# Typescript

Prof. José Matheus

# Introdução

TypeScript é um superset de JavaScript, ou seja, um conjunto de ferramentas e formas mais eficientes de escrever código JavaScript, adicionando recursos que não estão presentes de maneira nativa na linguagem.

O TypeScript foi divulgado pela primeira vez em outubro de **2012** (na versão 0.8), após dois anos de desenvolvimento interno na Microsoft.

O Typescript, adicionamos uma camada de tipagem ao Javascript, que permite um feedback mais rápido do código e refatorações mais simples.

*superset = superconjunto = Um conjunto A é um superconjunto de outro conjunto B se todos os elementos do conjunto B forem elementos do conjunto A. Ou seja, o TS possui tudo que o JS possui e mais.*

# Typescript

TypeScript (TS) é considerado um superset do JavaScript (JS) por algumas razões fundamentais:

**Compatibilidade com JavaScript:** TypeScript foi projetado para ser uma extensão do JavaScript. Isso significa que qualquer código JavaScript existente é, por padrão, código TypeScript válido. Você pode gradualmente adotar TypeScript em um projeto JavaScript existente sem a necessidade de uma grande reescrita.

**Tipagem Estática Opcional:** TypeScript adiciona um sistema de tipos estáticos ao JavaScript. No entanto, a tipagem é opcional. Você pode optar por adicionar ou não tipos às suas variáveis, funções e objetos. Isso é útil para desenvolvedores que desejam um código mais robusto com verificações de tipo em tempo de compilação, mas também para aqueles que desejam manter a flexibilidade do JavaScript.

**Sintaxe Estendida:** TypeScript estende a sintaxe do JavaScript com recursos como interfaces, enums, classes, módulos, tipos genéricos e muito mais. Essas adições oferecem uma maneira mais expressiva de modelar e organizar código, tornando-o mais fácil de entender e manter.

**Ferramentas de Desenvolvimento Melhoradas:** TypeScript é acompanhado de ferramentas poderosas, como o TypeScript Compiler (tsc) e editores de código como o Visual Studio Code, que oferecem sugestões de código, correções automáticas, documentação integrada e recursos de refatoração. Isso ajuda os desenvolvedores a escrever código mais eficiente e de alta qualidade.

**Maior Previsibilidade e Manutenibilidade:** Devido à sua tipagem estática, TypeScript pode pegar erros de tipo em tempo de compilação, antes de o código ser executado, tornando-o mais robusto e seguro. Isso é especialmente valioso em projetos grandes e complexos, onde a manutenção é crítica.

# Typescript: por que usar?

O TypeScript nos traz diversos benefícios, mas podemos destacar o potencial de detecção de erros durante o desenvolvimento de projetos e a possibilidade de incluir a inteligência da IDE que está sendo usada. Isso reflete num ambiente muito mais ágil e seguro enquanto o código está sendo digitado pelo usuário.

E além de ser uma linguagem fortemente tipada, trazendo vários conceitos de orientação a objetos como: classes, heranças, encapsulamento, interfaces, etc.

# Typescript: transpilação

Transpilação em JavaScript refere-se ao processo de converter código de uma linguagem para outra, geralmente de uma versão mais recente ou avançada para uma mais antiga ou amplamente suportada. No contexto do JavaScript, isso é comumente usado para converter código escrito em uma versão mais recente do ECMAScript (como ES6/ES2015) para uma versão mais antiga que é suportada por navegadores mais antigos.

No contexto do TypeScript (TS), a transpilação refere-se ao processo de converter código TypeScript em código JavaScript puro. TypeScript é uma linguagem superset do JavaScript que adiciona tipagem estática opcional e outros recursos de linguagem ao JavaScript.

