

Pós-Graduação Lato Sensu
Curso de Especialização em Tecnologia Java

Frameworks Web

TYPESCRIPT

Prof. Esp. Hugo Baker Goveia



1. O que é TypeScript?

TypeScript é um superset (superconjunto) de JavaScript, desenvolvido pela Microsoft, que adiciona tipagem estática opcional e outros recursos avançados à linguagem. Ele é projetado para desenvolver grandes aplicações JavaScript e se integra diretamente ao código JavaScript existente.

PRINCIPAIS CARACTERÍSTICAS DO TYPESCRIPT

- **Tipagem Estática:** Ao contrário do JavaScript, que é uma linguagem de tipagem dinâmica, TypeScript permite definir tipos para variáveis, funções, parâmetros e retornos de funções. Isso ajuda a identificar erros durante o desenvolvimento, em vez de encontrá-los apenas em tempo de execução.
- **Compilação para JavaScript:** TypeScript é compilado (**transpilado**) para JavaScript. Isso significa que o código TypeScript é convertido em JavaScript equivalente que pode ser executado em qualquer ambiente onde JavaScript é suportado, como navegadores web e servidores Node.js.
Essa transpilação para JavaScript é feita por meio de um compilador, pelo qual é possível escolher a versão do Javascript a ser usada.
- **Suporte a ES6/ESNext:** TypeScript suporta recursos modernos do JavaScript (ECMAScript 6 e posteriores) e pode transpilar esse código para versões mais antigas do JavaScript, garantindo compatibilidade com navegadores mais antigos.
- **Classes e Interfaces:** TypeScript adiciona suporte a classes e interfaces, facilitando a programação orientada a objetos. Interfaces permitem a definição de contratos explícitos dentro do código.
- **Ferramentas de Desenvolvimento Melhoradas:** A tipagem estática e outros recursos avançados melhoram significativamente a experiência de desenvolvimento, fornecendo autocompletar mais preciso, navegação pelo código e refatoração mais segura.
- **Controle de Erros:** Erros são detectados em tempo de compilação, proporcionando um ciclo de desenvolvimento mais seguro e reduzindo bugs em produção.

2. História do TypeScript

A história do TypeScript reflete a necessidade de uma linguagem que pudesse lidar com os desafios de desenvolvimento de grandes aplicações JavaScript, fornecendo uma melhor estrutura e ferramentas de desenvolvimento mais robustas.

Foi criada por Anders Hejlsberg, que foi o mesmo autor do “Turbo Pascal” e engenheiro chefe da equipe que criou o Delphi, sucessor do Pascal.

Anders também foi engenheiro chefe da equipe que criou o “C#”.

O JavaScript, embora amplamente utilizado e incrivelmente flexível, começou a mostrar suas limitações à medida que os aplicativos da web se tornaram mais complexos. Problemas como a ausência de tipagem estática, dificuldades na manutenção de código grande e a falta de funcionalidades modernas de linguagens de programação tornaram-se evidentes. Em resposta a esses desafios, a Microsoft começou a desenvolver TypeScript.

Lançamento Inicial

Outubro de 2012: Microsoft anunciou a primeira versão pública do TypeScript. A versão inicial (0.8) foi lançada como uma prévia para desenvolvedores, mostrando os recursos principais como tipagem estática, classes e módulos.

O TypeScript foi projetado para ser uma solução que adiciona tipagem opcional ao JavaScript, permitindo aos desenvolvedores detectar erros em tempo de compilação em vez de em tempo de execução, facilitando a construção de grandes aplicações robustas.

Evolução e Adoção

2013-2015: O TypeScript continuou a evoluir rapidamente, incorporando feedbacks da comunidade e adicionando novas funcionalidades. A versão 1.0 **foi lançada em 2014**, marcando a maturidade do TypeScript como uma linguagem confiável para produção.

2015: O lançamento da versão 1.5 trouxe suporte para ES6 (ECMAScript 2015) com recursos como módulos, classes e funções de seta (arrow functions). Isso ajudou TypeScript a ganhar mais popularidade, pois os desenvolvedores podiam usar recursos modernos do JavaScript mesmo em navegadores mais antigos.

2016: TypeScript 2.0 foi lançado, introduzindo recursos como null types (tipos null) e controles de fluxo de análise, que melhoraram ainda mais a segurança e a experiência de desenvolvimento.

