", "ES2017"],    // bibliotecas incluídas (DOM = navegador, ES2017 = recursos modernos)
5      "module": "ES2015",          // formato de módulo compatível com navegador (sem "exports")
6      "moduleResolution": "Node",  // forma de resolver módulos (padrão do Node, funciona bem em geral)
7      "rootDir": "./src",          // pasta onde ficam os arquivos .ts de origem
8      "outDir": "./dist",          // pasta onde serão gerados os .js compilados
9      "strict": true,              // ativa verificações mais rigorosas de tipagem
10     "esModuleInterop": true,      // compatibilidade para importar libs JS antigas (ex: import express from "express")
11     "skipLibCheck": true          // pula checagem de tipos em libs externas (compila mais rápido)
12   },
13   "include": ["src"]              // define quais pastas/arquivos entram na compilação
14 }
15
```

Explicação



# Typescript e Orientação a Objetos



## Estrutura de Pastas + HTML base

- Crie o arquivo HTML e carregue o JS que será compilado.

```
index.html > ...
1  <!DOCTYPE html>
2  <html lang="pt-BR">
3
4  <head>
5    <meta charset="UTF-8" />
6    <title>Aula 18</title>
7  </head>
8
9  <body>
10   <h1>TypeScript + P00</h1>
11   <p>Abra o console (F12) para ver a saída.</p>
12   <script src="./dist/main.js"></script>
13 </body>
14
15 </html>
```

*Estamos apontando para **./dist/main.js** pois é o diretório onde o .js final ficará.*



## Typescript e Orientação a Objetos



### “Assistir” mudanças (watch mode)

- Para compilar automaticamente ao salvar .ts, use o watch:
  - **npx tsc -w**
- Mantenha esse comando rodando enquanto edita TS.
  - Modifique a string da mensagem no arquivo **src/main.ts** e confirme que o **dist/main.js** é re-gerado.
- Atualize o navegador para ver mudanças.





# Typescript e Orientação a Objetos



## Relembrando: o que é um objeto?

- Um objeto representa uma entidade do mundo real dentro do programa.
- Cada objeto tem:
  - Atributos: características (dados).
  - Métodos: comportamentos (ações).



# Typescript e Orientação a Objetos



## Exemplo

```
src > ts main.ts > ...
1  class Pessoa {
2      nome: string;
3      idade: number;
4
5      constructor(nome: string, idade: number) {
6          this.nome = nome;
7          this.idade = idade;
8      }
9
10     cumprimentar() {
11         console.log(`Olá! Meu nome é ${this.nome}.`);
12     }
13 }
14
15 const pessoa1 = new Pessoa("Ana", 25);
16 pessoa1.cumprimentar();
```

**this.nome** (refere-se ao atributo do próprio objeto)

**new Pessoa(...)** (cria uma nova instância)



# Typescript e Orientação a Objetos



## Quando objetos interagem

- Até agora, cada classe que criamos fazia tudo sozinha.
- Mas na vida real, um objeto raramente funciona isolado.
  - Um carro precisa de um motor.
  - Um jogador precisa de um inventário.
  - Um professor precisa de uma turma.
- Essa colaboração entre objetos é o que chamamos de interação entre objetos.
- Existem tipos de relações:
  - **Composição**: um objeto depende completamente do outro.
  - **Agregação**: um objeto contém outro, mas eles podem existir separadamente.
  - **Associação**: um objeto usa outro.



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Composição

- Primeiro vamos criar a class **Motor**:

```
src > TS main.ts > ...  
1  class Motor {  
2      ligar() {  
3          console.log("Motor ligado!");  
4      }  
5  }
```

- Aqui definimos o que é um Motor.
- Ele tem um comportamento (método **ligar()**), que apenas mostra uma mensagem.
- É uma classe simples, mas independente.
- Pense como um “mini-programa” que pode ser reutilizado em qualquer carro.



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Composição

- Abaixo da class Motor, vamos criar a class **Carro**:

```
src > TS main.ts > ...  
7 class Carro {  
8     modelo: string;  
9     motor: Motor; // o carro tem um motor  
10 }
```

- A classe **Carro** tem um atributo motor, que é do tipo **Motor**.
- Isso significa que um Carro possui um Motor.
- Essa é uma relação de composição: o carro “contém” um motor.
- “O motor só existe porque o carro foi criado. Se o carro deixar de existir, o motor também vai.”



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Composição

- Dentro da classe Carro, vamos criar o método **constructor**:

```
src > TS main.ts > ...  
7   class Carro {  
  
11     constructor(modelo: string) {  
12         this.modelo = modelo;  
13         this.motor = new Motor(); // criando o motor junto com o carro  
14     }  
15 }
```

- O **constructor** é executado automaticamente quando criamos o objeto com `new Carro()`.
- Ele recebe o nome do modelo e cria um novo motor dentro do carro.
- A expressão **this.motor = new Motor()** significa:
  - “Esse carro vai ter um motor novo dentro dele.”



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Composição

- Dentro da classe Carro, vamos criar o método **dirigir()**:

```
src > TS main.ts > ...  
7   class Carro {  
...  
16   dirigir() {  
17       this.motor.ligar();  
18       console.log(`${this.modelo} está em movimento!`);  
19   }  
20 }  
21
```

- Aqui acontece a interação real entre objetos.
- O carro chama um método de outro objeto (motor.ligar()).
- O método **ligar()** pertence ao objeto motor, não ao carro.
- Depois disso, o carro imprime a mensagem informando que está se movendo.



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Composição

```
src > ts main.ts > ...
1  class Motor {
2      ligar() {
3          console.log("Motor ligado!");
4      }
5  }
6
7  class Carro {
8      modelo: string;
9      motor: Motor; // o carro tem um motor
10
11     constructor(modelo: string) {
12         this.modelo = modelo;
13         this.motor = new Motor(); // criando o motor junto com o carro
14     }
15
16     dirigir() {
17         this.motor.ligar();
18         console.log(`${this.modelo} está em movimento!`);
19     }
20 }
21
22 const carro = new Carro("Tesla");
23 carro.dirigir();
24
```

### Fluxo de execução:

- O método **dirigir()** é chamado.
- Ele acessa o motor através de **this.motor**.
- O motor executa seu próprio método **ligar()**.
- Em seguida, o carro mostra a mensagem "Tesla está em movimento!".





# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Agregação

- Agora fazer o carro receber o motor de fora.
- Isso é uma agregação, e torna o código mais flexível. O mesmo motor poderia ser usado em outro carro.
- Primeiro vamos criar a class **Motor**:

```
src > TS main.ts > ...  
1 class Motor {  
2     ligar() {  
3         console.log("Motor ligado!");  
4     }  
5 }
```



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Agregação

- Abaixo da class Motor, vamos criar a class **Carro**:

```
src > TS main.ts > ...  
7   class Carro {  
8     modelo: string;  
9     motor: Motor;  
10  }
```



# Typescript e Orientação a Objetos



## Exemplo prático de interação entre objetos - Agregação

- Dentro da classe Carro, vamos criar o método **constructor** e o método **dirigir()**:

```
src > TS main.ts > ...
7  class Carro {
11     constructor(modelo: string, motor: Motor) {
12         this.modelo = modelo;
13         this.motor = motor;
14     }
15
16     dirigir() {
17         this.motor.ligar();
18         console.log(`${this.modelo} está em movimento!`);
19     }
20 }
21
22 const motorNovo = new Motor();
23 const carro = new Carro("Tesla", motorNovo);
24 carro.dirigir();
```



# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Vamos criar um pequeno sistema com uma Loja e Produtos.
- A loja pode adicionar produtos e mostrar o nome de cada produto cadastrado.



# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Vamos criar uma classe **Produto**
- Ele vai representar um produto simples, com nome e preço.
- Cada vez que usarmos **new Produto(...)**, criamos um objeto diferente.

```
src > TS main.ts > ...  
1  class Produto {  
2      nome: string;  
3      preco: number;  
4  
5      constructor(nome: string, preco: number) {  
6          this.nome = nome;  
7          this.preco = preco;  
8      }  
9  }  
10
```




# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Agora vamos criar uma classe **Loja**
- A loja tem um nome e uma lista de produtos.
- No constructor, começamos com a lista vazia.

src >  main.ts > ...

