



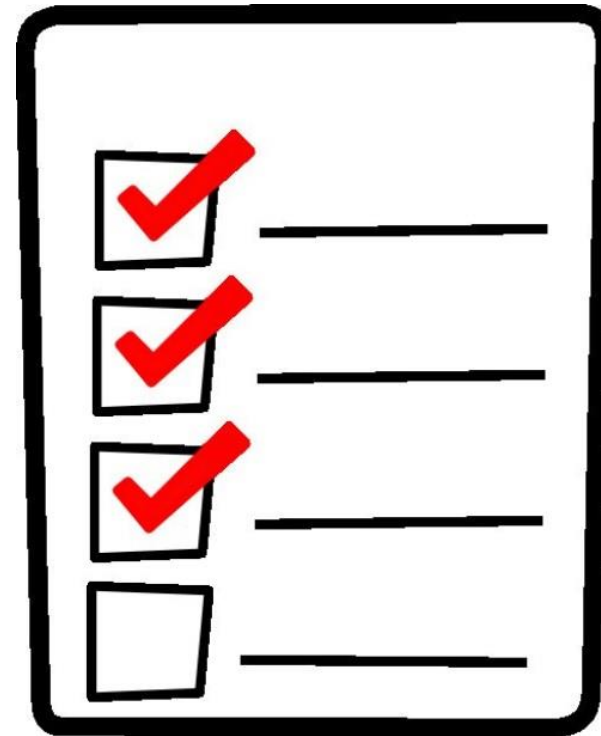
# Iniciando no TypeScript



# Conteúdo

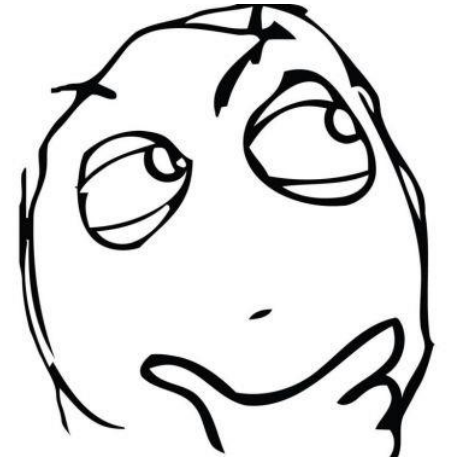
Sobre o que vamos falar hoje?

- O que é Typescript?
- Por que usar?
- Desvantagens
- Tipos primitivos/básicos
- Recursos mais abrangentes
- Projetinho prático
- Dúvidas



# O que é Typescript?

- É um superset para Javascript desenvolvido pela Microsoft.
- Adiciona recursos de tipagem estática opcionais e outros recursos à linguagem Javascript.
- *Basicamente, permite que os desenvolvedores escrevam código Javascript de forma mais estruturada e com menos erros, implementando tipagens e melhorando a consistência dos dados.*



Bill Gates ->



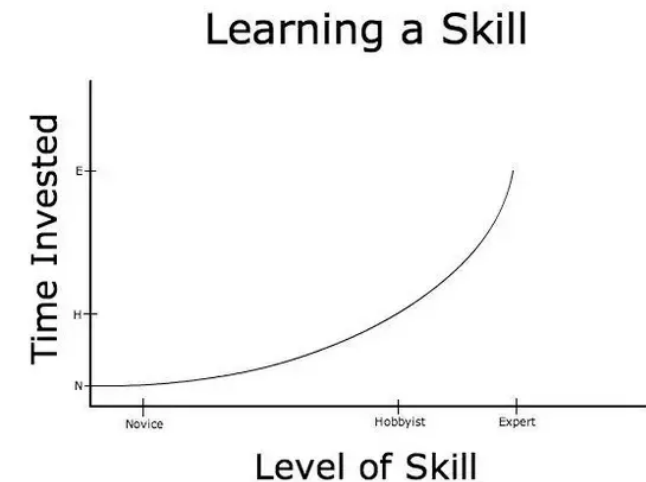
# Por que usar TypeScript em vez de JavaScript puro?

