

# Introdução a Typescript

Labenu\_



# O que vamos ver hoje? 🙄

- Typescript: uma nova linguagem de programação;
- Paradigmas de programação;
- Níveis de abstração;
- Os processos de transpilação e compilação;
- Instalação e config;
- Como tipar variáveis e funções;
- Tipo any.



# Linguagens de Programação

Labenu\_



# Linguagens de Programação

- As linguagens de programação são um **conjunto de sintaxes** que permitem a comunicação humana com a máquina
- Foram feitas para que o **ser humano** conseguisse se **comunicar** diretamente com o **computador** sem precisar conhecer cada detalhe de seu funcionamento
- Acessam cada posição de memória e solicitam que o processador realize ações que nós não nos preocupamos em fazer



# Linguagens de Programação

- Exemplo de Linguagem de Máquina

```
00000000  B4 03 CD 10 B0 01 B3 0A B9 0E 00 BD 13 7C B4 13 .....|..  
00000010  CD 10 F4 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0D ...Hello World!..  
00000020  0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
-- ex      --0x0/0x200-----
```



# Linguagens de Programação

- Exemplo de Assembly

```
lea si, string ; Atribui SI ao endereço de string.
call printf    ; Coloca o endereço atual na pilha e chama o processo printf

hlt            ; Encerra o computador.
string db "Ola mundo!", 0

printf PROC
    mov AL, [SI] ; Atribui à AL o valor no endereço SI.
    cmp AL, 0    ; Compara AL com nulo.
    je pfend     ; Pula se comparação der igual.

    mov AH, 0Eh
    int 10h      ; Executa uma função da BIOS que imprime o caractere em AL.
    inc SI       ; Incrementa em um o valor de SI.
    jmp printf   ; Pula para o início do processo.

pfend:
    ret         ; Retorna para o endereço na posição atual da pilha.
printf ENDP
```



# Linguagens de Programação

- As linguagens de programação podem ser classificadas de acordo com o seu **nível de abstração**
- **Baixo nível (LLL - Low Level Language)** é quando se aproxima da linguagem de máquina, sendo menos amigável a humanos (ex: Assembly)
- **Alto nível (HLL - High Level Language)** é quando se aproxima à linguagem humana (ex: Javascript, Typescript, Java, Python).



# Linguagens de Programação

- O processo de transformar uma linguagem de alto nível em Assembly é chamado de **compilação**. O código gerado por ele é um **executável**
- Nos navegadores, a compilação do código-fonte é feita em tempo de execução (*runtime*), gerando um arquivo temporário que é, então executado. Esse processo recebe o nome de **interpretação**
- Há certos momentos em que precisamos **converter uma HLL em outra**, antes de realizar a compilação. A esse processo, damos o nome de **transpilação**





# Paradigmas de Programação

Labenu\_



# Paradigmas de Programação

- Paradigmas de programação são os conjuntos de características de uma linguagem que definem a **forma de se pensar** para a resolução de problemas. Exemplos são:
  - Paradigma funcional;
  - Paradigma declarativo;
  - Paradigma orientado a eventos;
  - Paradigma orientado a objetos.



# Paradigmas de Programação

- **Paradigma Funcional**

- É a ideia de se utilizarem **funções** para executar o código
- Neste paradigma, normalmente, todo o código é dividido em funções e o resultado final é a **chamada de todas essas funções**
- Exemplo: Clojure, Elixir, JS, TS, C, etc



# Paradigmas de Programação

- **Paradigma Declarativo**

- Neste paradigma, espera-se que se **descreva** o que o código faz, mas não como ele faz
- O melhor exemplo deste paradigma são as **linguagens de marcação**: HTML, XML, etc.



# Paradigmas de Programação

- **Paradigma Orientado a Eventos**
  - Aqui o código **depende que algo aconteça** para ser executado
    - Você só acorda quando o despertador toca
    - Um formulário só envia as informações quando um usuário clica num botão
  - Exemplos: C#, Java, JS Vanilla



# Paradigmas de Programação

- **Paradigma Orientado a Objetos**

- Neste paradigma, o objetivo é tentar criar um sistema com **modelos** que mais se aproximem do mundo real
- Para isto, fazemos o uso de **objetos** e **classes**
- Exemplos: TS, Java, C++



# Conhecendo o Typescript

Labenu\_



# O que é Typescript?

- É uma **nova linguagem** de programação **orientada a objetos** e com a **sintaxe igual do Javascript**.
- Criada pela Microsoft, tem o intuito de **melhorar a produtividade** e garantir a construção de **aplicações mais seguras**.