# Typescript: principais conceitos

Em termos de conceitos essenciais para aprender TypeScript (TS). Dominar esses conceitos irá ajudá-lo a escrever código TypeScript limpo, seguro e eficiente. Aqui estão alguns fundamentos importantes:

- **Tipagem Estática:** TypeScript é conhecido por sua tipagem estática *opcional*. Isso significa que você pode definir tipos para variáveis, parâmetros de função, propriedades de objeto e outros elementos do seu código. Compreender como usar e aproveitar os tipos estáticos é fundamental para escrever código TypeScript eficaz;
- **Interfaces e Tipos:** TypeScript oferece uma variedade de maneiras de definir tipos, incluindo interfaces. As interfaces são especialmente úteis para descrever a forma de objetos, enquanto os tipos podem ser usados para criar tipos de união, tipos condicionais e outros tipos complexos;
- **Inferência de Tipos:** TypeScript é capaz de inferir tipos automaticamente em muitos casos. Isso significa que você nem sempre precisa especificar os tipos explicitamente; o TypeScript pode deduzir os tipos com base no contexto do seu código;
- **Classes e Herança:** TypeScript suporta classes e herança, permitindo a criação de hierarquias de classes e a reutilização de código por meio da herança.

# Typescript: principais conceitos

- **Enums:** Enums, ou enumerações, são uma maneira de definir um conjunto nomeado de constantes. Eles podem ser úteis para tornar o código mais legível e mais fácil de manter;
- **Decoradores:** Decoradores são uma característica avançada do TypeScript que permite adicionar metadados e funcionalidades extras a classes e membros de classe;
- **Genéricos:** Genéricos permitem escrever código que pode funcionar com uma variedade de tipos diferentes. Eles são frequentemente usados em bibliotecas e estruturas de dados para criar componentes reutilizáveis e flexíveis;
- **Módulos:** TypeScript suporta o uso de módulos para organizar e reutilizar código. Compreender como importar e exportar módulos é importante para trabalhar com bibliotecas e projetos grandes;

# Typescript: principais conceitos

- **Definição de Tipos (Type Definitions):** As definições de tipos são arquivos ou pacotes que fornecem informações sobre a forma de objetos, tipos de retorno de função e outras informações de tipo para bibliotecas JavaScript que não foram originalmente escritas em TypeScript. Isso permite que os desenvolvedores usem essas bibliotecas JavaScript em projetos TypeScript sem perder o benefício da tipagem estática. As definições de tipos geralmente têm a extensão ".d.ts" e podem ser escritas manualmente ou geradas automaticamente por ferramentas como o @types/ do npm.

# Typescript: definição de tipos

**Exemplo:** Digamos que você tenha uma biblioteca JavaScript chamada "math-library.js", que contém uma função chamada "add" para adicionar dois números e agora, você deseja usar essa biblioteca em um projeto TypeScript.

Primeiro, você precisa criar uma definição de tipo para a biblioteca. Você pode fazer isso criando um arquivo "math-library.d.ts" com as informações de tipo para a biblioteca.

Então, você pode usar a biblioteca "math-library.js" em seu projeto TypeScript. Suponha que você tenha um arquivo TypeScript chamado "app.ts" onde você quer usar a função "add".

math-library.js

```
function add(a, b) {
  return a + b
}
```

math-library.d.ts

```
declare function add(a:
number, b: number): number
```

app.ts

```
import { add } from
'./math-library'

const result = add(10, 20)
console.log(result) // Saída:
30
```

# Typescript: .ts e .tsx

- **.ts (TypeScript):** Arquivos com a extensão ".ts" são usados para escrever código TypeScript puro, ou seja, código TypeScript que não contém marcação JSX (JavaScript XML). Esses arquivos são usados principalmente para escrever lógica de negócios, classes, funções, interfaces e outros elementos do código TypeScript.
- **.tsx (TypeScript com JSX):** Arquivos com a extensão ".tsx" são usados quando você está escrevendo código TypeScript que inclui marcação JSX. JSX é uma extensão de sintaxe que permite escrever elementos de interface do usuário de forma declarativa dentro do código JavaScript ou TypeScript. É comumente usado com bibliotecas como React para descrever a estrutura do componente. Portanto, se você estiver trabalhando com React ou qualquer outra biblioteca que utilize JSX, você usará arquivos com a extensão ".tsx" para seus componentes.