1. **Tipagem Estática Opcional:** permite adicionar tipos estáticos opcionais para detectar erros de tipo durante o desenvolvimento.
2. **Melhor Suporte a Ferramentas de Desenvolvimento:** oferece recursos como autocompletar, refatoração e verificação de erros em tempo real, melhorando a eficiência do desenvolvimento.
3. **Manutenção de Código:** facilita a compreensão e alteração do código existente, ajudando a entender a estrutura do código.
4. **Compatibilidade com JavaScript:** por ser um superset de JavaScript, pode-se utilizar código JavaScript existente em projetos TypeScript, facilitando a transição.
5. **Ecosistema Forte:** é amplamente adotado pela comunidade de desenvolvimento, com suporte oficial ou não oficial em muitas bibliotecas e estruturas populares.

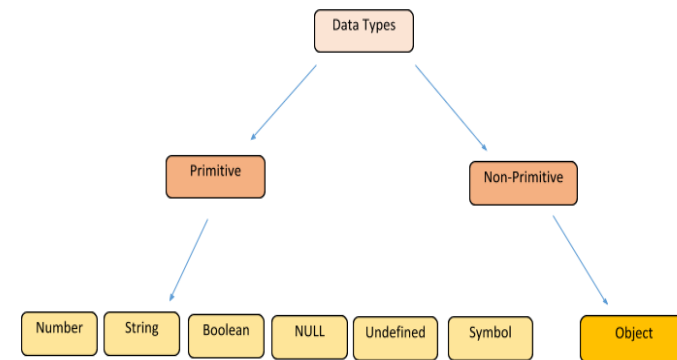


# “Desvantagens” de se utilizar TypeScript

1. **Curva de aprendizado:** Pode ser mais difícil para quem não está acostumado;
2. **“Menor flexibilidade”:** Limita a flexibilidade do código por conta da tipagem;
3. **“Maior complexidade”:** O código fica mais complexo por conta das tipagens, se não usadas corretamente;
4. **Maior uso de memória:** Requer mais memória do que Javascript, pois precisa armazenar informações adicionais sobre tipos etc.



# TypeScript: tipos básicos (ou primitivos)



Tipos primitivos ou básicos em TypeScript são recursos fundamentais para a definição de variáveis e estruturas de dados. Eles representam os tipos de dados mais simples e diretos que podem ser manipulados em um programa TypeScript.

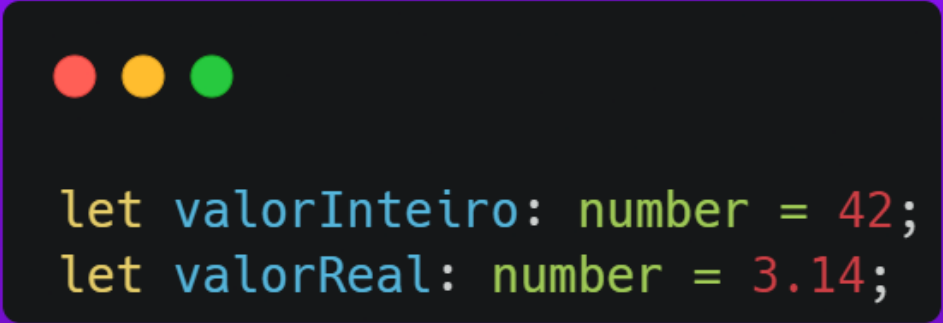
Os tipos primitivos mais utilizados, são:

- **number**: Representa valores numéricos, incluindo inteiros e decimais/reais.
- **string**: Representa valores de texto, como palavras, frases ou caracteres.
- **boolean**: Representa valores lógicos, que podem ser verdadeiro (true) ou falso (false).
- **null** e **undefined**: Representam valores nulos e indefinidos, respectivamente.
- **void**: Representa a não existência de um retorno para uma função.



