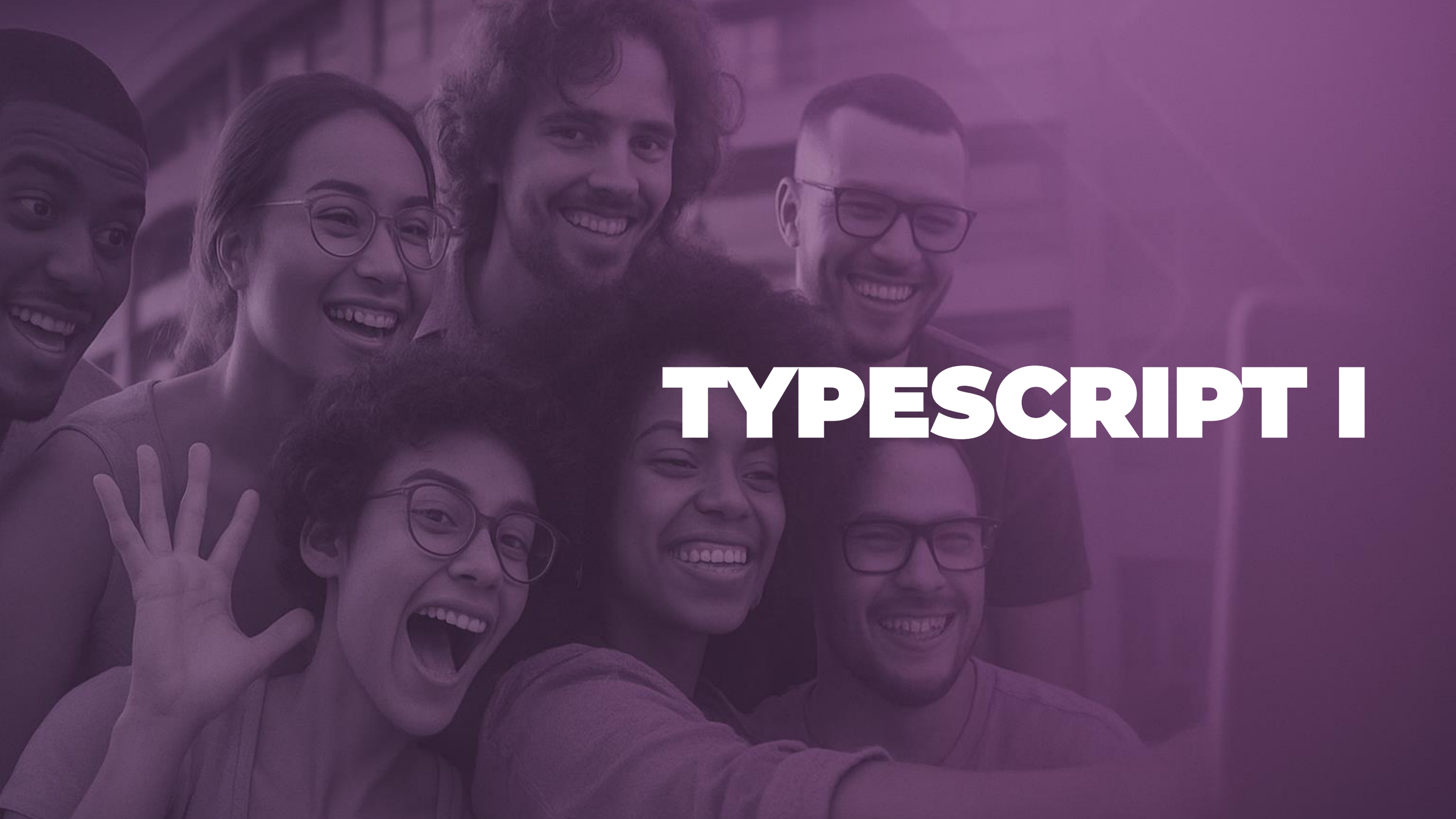
KORU


DESENVOLVE
programa de formação
e inclusão de talentos
em tecnologia
Grupo
Boticário
KORU



TYPESCRIPT I

ONDE ESTAMOS?

- O que é **TypeScript** e por que usá-lo
- **Vantagem** em relação ao JavaScript puro
- **Tipagem Básica** (Primitivos, Inferência vs Explícita, any, unknown, null, undefined)
- **Variáveis** e **Funções** Tipadas
- **Tipos Compostos** (Arrays, Tuplas, Objetos, Type Aliases)
- **Union** e **Literal Types**

O QUE É TYPESCRIPT

O QUE É TYPESCRIPT

TypeScript é um Superset do JavaScript

- Significa que ele adiciona funcionalidades ao JS.
- Todo código JavaScript válido é (quase sempre) um código TypeScript válido!