```
11 class Loja {  
12     nome: string;  
13     produtos: Produto[]; // a loja tem vários produtos  
14  
15     constructor(nome: string) {  
16         this.nome = nome;  
17         this.produtos = []; // começa vazia  
18     }  
19
```



# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Agora vamos criar o método **adicionarProduto()**:
- Ele vai receber um objeto do tipo Produto.
- Usaremos o comando **push** para colocar esse produto dentro da lista da loja.
- “É como colocar um item dentro de uma sacola.”

```
src > TS main.ts > ...  
11 class Loja {  
19  
20     adicionarProduto(produto: Produto) {  
21         this.produtos.push(produto);  
22     }  
23 }
```



# TypeScript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Agora vamos criar o método **mostrarProdutos()**:
- O comando for passa por cada produto que está dentro da lista.
- A cada volta, ele pega o produto da posição **i** e mostra seu nome e preço.

```
src > TS main.ts > ...
11  class Loja {
24      mostrarProdutos() {
25          console.log(`Produtos da loja ${this.nome}:`);
26
27          // Percorrendo os produtos
28          for (let i = 0; i < this.produtos.length; i++) {
29              const p = this.produtos[i];
30              console.log(`- ${p.nome} (R$ ${p.preco})`);
31          }
32      }
33  }
```





# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Agora vamos criar alguns produtos, criar a loja, adicionar produtos à loja e depois mostrar os produtos cadastrados:

```
src >  main.ts > ...  
35 // Criando alguns produtos  
36 const p1 = new Produto("Camisa", 59);  
37 const p2 = new Produto("Calça", 89);  
38 const p3 = new Produto("Tênis", 199);  
39  
40 // Criando a loja  
41 const loja = new Loja("Loja do Hygor");  
42  
43 // Adicionando produtos à loja  
44 loja.adicionarProduto(p1);  
45 loja.adicionarProduto(p2);  
46 loja.adicionarProduto(p3);  
47  
48 // Mostrando os produtos cadastrados  
49 loja.mostrarProdutos();  
50
```



# Typescript e Orientação a Objetos



## Exercício Guiado: Loja e Produto

- Desafio:
  - Adicione agora um novo produto chamado **Boné** com preço **29**.
  - Visualize se esse novo produto adicionado vai aparecer no console.



## Typescript e Orientação a Objetos



### O que é o constructor e para que ele serve?

- Toda vez que criamos um objeto com new, o TypeScript precisa saber como montá-lo.
- O método constructor é o primeiro código executado dentro de uma classe quando criamos um novo objeto.
- Ele serve para:
  - Definir valores iniciais para os atributos.
  - Receber informações que o objeto precisa para existir.
  - Garantir que o objeto já nasça completo.



# Typescript e Orientação a Objetos



O que é o constructor e para que ele serve?

```
src > TS main.ts > ...
1  class Pessoa {
2      nome: string;
3      idade: number;
4
5      constructor(nome: string = "Desconhecido", idade: number = 0) {
6          this.nome = nome;
7          this.idade = idade;
8      }
9
10     apresentar() {
11         console.log("Olá, meu nome é " + this.nome + " e tenho " + this.idade + " anos.");
12     }
13 }
14
15 // Criando um objeto
16 const pessoa1 = new Pessoa("Lucas", 25);
17 const pessoa2 = new Pessoa();
18 pessoa1.apresentar();
19 pessoa2.apresentar();
20
```



## Typescript e Orientação a Objetos



### Entendendo o this: quem ele é e por que é importante

- Toda vez que criamos um objeto, precisamos de uma forma de acessar os dados que pertencem a ele mesmo.
- O **this** é o pronome pessoal dos objetos.
- Ele significa: “eu mesmo que estou neste objeto aqui!”.



# Typescript e Orientação a Objetos



## Entendendo o this: quem ele é e por que é importante