# Number

O tipo **number** é utilizado para representar valores numéricos, como inteiros e decimais.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of code.

```
let valorInteiro: number = 42;  
let valorReal: number = 3.14;
```



# Boolean

O tipo **boolean** é utilizado para representar valores verdadeiros ou falsos.



```
let verdadeiro: boolean = true;  
let falso: boolean = false;
```

# String

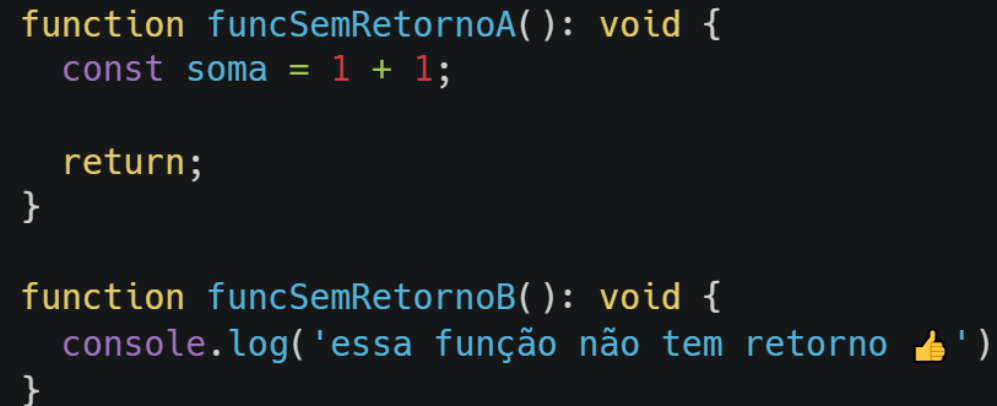
O tipo **string** é utilizado para representar valores que sejam textos.



```
let nome: string = 'Mike Baguncinha';
```

# Void

O tipo **void** é utilizado para representar o valor de retorno de funções que não possuem retorno.



```
function funcSemRetornoA(): void {  
  const soma = 1 + 1;  
  
  return;  
}  
  
function funcSemRetornoB(): void {  
  console.log('essa função não tem retorno 👍')  
}
```

# Null

- O tipo **null** é utilizado para representar ausência intencional de valor ou objeto.
- Quando uma variável é atribuída como null, ela explicitamente não aponta para nenhum objeto ou valor válido.

```
class Node {  
  value: number;  
  next: Node | null;  
  
  constructor(value: number) {  
    this.value = value;  
    this.next = null;  
  }  
}  
  
class LinkedList {  
  head: Node | null;  
  
  constructor() {  
    this.head = null;  
  }  
  
  // ...  
}
```

# Undefined

O tipo **undefined** é utilizado para indicar que uma variável foi declarada, mas ainda não recebeu um valor.



# 0 (zero) x null x undefined

Resumindo a diferença entre 0, null e undefined.



**0**



**null**



**undefined**



# Arrays

Podemos criar arrays que armazenarão tipos diferentes de dados, como strings, numbers, booleans etc.

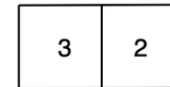
```
const numeros: number[] = [1,2,3,4];

const nomes: string[] = ['Ronaldo Gaúcho',
  'Relâmpago Marquinhos', 'Tony, o Tigre da marca de
  sucrilhos', 'Goku', 'Bananas de Pijama'],

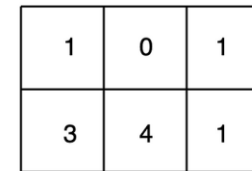
const booleans: boolean[] = [true, false, false,
  false]

const objetos: {nome: string, idade: number}[] = [{
  nome: 'Shrek',
  idade: 50
}]
```