# Diferença entre JS e TS

- O Javascript é uma **linguagem dinâmica** e isso significa que podemos enviar qualquer valor para uma função, por exemplo.
- Por isso, podemos cometer erros ao enviar tipos errados de valores para funções, por exemplo. Alguns erros comuns são:
  - **Passar tipos errados de valor** para funções;
  - **Trocar duas letras** de uma variável (sem querer);
  - **Errar o nome** de uma props ou de algo dentro do state;



# Diferença entre JS e TS

- O **Typescript**, por ser uma linguagem **fortemente tipada**, surgiu para diminuir a quantidade de erros.
- Com ele podemos descobrir erros **durante** a escrita de código!
- Seu funcionamento depende de sua **transpilação** para Javascript. Isto é, o código escrito é convertido para JS no build de produção. Isso permite seu código funcionar em todo e qualquer navegador e sistema operacional que o Javascript é executado.



# Instalação

Labenu\_



# Instalação

- A partir de hoje, em cada novo projeto, vamos iniciar instalando o TS com um dos seguintes comandos:

```
$ npm install typescript
```

```
$ npm i typescript
```

- Além disso, nossos arquivos vão sempre terminar com a extensão **.ts**



# Instalação

## Para rodar o projeto:

- Criar um script que **transpile** o código fonte (TS) e **execute** o código de produção (JS):

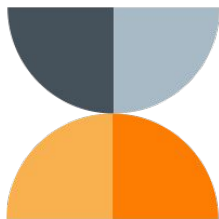
```
"start": "tsc index.ts && node ./build/index.js"
```



# Exercício 1

1. Dentro da pasta do projeto, instale o Typescript, rodando no terminal o comando `npm i typescript`
2. Crie uma pasta **first-script**, contendo um arquivo **index.ts** que imprima no terminal a mensagem *"Hello, world!"*. Para executá-lo, crie um script para rodar os comandos `tsc index.ts` e `node build/index.js`





**10 min**

- **Typescript** é uma linguagem de programação orientada a objetos que pode, como o JS, ser utilizada em outros paradigmas
- Para rodar um código em Typescript, ele passa pelo processo de **transpilação**, gerando um código em **Javascript**, que é, então, executado.
  - **tsc**: realiza a transpilação;
  - **node**: executa o arquivo Javascript gerado.



# Configuração

Labenu\_





# tsconfig.json

- Em um projeto com vários arquivos, o processo de transpilação que vimos até agora é muito pouco prático
- Por isso, vamos criar um **arquivo de configuração** chamado **tsconfig.json** dentro da pasta do nosso projeto, usando o comando `npx tsc --init`
- Configurando esse arquivo, só precisamos usar o comando **tsc** no script e, depois, rodar os arquivos javascript que forem criados

```
"start": "tsc && node ./build/index.js"
```



# tsconfig.json

- Vamos ver este arquivo:

```
{
  "compilerOptions": {
    "target": "es6",          /* Specify ECMAScript target version */
    "module": "commonjs",    /* Specify module code generation */
    "sourceMap": true,       /* Generates corresponding '.map' file. */
    "outDir": "./build",     /* Redirect output structure to the directory. */
    "rootDir": "./src",      /* Specify the root directory of input files. */
    "removeComments": true,  /* Do not emit comments to output. */
    "noImplicitAny": true,   /* Raise error on declarations with an implied 'any' type. */
  }
}
```



## Exercício 2

1. Crie o arquivo de configuração do Typescript, rodando no terminal o comando `npx tsc --init`
2. Altere as chaves: **target** para "es6", **rootDir** para "./src", **outDir** para "./build"
3. Descomente as chaves **sourceMap**, **removeComments** e **noImplicitAny**
4. Crie as pastas **src** e **build**, movendo para elas os arquivos **.ts** e **.js**, respectivamente



# Declarando variáveis

Labenu\_



# Declarando variáveis

- No TS, usamos os mesmos declaradores do Javascript: **const**, **let** e **var**.
- Agora, precisamos colocar o tipo delas logo após o nome

```
const company: string = "Labenu"
```

```
let age: number = 5
```


```
let passwordIsCorrect: boolean = false
```



# Declarando variáveis

- No TS, usamos os mesmos declaradores do Javascript: **const**, **let** e **var**.
- Agora, precisamos colocar o tipo delas logo **após o nome**

```
const company: string = "Labenu"  
let age: number = 5  
let passwordIsCorrect: boolean = false
```



# Declarando Arrays e Objetos

Labenu\_



# Declarando Arrays e Objetos

- Para **arrays**, temos duas opções:

```
const arr: Array<number> = [1, 2, 3]
const array: number[] = [1, 2, 3]
```