```
src > TS main.ts > ...
1  class Pessoa {
2      nome: string;
3      idade: number;
4
5      constructor(nome: string = "Desconhecido", idade: number = 0) {
6          this.nome = nome;
7          this.idade = idade;
8      }
9
10     apresentar() {
11         console.log("Olá, meu nome é " + this.nome + " e tenho " + this.idade + " anos.");
12     }
13 }
14
15 // Criando um objeto
16 const pessoa1 = new Pessoa("Lucas", 25);
17 const pessoa2 = new Pessoa();
18 pessoa1.apresentar();
19 pessoa2.apresentar();
20
```



# Typescript e Orientação a Objetos



## Exercício 1

- Crie uma classe chamada **Cachorro** com os atributos **nome** e **raca**.
- Crie um método **latir()** que mostra a mensagem:
  - “O cachorro NOME (RAÇA) está latindo!”



# Typescript e Orientação a Objetos



## Modificadores de acesso: public e private

- Em um sistema real, nem toda informação deve estar visível ou ser modificada por qualquer parte do programa.
- Assim como na vida real, existem dados públicos e dados privados.
- Exemplo cotidiano:
  - Público: o nome de uma pessoa (pode ser mostrado).
  - Privado: sua senha bancária (não deve ser acessada diretamente).





# Typescript e Orientação a Objetos



## Modificadores de acesso: public e private

```
src > TS main.ts > ...  
1 class ContaBancaria {  
2     titular: string;  
3     private saldo: number;  
4  
5     constructor(titular: string, saldoInicial: number) {  
6         this.titular = titular;  
7         this.saldo = saldoInicial;  
8     }  
9 }
```

- **titular** é público (pode ser acessado diretamente de fora da classe)
  - Isso é útil, pois o nome do titular não precisa ser escondido.
- **saldo** é privado (só pode ser acessado de dentro da classe)
  - Isso impede que outras partes do código modifiquem o saldo diretamente, como:
  - **conta.saldo = 9999999;** (isso protege o código de erros e de alterações indevidas)



# Typescript e Orientação a Objetos



## Modificadores de acesso: public e private

- Métodos como “portas seguras”
  - Eles são o único jeito seguro de mexer no saldo.
  - Assim, o saldo só muda de forma controlada e intencional.

```
src > TS main.ts > ...
1  class ContaBancaria {
10     depositar(valor: number) {
11         this.saldo = this.saldo + valor;
12         console.log("Depósito de R$" + valor + " realizado com sucesso!");
13     }
14
15     verSaldo() {
16         console.log("Saldo atual: R$" + this.saldo);
17     }
18 }
19
```



# Typescript e Orientação a Objetos



## Exercício 2

- Crie uma classe **Cofrinho** com um valor inicial de 0.
- O valor deve ser privado, e o cofrinho deve ter métodos para:
  - depositar(valor)
  - verTotal()



# Typescript e Orientação a Objetos



## Gabarito Exercício 1

```
src > TS main.ts > ...
1  class Cachorro {
2      nome: string;
3      raca: string;
4
5      constructor(nome: string, raca: string) {
6          this.nome = nome;
7          this.raca = raca;
8      }
9
10     latir() {
11         console.log("O cachorro " + this.nome + " (" + this.raca + ") está latindo!");
12     }
13 }
14
15 const c1 = new Cachorro("Rex", "Labrador");
16 const c2 = new Cachorro("Luna", "Poodle");
17
18 c1.latir();
19 c2.latir();
20
```



# Typescript e Orientação a Objetos



## Gabarito Exercício 2

```
src > TS main.ts > ...
1  class Cofrinho {
2      private total: number = 0;
3
4      depositar(valor: number) {
5          this.total = this.total + valor;
6      }
7
8      verTotal() {
9          console.log(`Total guardado: R${this.total}`);
10     }
11 }
12
13 const meuCofrinho = new Cofrinho();
14 meuCofrinho.depositar(10);
15 meuCofrinho.depositar(5);
16 meuCofrinho.verTotal();
17
```



# ATÉ A PRÓXIMA AULA!

*Front-end - Design. Integração. Experiência.*

**Professor:** Hygor Rasec

<https://www.linkedin.com/in/hygorrasec>

<https://github.com/hygorrasec>