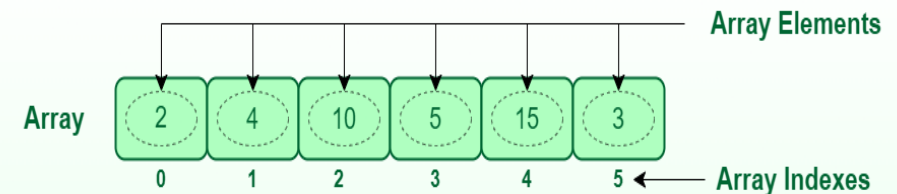
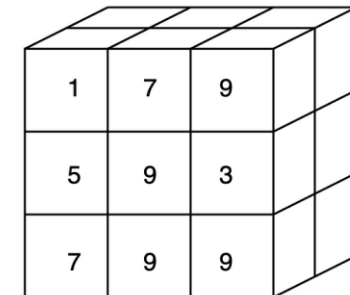
1D Array



2D Array



3D Array



# Objects

Objetos são estruturas de dados que armazenam valores em esquema chave-valor.



```
const objeto: { nome: string, idade: number } = {  
  nome: 'Julian',  
  idade: 23  
}
```

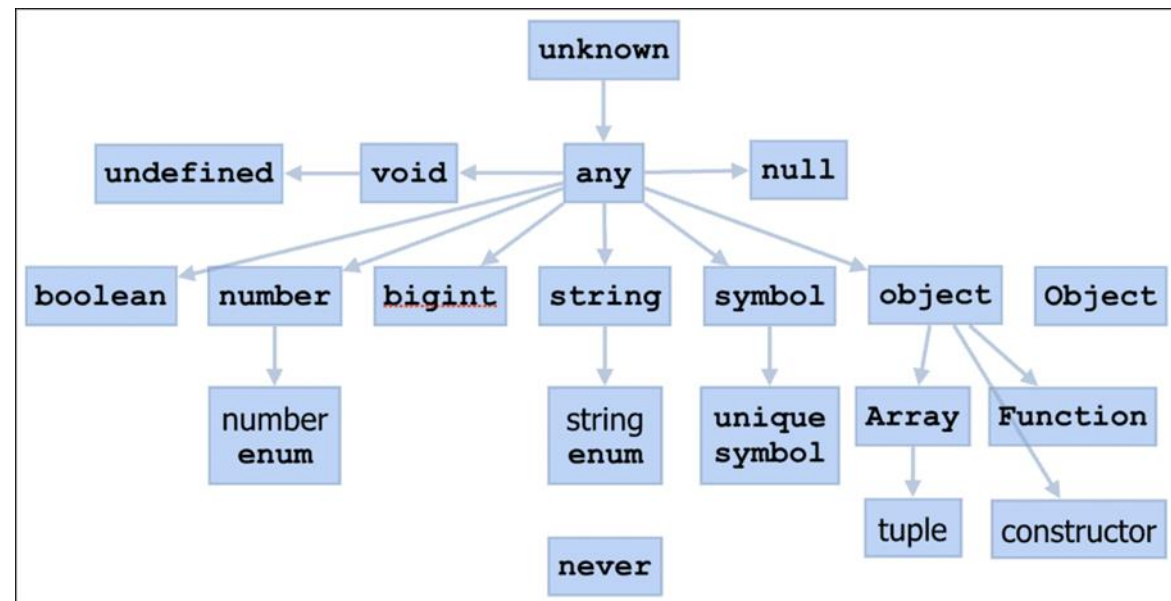


Mas mais do que isso, objetos são estruturas de dados que combinam dados e operações (ou funcionalidades) associadas a esses dados em uma única unidade. Eles permitem modelar entidades complexas e abstratas de maneira mais organizada e eficiente.

```
const pessoa: {  
  nome: string,  
  idade: number,  
  getNome: () => string,  
  getIdade: () => number  
} = {  
  nome: 'Smzinho',  
  idade: 32,  
  getNome: () => {  
    return pessoa.nome  
  },  
  getIdade: () => {  
    return pessoa.idade  
  }  
}
```



# Typescript: recursos mais abrangentes



O TypeScript oferece uma gama abrangente de recursos que proporcionam maior flexibilidade e ferramentas para aprimoramento do código durante o processo de desenvolvimento.