- Para **objetos**, o tipo é bem parecido com a **declaração**. Nas propriedades do objeto, colocamos os seus respectivos tipos.

```
const person: { name: string, age: number } = {
  name: "Astrodev",
  age: 30
}
```





# Declarando Arrays e Objetos

- Para **arrays**, temos duas opções:

```
const arr: Array<number> = [1, 2, 3]  
const array: number[] = [1, 2, 3]
```

- Para **objetos**, o tipo é bem parecido com a **declaração**. Nas propriedades do objeto, colocamos os seus respectivos tipos.

```
const person: { name: string, age: number } = {  
  name: "Astrodev",  
  age: 30  
}
```



Tipo Any 🌐

Labenu\_



# Tipo Any 🌐


- Há um tipo especial, que deve ser evitado, mas as vezes é a única opção: **any**. Indica que a variável **pode assumir qualquer valor**.

```
let aux: any
aux = "aux"
aux = 0
aux = true
```



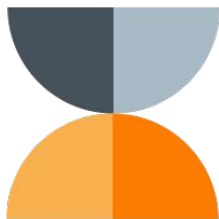
# Tipo Any 🌐

- Há um tipo especial, que deve ser evitado, mas as vezes é a única opção: **any**. Indica que a variável **pode assumir qualquer valor**.



```
let aux: any  
aux = "aux"  
aux = 0  
aux = true
```





*5 min*

- Projetos em TS possuem um arquivo de configuração chamado **tsconfig.json**
- **Variáveis** em TS são declaradas seguindo o padrão **nome: tipo = valor**



# Funções

Labenu\_



# Funções tipadas

- No **Javascript**, podemos criar uma função que recebe dois números e chamá-la passando duas strings, por exemplo. Isso abre margem para bugs!
- No **Typescript**, podemos **tipar sua entrada (parâmetros) e saída (retorno)**, evitando que qualquer tipo de dado seja enviado a função



# Tipando parâmetros

- Abaixo temos uma função **sum** que **recebe dois parâmetros: n1 e n2**, ambos números

parâmetros **não tipados**

```
function sum(n1, n2){  
  return a + b  
}
```

parâmetros **tipados**

```
function sum(  
  n1: number,  
  n2: number  
): number {  
  return a + b  
}
```





# Tipando retorno

- Abaixo temos uma função sum que precisa **retornar a soma de dois números**

retorno **não tipado**

```
function sum(n1, n2){  
  return a + b  
}
```

retorno **tipado**

```
function sum(  
  n1: number,  
  n2: number  
): number {  
  return a + b  
}
```



# Parâmetros

```
function sum(  
  n1: number,  
  n2: number  
): number {  
  return a + b  
}
```

tipagem dos  
parâmetros



# Retorno

```
function sum(  
  n1: number,  
  n2: number  
): number {  
  return a + b  
}
```

tipagem do  
retorno



# Void e parâmetros opcionais

- Quando é uma função que não retorna nada, o tipo dela é **void**
- Parâmetros opcionais são declarados usando **?**

```
function sayHello(name?: string): void {  
    console.log("Hello,", name || "World")  
}
```



# Tipando métodos e callbacks 📞

- Para declaramos **métodos** e **callbacks** , usamos a **sintaxe** parecida com a **arrow function**:

```
type person = {  
  name: string,  
  age: number,  
  sayHello: (name?: string) => void  
}
```

```
function method(  
  condition: boolean,  
  callback: () => void  
): void {  
  if (condition) { callback() }  
}
```



## Exercício 3

Ao lado, há uma função que recebe um array de carros e uma marca. Ela devolve os carros desta marca ou o array inteiro, caso uma marca não seja passada.

1. Faça a tipagem correta dessa função
2. Torne o parâmetro *marca* opcional

```
function buscarCarrosPorMarca(frota, marca) {  
  if (marca === undefined) {  
    return frota  
  }  
  
  return frota.filter(  
    (carro) => {  
      return carro.marca === marca  
    }  
  )  
}
```



# Resumo



Labenu\_



# Resumo

- **Typescript** é uma linguagem orientada a objetos, que precisa ser **transpilada** em JS antes de ser **interpretada**
- Projetos em TS possuem um arquivo de configuração chamado **tsconfig.json**
- **Variáveis** em TS são declaradas seguindo o padrão **nome: tipo = valor**



# Resumo

- Ao declarar **funções**, devemos **tipar** tanto seus **parâmetros** quanto seu **retorno**
- **void** é o tipo de função que não retorna nada



# Dúvidas?





Obrigado!