# Typescript: Sintaxe

# Variáveis tipadas

Em TypeScript, você pode declarar o tipo de uma variável explicitamente.

TYPESCRIPT

```typescript
let nome: string = "João"
let idade: number = 25
const isAdulto: boolean = true
let numeros: number[] = [1, 2, 3, 4]
let pessoa: { nome: string, idade: number } = { nome: "Maria", idade: 30 }
```

# Tipos primitivos

Strings, números, booleanos, etc.

TYPESCRIPT

```typescript
let nome: string = "João"
let idade: number = 25
const isAdulto: boolean = true
```

# Inferência de tipos

O TypeScript pode inferir tipos automaticamente com base no valor atribuído à variável.

TYPESCRIPT

```
let nome = "João" // automaticamente inferido como string
let idade = 25 // automaticamente inferido como number
```

# Tipos de Array e Objeto

Você pode declarar arrays e objetos com tipos específicos.

TYPESCRIPT

```typescript
let numeros: number[] = [1, 2, 3, 4]
let pessoa: { nome: string, idade: number } = { nome: "Maria", idade: 30 }
```

# Tipos de Função

O TypeScript permite tipar os parâmetros e o retorno das funções.

TYPESCRIPT

```typescript
function soma(a: number, b: number): number {
    return a + b
}
let resultado: number = soma(3, 4)
// resultado = 7

// Arrow function com tipagem explícita
const dobrar = (numero: number): number => {
    return numero * 2
}

// Arrow function com inferência de tipo
const quadrado = (numero: number) => numero ** 2
```

# Interfaces

Interfaces são usadas para definir a estrutura de objetos.

TYPESCRIPT

```typescript
// Definição da interface
interface Pessoa {
    nome: string
    idade: number
}

// Função que recebe um objeto do tipo Pessoa
function saudar(pessoa: Pessoa) {
    console.log(`Olá, ${pessoa.nome}! Você tem ${pessoa.idade} anos.`)
}

// Objeto que segue a estrutura da interface Pessoa
let joao: Pessoa = {
    nome: 'João',
    idade: 30
}

// Chamando a função com o objeto
saudar(joao)
```

# Tipos Personalizados

O TypeScript permite a criação de tipos personalizados usando a palavra-chave type.

TYPESCRIPT

```typescript
// Definição do tipo personalizado
type Coordenadas = [number, number]

// Função que recebe um parâmetro do tipo Coordenadas
function mostrarCoordenadas(coordenadas: Coordenadas) {
    console.log(`Latitude: ${coordenadas[0]}, Longitude: ${coordenadas[1]}`)
}

// Chamando a função com um array de números
mostrarCoordenadas([40.7128, -74.0060])
```

# Tipos e Interface

No TypeScript, tanto as interfaces quanto os tipos podem ser usados para definir a estrutura de um objeto. A principal diferença entre eles é que as interfaces são mais adequadas para definir a forma de objetos enquanto os tipos podem ser usados para definir tipos que não são apenas objetos, como union types, intersection types, entre outros.

TYPESCRIPT

```typescript
// Definição da interface Pessoa
interface Pessoa {
    nome: string
    idade: number
}

// Utilização da interface
let pessoa: Pessoa = {
    nome: 'João',
    idade: 30
}

console.log(pessoa) // Saída: { nome: 'João', idade: 30 }
```

TYPESCRIPT

```typescript
// Definição do tipo Pessoa
type PessoaType = {
    nome: string
    idade: number
}

// Utilização do tipo
let pessoa: PessoaType = {
    nome: 'Maria',
    idade: 25
}

console.log(pessoa) // Saída: { nome: 'Maria', idade: 25
}
```

# Union Types

- Union types permitem que uma variável ou parâmetro de função aceite valores de diferentes tipos;
- Eles são definidos usando o operador de pipe | entre os tipos;
- Isso é útil quando você precisa que uma variável possa conter mais de um tipo de valor.