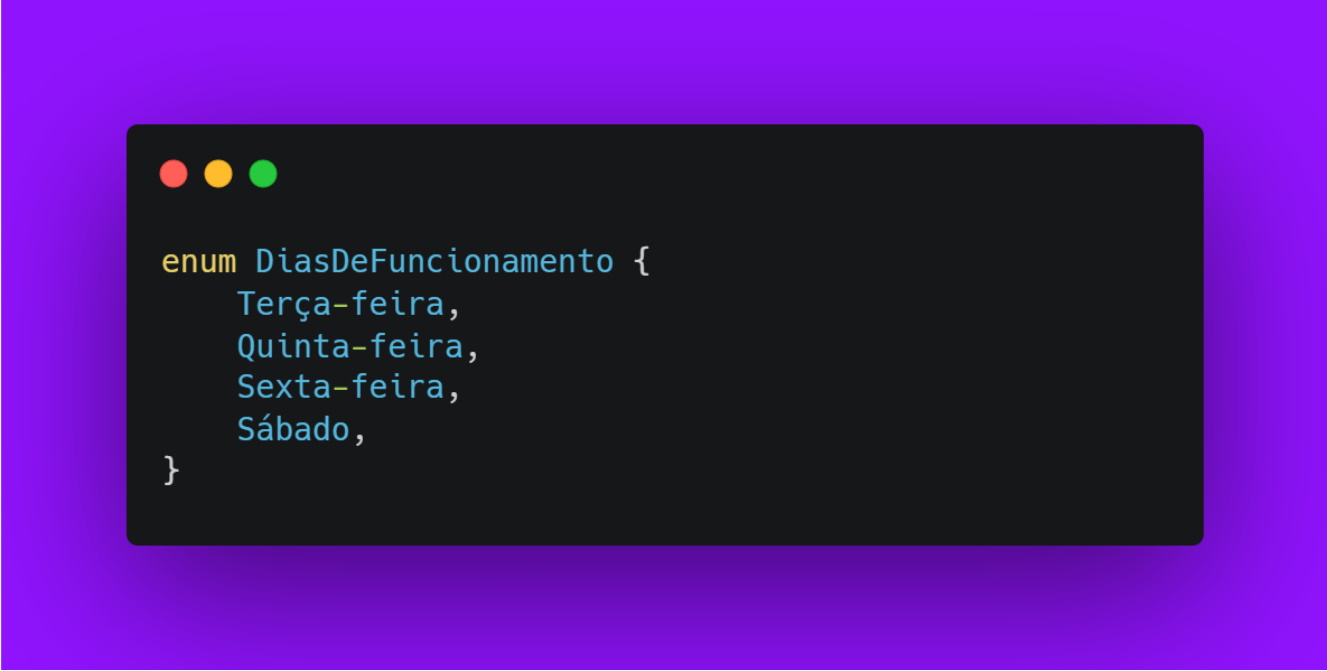
Abordaremos os seguintes recursos:

- Enums/constantes.
- Union types e type aliases.
- Funções
- Interfaces e Types
- Generics
- Any type



# Enums

Permitem definir um conjunto nomeado de constantes.  
Eles são úteis quando você tem um conjunto fixo de valores que uma variável pode ter.



```
enum DiasDeFuncionamento {  
    Terça-feira,  
    Quinta-feira,  
    Sexta-feira,  
    Sábado,  
}
```

# Union types

Permitem que uma variável possa ter mais de um tipo. Isso é útil quando uma variável pode armazenar diferentes tipos de valores



```
let cpf: string | number = '175.661.550-08';
```

```
cpf = 17566155008
```

# Type aliases

Permitem que você defina um nome (ou apelido) para um tipo existente. Isso é útil para criar tipos complexos ou para dar nomes mais descritivos a tipos existentes.

```
type UserCPF = string | number;  
let cpf: UserCPF = '175.661.550-08';  
cpf = 17566155008
```

```
type UserOrders = User & Orders;  
const retrievedUserOrders: UserOrders = {...};
```

# Interfaces

É uma ferramenta poderosa para definir contratos ou estruturas de objetos em uma aplicação.