Adiciona Tipagem Estática Opcional

- Podemos (e devemos!) dizer ao código qual tipo de dado esperamos.

É desenvolvido e mantido pela Microsoft.

TYPESCRIPT = JS + SUPERPODERES 🦸‍♂️

Pense no TypeScript como o JavaScript, mas com ferramentas extras que te ajudam a escrever código mais robusto e fácil de manter.

Esses superpoderes vêm na forma de **tipos**.

POR QUE USAR?

POR QUE USAR?

Encontrar Erros Cedo: O principal benefício!

- Identifica muitos erros durante o desenvolvimento (no seu editor), não só quando você roda o código.

Código Mais Claro: Os tipos servem como documentação. Fica fácil entender o que uma função espera ou retorna.

Refatoração Mais Segura: Mudar o código existente fica menos arriscado, pois o TS te avisa se algo quebrou por causa dos tipos.

Melhor Suporte de Ferramentas: O VS Code (e outros editores) entendem TS perfeitamente, oferecendo autocompletar inteligente, navegação, etc.

COMPILAÇÃO

O PROCESSO DE COMPILAÇÃO

Navegadores e Node.js entendem JavaScript (.js). Eles não entendem TypeScript (.ts) diretamente.

Precisamos "traduzir" o código TS para JS. Isso se chama Compilação.

Usamos o **compilador TypeScript** (tsc).

- *tsc seu_arquivo.ts → gera seu_arquivo.js*

No nosso dia a dia, ferramentas como Vite ou Next.js fazem essa compilação automaticamente para nós.

TIPAGEM BÁSICA

TIPAGEM BÁSICA

Os tipos mais simples e fundamentais:

string: Texto (ex: "olá", "TypeScript é legal")

number: Números (ex: 10, 3.14, -5)

boolean: Valores lógicos (ex: true, false)

```
let userName: string = 'Carlos'

let age: number = 30;

let isActive: boolean = true;

const PI: number = 3.14159;
```

TIPAGEM EXPLÍCITA VS INFERÊNCIA

Tipagem Explícita: Você diz qual é o tipo.

Inferência de Tipo: O TypeScript descobre o tipo com base no valor inicial.

Na dúvida, use inferência.

```typescript
let quantity: number = 10;
let message: string; // Declarado sem valor inicial, precisa do tipo


let price = 19.99; // TS infere que é `number`
const welcomeMessage = "Olá!"; // TS infere que é `string`
let isReady = false; // TS infere que é `boolean`
```

TIPOS ESPECIAIS - ANY

any: O tipo mais flexível (e perigoso!).

Desliga a checagem de tipo para a variável.

Regra de Ouro: **Evite any sempre que possível**! Ele nega os benefícios do TypeScript.

```
let dados: any = "qualquer coisa";

dados = 123; // tudo vale...

dados = { nome: "Zé" }; // ... tudo pode!
```

TIPOS ESPECIAIS - NULL E UNDEFINED

null: Indica a ausência intencional de um valor. Algo deveria estar lá, mas não está.

undefined: Indica que uma variável foi declarada, mas nunca teve um valor atribuído, ou que uma propriedade de objeto não existe.

Em TypeScript, eles são tipos distintos.

```
let variableA: null = null;

let variableB: undefined
```

FUNÇÕES

FUNÇÕES TIPADAS - PARÂMETROS

Adicionamos **tipos aos parâmetros** esperados pela função. Isso garante que a função seja chamada com os tipos de dados corretos

```ts
function greet(name: string, age: number): void { // Tipos dos parâmetros
    console.log(`Olá, ${name}! Você tem ${age} anos.`);
}

greet("João", 28); // OK
// greet("Maria", "trinta"); // ERRO no TS!
// greet("Pedro"); // ERRO no TS! (faltou parâmetro)
```

FUNÇÕES TIPADAS - RETORNO

Adicionamos `: TipoDeRetorno` antes das chaves `{}` da função.

void é usado para funções que não retornam explicitamente um valor

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

let result: number = sum(5, 3); // OK

function logAction(mensagem: string): void { // Tipo de retorno: void (não retorna nada)
    console.log(mensagem);
    // return "algo"; // ERRO no TS!
}
```

FUNÇÕES ANÔNIMAS E ARROW FUNCTIONS TIPADAS

As regras de tipagem são as mesmas!

TypeScript frequentemente consegue inferir os tipos de parâmetros em funções anônimas se ele sabe de onde a função vem (ex: um callback de forEach).

Mas você pode tirar explicitamente para clareza.