TYPESCRIPT

```typescript
// Definindo um tipo que pode ser uma string ou um número
type ID = string | number

// Variável do tipo ID que pode conter uma string ou um número
let identificador: ID
identificador = '123' // OK
identificador = 456   // OK
identificador = true  // Erro: Tipo 'boolean' não é atribuível ao tipo 'string | number'
```

# Intersection Types

- Intersection types permitem combinar vários tipos em um só, criando um novo tipo que possui todas as propriedades de cada tipo original;
- Eles são definidos usando o operador de interseção & entre os tipos;
- Isso é útil quando você precisa combinar as propriedades de vários tipos em um único tipo.

TYPESCRIPT

```typescript
// Definindo dois tipos com propriedades diferentes
interface Animal {
    nome: string
    tipo: string
}

interface Mamifero {
    gestacao: boolean
}

// Criando um novo tipo que contém as propriedades de ambos os tipos
type AnimalMamifero = Animal & Mamifero

// Objeto que segue o tipo AnimalMamifero
let cachorro: AnimalMamifero = {
    nome: 'Rex',
    tipo: 'cachorro',
    gestacao: false
}
```

# ENUNS

O TypeScript oferece suporte a enums, que são uma forma de definir um conjunto de valores nomeados.

TYPESCRIPT

```typescript
enum DiaDaSemana {
    Segunda,
    Terca,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo
}

let dia: DiaDaSemana =
DiaDaSemana.Quarta
```

# Generics

O TypeScript suporta generics, permitindo criar componentes reutilizáveis que funcionam com vários tipos de dados.

**Sintaxe de Generics:**

- A sintaxe básica de generics usa parâmetros de tipo, que são representados entre colchetes angulares (< >).
- Esses parâmetros de tipo são usados como marcadores de posição para os tipos de dados que serão fornecidos quando a função ou componente for chamado.

**Neste exemplo:**

- <T> é o parâmetro de tipo genérico.
- arg: T indica que o parâmetro arg e o tipo de retorno são do mesmo tipo T.
- Na chamada da função, o tipo T é especificado explicitamente (number) ou inferido implicitamente (string).

TYPESCRIPT

```typescript
function identidade<T>(arg: T): T {
    return arg
}

let numero: number = identidade<number>(5) // especificando o tipo
let texto: string = identidade('Olá') // inferindo o tipo

console.log(numero) // Saída: 5
console.log(texto) // Saída: Olá
```

# Decorators

O TypeScript introduz o conceito de decorators, que são uma forma de modificar ou adicionar metadados a classes, métodos, propriedades e parâmetros de função.

TYPESCRIPT

```typescript
function logar(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const metodoOriginal = descriptor.value

    descriptor.value = function(...args: any[])
{
        console.log(`Chamando método ${propertyKey}`)
        return metodoOriginal.apply(this, args)
    }

    return descriptor
}

class MinhaClasse {
    @logar
    minhaFuncao() {
        console.log('Executando minha função ')
    }
}

const instancia = new MinhaClasse()
instancia.minhaFuncao()
```

# Dúvidas?

# Antes do ReactJS e NextJS

**Principais Conceitos em TypeScript (TS):**

- **Tipagem Estática:** Compreender como utilizar tipos estáticos para declarar variáveis, parâmetros de função, propriedades de objetos, entre outros;
- **Interfaces e Tipos Personalizados:** Saber como definir interfaces para descrever a estrutura de objetos e criar tipos personalizados;
- **Enums:** Entender como utilizar enums para definir conjuntos de valores nomeados;
- **Generics:** Compreender o uso de generics para criar componentes reutilizáveis que funcionam com vários tipos de dados;
- **Decorators:** Conhecer o conceito de decorators e como usá-los para modificar ou adicionar metadados a classes, métodos, propriedades e parâmetros de função.

# Referências

TypeScript Documentation, 2024. Disponível em: <https://www.typescriptlang.org/docs/>

# Cursos recomendados

UDEMY: Curso de JavaScript e TypeScript do básico ao avançado JS/TS

DIO: Formação TypeScript Fullstack Developer