Com interfaces podemos “tipar” variáveis, objetos, parâmetros de funções, retornos de funções etc.

```
interface Pedido {  
  id: number;  
  id_produto: number;  
  valor: number;  
  data: Date;  
}  
  
interface Pessoa {  
  id: number;  
  nome: string;  
  cpf: number | string;  
  email: string;  
  pedidos: Pedido[];  
}  
  
const pessoa: Pessoa = {  
  id: 1,  
  nome: 'Comprador A',  
  email: 'comprador_a@email.com',  
  cpf: '111.222.333-44',  
  pedidos: [{  
    id: 1,  
    data: new Date('2024-02-29'),  
    id_produto: 1,  
    valor: 5000  
  }]  
}
```

# Type

É uma ferramenta poderosa que permite criar tipos personalizados, fornecendo flexibilidade e legibilidade ao código.

*Essencialmente, ele permite definir aliases (apelidos) para tipos existentes ou criar novos tipos com base em tipos existentes.*

```
type Usuario = {  
  nome: string;  
  idade: number;  
};  
  
type Coordenada = [number, number];  
  
type Ponto = {  
  x: number;  
  y: number;  
};  
  
type ComCor = Ponto & { cor: string };
```



# Interface x Type (e agora, qual utilizar?)

As principais diferenças entre **Interface** e **Type**, são:

1. Herança:
  - **interface** suporta herança, permitindo estender uma interface por outra.
  - **type** não suporta herança.
2. Mesclagem:
  - **interface** oferece mesclagem automática quando você declara uma nova interface com o mesmo nome de uma existente.
  - **type** não oferece mesclagem automática; a declaração de um novo tipo com o mesmo nome sobrescreve o tipo existente.
3. Tipos complexos:
  - **type** é mais utilizado para criar tipos mais complexos com maior facilidade.



# Generics (tipos genéricos)

Generics em TypeScript oferecem uma maneira poderosa de criar componentes e estruturas de dados que são flexíveis e reutilizáveis com diferentes tipos de dados.

Eles permitem escrever código que pode trabalhar com uma variedade de tipos sem perder segurança de tipo.

```
class Pilha<T> {  
    private elementos: T[] = [];  
  
    push(elemento: T) {  
        this.elementos.push(elemento);  
    }  
  
    pop(): T | undefined {  
        return this.elementos.pop();  
    }  
}
```



# Functions

Funções são elementos importantes no mundo da programação, pois permitem que você agrupe blocos de código em unidades reutilizáveis e modulares.

Em Typescript, podemos:

- “tipar” parâmetros de funções;
- “tipar” retornos de funções;
- criar funções como tipos.



```
function saudacao(nome: string) {  
    console.log(`Olá, ${nome}!`);  
}  
  
function soma(a: number, b: number): number {  
    return a + b;  
}  
  
type OperacaoMatematica = (x: number, y: number) => number;  
  
const subtracao: OperacaoMatematica = (a, b) => a - b;  
const multiplicacao: OperacaoMatematica = (a, b) => a * b;
```

Além disso, podemos passar funções como parâmetros para outras funções, permitindo uma maior flexibilidade e reutilização de código.  
Isso é útil quando precisamos definir um comportamento específico que pode variar dependendo da situação.

```
type OperacaoMatematica = (a: number, b: number) => number;

const somar: OperacaoMatematica = (a: number, b: number) => a + b;

const subtrair: OperacaoMatematica = (a: number, b: number) => a - b;

function executarOperacao(a: number, b: number, operacao: OperacaoMatematica): number
{
    return operacao(a, b);
}
```



# Any type

É um tipo especial que representa **qualquer tipo de valor**.