```
const multiply = (x: number, y: number): number => { // Arrow function
    return x * y;
};

const numbers = [1, 2, 3];
numbers.forEach((num: number) => { // Função anônima tipada
    console.log(num * 2);
});
```

PARÂMETROS OPCIONAIS

Use ? após o nome do parâmetro para torná-lo opcional.

Se opcional, o parâmetro pode ser fornecido ou ser undefined.

O tipo de greeting é string | undefined

```
function greet(name: string, greeting?: string): void {
    if (greeting) {
        console.log(`${greeting}, ${name}!`);
    } else {
        console.log(`Olá, ${name}!`);
    }
}

greet("Ana"); // OK
greet("Carlos", "Bom dia"); // OK
// greet("Diana", 123); // ERRO no TS!
```

PARÂMETROS COM VALOR PADRÃO

Você pode definir um valor padrão para um parâmetro usando =.

Se um valor padrão é fornecido, o parâmetro é automaticamente opcional (mas você pode deixá-lo explícito com ? se quiser).

O tipo de `prefix` aqui é inferido como string

```
function showInfo(item: string, prefix: string = "Item:"): void {
    console.log(`${prefix} ${item}`);
}

showInfo("Caneta"); // Saída: "Item: Caneta"
showInfo("Livro", "Produto:"); // Saída: "Produto: Livro"
```

RECAP - VARIÁVEIS E FUNÇÕES

```ts
// variáveis
let variable1: string;

// Função
function function1(param1: string, optionalParam?: string): void {}

// Arrow function
const function2 = (
  param1: string,
  paramWithDefault: string = 'default'
): void => {};
```

TIPOS COMPOSTOS

TIPOS COMPOSTOS

Tipos Compostos são estruturas que agrupam múltiplos valores:

- Arrays
- Tuplas
- Objetos

```
// Tuplas
let tuple: [string, number] = ['João', 30]; // Tupla com string e number
let tuple2: [string, number][] = [
  ['Maria', 25],
  ['Pedro', 40],
  ['Ana', 35],
]; // Array de tuplas
```

```
// Arrays
let numbersArray: number[] = [1, 2, 3, 4, 5];
let stringsArray: Array<string> = ['a', 'b', 'c'];
let mixedArray: (number | string)[] = [1, 'a', 2, 'b'];
let anyArray: any[] = [1, 'a', true, { name: 'Zé' }];
```

```
// Objetos
// Objeto com propriedades nome e idade
let person: { name: string; age: number } = {
  name: 'Carlos',
  age: 30,
};

// Array de objetos
let person2: { name: string; age: number }[] = [
  { name: 'Maria', age: 25 },
  { name: 'Pedro', age: 40 },
  { name: 'Ana', age: 35 },
];

// Objeto com propriedades opcionais
let person3: { name: string; age?: number } = {
  name: 'Carlos',
  // age: 30, // idade é opcional
};
```

ARRAYS TIPADOS

Coleções de valores do mesmo tipo.

Sintaxe comum: Tipo[]

Sintaxe alternativa (menos comum): Array<Tipo>

```
// Arrays
let numbersArray: number[] = [1, 2, 3, 4, 5];
let stringsArray: Array<string> = ['a', 'b', 'c'];
let mixedArray: (number | string)[] = [1, 'a', 2, 'b'];
let anyArray: any[] = [1, 'a', true, { name: 'Zé' }];
```

TUPLAS

Tuplas são arrays com:

- **Número fixo de elementos**.
- **Tipos específicos em posições específicas**.

Útil para pares ordenados (ex: coordenadas [x, y], [id, nome] de um usuário específico).

```
let coordinates: [number, number];
coordinates = [10, 20]; // OK
// coordinates = [10, 20, 30]; // ERRO no TS! (tamanho fixo)
// coordinates = ["a", 20]; // ERRO no TS! (tipo errado na posição 0)

let userInfo: [number, string, boolean] = [1, "Alice", true];
console.log(userInfo[0]); // Tipo é number
console.log(userInfo[1]); // Tipo é string
// console.log(userInfo[3]); // ERRO no TS! (índice fora do limite)
```

OBJETOS COM TIPAGEM EXPLÍCITA

Definimos a estrutura do objeto
e os tipos de suas propriedades.