Integração com Frameworks Populares

TypeScript ganhou grande tração quando grandes frameworks e bibliotecas começaram a adotá-lo, como por exemplo o **Angular 2**. Quando a equipe do Angular anunciou que o Angular 2 seria escrito em TypeScript, isso deu um grande impulso à adoção da linguagem. TypeScript se tornou a linguagem preferida para desenvolvimento com Angular devido às suas funcionalidades robustas de tipagem e ferramentas.

Suporte da Comunidade e Ferramentas

- **IDE e Editores:** Ferramentas de desenvolvimento como **Visual Studio Code** (também da Microsoft) integraram suporte robusto ao TypeScript, proporcionando autocompletar, refatoração e navegação pelo código de maneira eficiente. Isso melhorou a produtividade dos desenvolvedores e facilitou a adoção de TypeScript.
- **Bibliotecas e Ferramentas:** Muitas bibliotecas populares, como React e Vue.js, começaram a oferecer tipagens TypeScript ou foram reescritas em TypeScript.

Ferramentas como TSLint (agora ESLint com suporte TypeScript) ajudaram a manter o código limpo e consistente.

3. Instalação e configuração

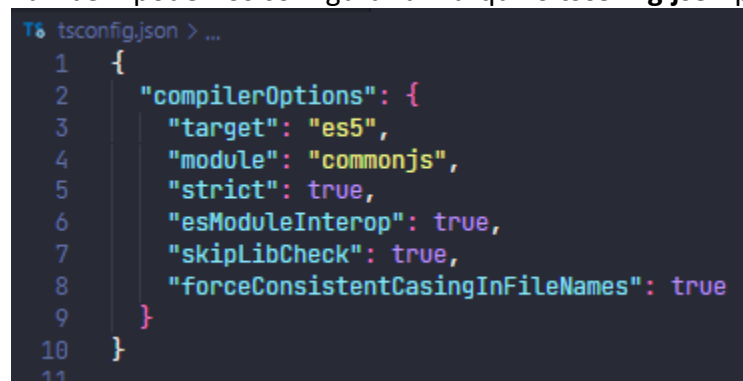
Para começar a usar TypeScript, você precisa instalá-lo globalmente no seu sistema. Você pode fazer isso usando o npm (Node Package Manager):

npm install -g typescript

Depois de instalado, você pode compilar arquivos TypeScript (.ts) para JavaScript (.js) usando o comando tsc:

tsc arquivo.ts

Também podemos configurar um arquivo **tsconfig.json** para definir opções de compilação:



```
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "module": "commonjs",
5      "strict": true,
6      "esModuleInterop": true,
7      "skipLibCheck": true,
8      "forceConsistentCasingInFileNames": true
9    }
10 }
11
```

4. Tipos de dados em TypeScript

TIPOS PRIMITIVOS

TypeScript oferece suporte aos mesmos tipos primitivos do JavaScript, com a adição de algumas funcionalidades específicas.

- **number:** Representa valores numéricos, incluindo inteiros e números de ponto flutuante.

Exemplos:

let idade: number = 30;

let altura: number = 1.75;

- **string:** Representa sequências de caracteres, que podem ser delimitadas por aspas simples, duplas ou crases (para template strings).

Exemplos:

let nome: string = "Alice";

let saudacao: string = `Olá, \${nome}!`;

- **boolean:** Representa valores verdadeiros ou falsos.

Exemplo:

```
let ativo: boolean = true;
```

- **null e undefined:** Representam valores nulos ou indefinidos.

Exemplos:

```
let nulo: null = null;
```

```
let indefinido: undefined = undefined;
```

- **symbol:** Representa valores únicos e imutáveis, frequentemente usados como identificadores de propriedades de objetos.

Exemplo:

```
let sym: symbol = Symbol('descricao');
```

TIPOS DE COLEÇÕES

- **array:** Representa uma coleção de valores do mesmo tipo.

```
let numeros: number[] = [1, 2, 3, 4, 5];
```

```
let palavras: Array<string> = ["um", "dois", "três"];
```

Também é possível definir que o array aceite dois tipos de dados usando:

```
const personContacts: (string | number)[] = ['contato1', 10];
```

- **tuple(tupla):** Representa um array com um número fixo de elementos de tipos específicos.

```
let pessoa: [string, number] = ["Alice", 30];
```

Se eu tentar fazer a seguinte instrução => **pessoa[1] = 'teste';**

Irei receber um erro, pois na segunda posição da tupla só é possível definir valores do tipo "number"

TIPOS DE OBJETOS

"object" Representa uma coleção de propriedades chave-valor.