*Ele é usado quando o tipo de uma variável é desconhecido ou quando você está lidando com valores de tipos variados que não têm uma estrutura de tipo consistente.*



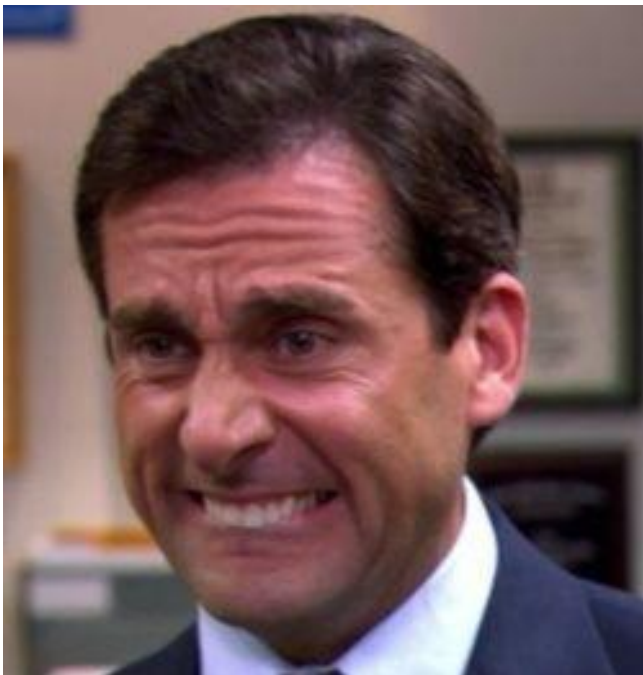
```
let valor: any;  
  
valor = 5;  
console.log(valor.toFixed(2));  
  
valor = "Olá";  
console.log(valor.length);
```



Na teoria é tudo bonito, é um paraíso, é uma paz...



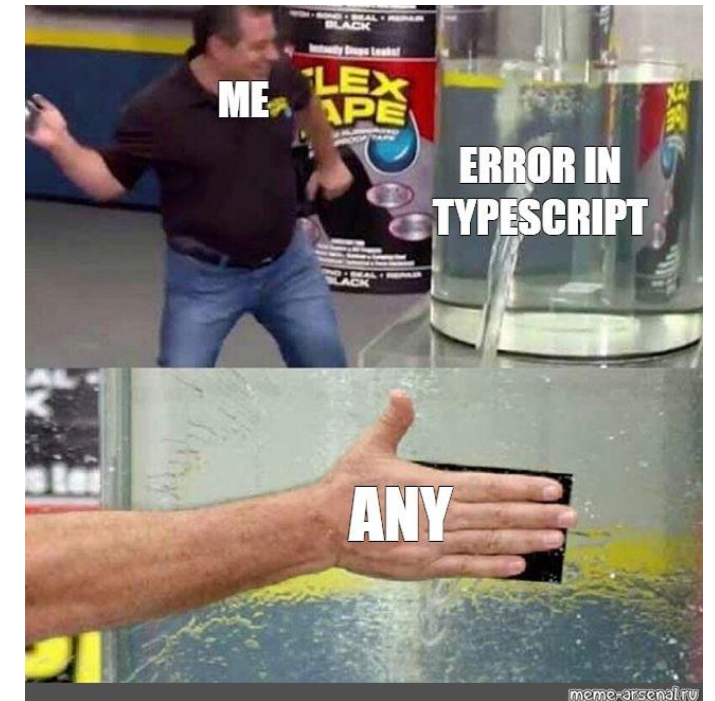
# Mas na prática...





Muitas das vezes o **any** é utilizado da maneira errada, como se fosse um analgésico.

O uso errado do **any** pode causar inconsistências no código e também futuros erros, pois perde-se a segurança dos tipos, uma vez que o compilador não realiza verificação de tipos em valores do tipo **any**.





## A má utilização do **any** e seus efeitos...

```
function multiplicadorDeGanhos(valorJogado: any, coeficiente: any): any {  
  const bonusExtra = 0.5 * coeficiente;  
  const valorMultiplicado = valorJogado * coeficiente;  
  const valorFinal = bonusExtra + valorMultiplicado;  
  
  return valorFinal;  
}  
  
const valorJogado = 5;  
let coeficiente;  
const resultado = multiplicadorDeGanhos(valorJogado, coeficiente);
```



# Dúvidas?



Muito obrigado pela atenção!