```
let book: { title: string, author: string, pages: number, published?: boolean };

book = {
    title: "O Senhor dos Anéis",
    author: "J.R.R. Tolkien",
    pages: 1178,
    published: true // Propriedade opcional
};

// book = { title: "1984", author: "Orwell" }; // ERRO no TS! (faltou 'pages')
// book.year = 1949; // ERRO no TS! ('year' não existe na definição)
```

TYPE ALIAS

TYPE ALIASES

Criamos um nome para uma definição de tipo (geralmente objetos ou uniões complexas).

Melhora a legibilidade e permite reutilizar a definição.

```
type Book = {
    title: string;
    author: string;
    pages: number;
    published?: boolean; // Opcional
};

let book1: Book = {
    title: "Fundação",
    author: "Isaac Asimov",
    pages: 244
};

let book2: Book = { // Reutilizamos o tipo!
    title: "Duna",
    author: "Frank Herbert",
    pages: 412,
    published: true
};
```

UNION E LITERAL TYPES

UNION TYPES

Uma variável ou parâmetro pode ser um de vários tipos possíveis.

Usamos o símbolo |.

```
let id: number | string; // Pode ser number OU string
id = 10;      // OK
id = "abc";   // OK
// id = true; // ERRO no TS!

function printID(id: number | string): void {
    console.log(`O ID é: ${id}`);

    // Para usar métodos específicos, precisamos verificar o tipo (Type Narrowing)
    if (typeof id === 'string') {
        console.log(id.toUpperCase()); // Agora sabemos que 'id' é string
    }
}
```

LITERAL TYPES

O **tipo é exatamente um valor específico** (uma string ou um número literal).

Útil para restringir valores possíveis.

```ts
let direction: "norte" | "sul" | "leste" | "oeste";
direction = "norte"; // OK
// direction = "cima"; // ERRO no TS!

let statusCode: 200 | 404 | 500;
statusCode = 200; // OK
// statusCode = 201; // ERRO no TS!
```

UNION + LITERAL TYPES: COMBINANDO

Podemos combinar tipos literais com outros tipos usando |.

```ts
type Status = "sucesso" | "erro" | number; // Pode ser "sucesso", "erro", ou qualquer number

let answer1: Status = "sucesso"; // OK
let answer2: Status = 404;       // OK
// let answer3: Status = true;   // ERRO no TS!
```

PRÁTICA 01.
UM GERENCIADOR SIMPLES DE TAREFA (CLI) UNION E LITERAL TYPES

PRÁTICA - GERENCIADOR DE TAREFAS

Objetivo: Criar um pequeno programa de linha de comando (CLI) em TypeScript que simula um gerenciador de tarefas simples. O programa deverá permitir adicionar tarefas e listar as tarefas existentes, utilizando os conceitos de tipagem aprendidos na aula.

Defina o Tipo de Tarefa. Uma tarefa deve ser um objeto que tenha um id, description e um status (pending ou completed).

Defina o tipo da Lista de Tarefas

Função para Adicionar Tarefas

Função para Listar Tarefas

Exiba os resultados usando muitos console.logs!

PRÁTICA - GERENCIADOR DE TAREFAS

Você, ao final poderá testar seu programa assim:

```
// Testando o programa
console.log('--- Gerenciador de Tarefas ---');

addTask('Comprar leite');
addTask('Estudar TypeScript');
addTask('Fazer exercício');

listTasks();
removeTask(1);
listTasks();

console.log('--- Fim ---');
```

RESUMO

RESUMO

TypeScript adiciona **tipagem ao JavaScript** para mais segurança e clareza.

Tipos Primitivos: string, number, boolean.

Inferência vs. Explícito: TS pode descobrir tipos, mas tipar manualmente ajuda na clareza.

any (evitar!), **unknown** (seguro, exige checagem).

Tipamos **variáveis**, **parâmetros** e **retornos de funções**.

Arrays (Tipo[]), **Tuplas** ([T1, T2]), **Objetos** ({ p: T }) são **tipos compostos**.

type aliases ajudam a nomear e reutilizar tipos.

Union Types (|) permitem múltiplos tipos, **Literal Types** restringem a valores específicos.

KORU


DESENVOLVE
programa de formação
e inclusão de talentos
em tecnologia
Grupo
Boticário
KORU