Exemplo => `let pessoa: { nome: string; idade: number } = { nome: "Alice", idade: 30 };`

ENUM

Enums permitem definir um conjunto de valores nomeados, facilitando o trabalho com um grupo de constantes relacionadas.

```
enum Cor {
  Vermelho,
  Verde,
  Azul
}
let minhaCor: Cor = Cor.Verde;
```

ANY

O tipo any permite que uma variável contenha qualquer tipo de valor. Isso é útil quando você não sabe o tipo exato que uma variável irá armazenar no momento da escrita do código.

```
let qualquerValor: any = 4;
qualquerValor = "Texto";
qualquerValor = true;
```

VOID

O tipo void é usado para funções que não retornam um valor.

```
function saudar(): void {
  console.log("Olá, mundo!");
}
```

NEVER

O tipo never representa o tipo de valores que nunca ocorrem. É usado em funções que lançam exceções ou nunca retornam.

```
function erro(mensagem: string): never {
  throw new Error(mensagem);
}
```

UNION TYPES

Union types permitem que uma variável tenha mais de um tipo. Isso é útil quando um valor pode ser de diferentes tipos.

```
let id: number | string;
id = 123; // Correto
id = "ABC"; // Correto
```

TYPE ALIASES

Type aliases permitem criar nomes personalizados para **tipos existentes**, facilitando a leitura e manutenção do código.

```
type ID = number | string;
let userId: ID;
userId = 123; // Correto
userId = "ABC"; // Correto
```

No exemplo acima, estamos criando o tipo de variável chamado ID, que podemos atribuir esse tipo de variável para outras novas variáveis que vamos criar no nosso programa. Nesse exemplo também, quando alguma variável for do tipo "ID" ela vai poder receber "number" e "string"

INTERSECTION TYPES

Intersection types combinam múltiplos tipos em um. Isso é útil para combinar interfaces. Usamos o símbolo "&" para combinar os tipos que vamos usar.

```
interface Pessoa {
  nome: string;
}

interface Trabalhador {
  empresa: string;
}

type Empregado = Pessoa & Trabalhador;

let empregado: Empregado = {
  nome: "Alice",
  empresa: "Empresa XYZ"
};
```

No exemplo acima combinamos os tipos “Pessoa” e “Trabalhador” e quando definimos o objeto empregado do tipo “Empregado”, ele tem as propriedades “nome” e “empresa”

5. Tipos de argumentos em funções

Em TypeScript, a tipagem de argumentos de funções permite especificar os tipos de entrada que uma função aceita, bem como o tipo de valor que ela retorna. Isso adiciona uma camada de segurança e clareza ao código, ajudando a evitar erros comuns e facilitando a manutenção e compreensão do código.

DEFININDO TIPOS DE ARGUMENTOS

Ao definir uma função, você pode especificar os tipos de seus parâmetros. Isso garante que a função só será chamada com os tipos corretos de argumentos.

```
function saudacao(nome: string): string {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao(42)); // Erro: Argument of type 'number' is not assignable to parameter of type 'string'.
```

No exemplo acima temos a função “saudacao” que está tipada no parâmetro “nome” para que somente seja possível passar o tipo de string como valor e também temos a tipagem de retorno após os “:” definindo como string o retorno dessa função.

E mais abaixo no código temos a primeira chamada à função “saudacao” passando valor de string corretamente. Já na segunda vez que a função “saudacao” foi chamada, é passado o valor do tipo “number” e já podemos observar que uma traço vermelho aparece logo abaixo do número “42” dentro do parâmetro. Isso ocorre porque a função foi “tipada” e aguarda receber uma string como parâmetro.

Observe abaixo, que se eu levo o cursor do mouse sobre o parâmetro, podemos observar que é informado o erro que ali estamos recebendo.

```
function saudacao(nome: string): string {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao(42)); // Erro: Argument of type 'number' is not assignable to parameter of type 'string'.
```

Argument of type 'number' is not assignable to parameter of type 'string'. ts(2345)

View Problem (Alt+F8) No quick fixes available

Dessa forma o TypeScript já nos ajuda em tempo de desenvolvimento a não cometer erros que poderíamos perceber antes com o JavaScript puro somente em tempo de execução de nosso programa.

```
function saudacao(nome: string): string {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

Feita a mudança para que o parâmetro seja string o erro para de acontecer.

TIPAGEM DO VALOR DE RETORNO

Além de definir os tipos dos argumentos, é possível especificar o tipo de valor que a função deve retornar. Se a função tentar retornar um valor de tipo diferente, TypeScript gerará um erro.

```
function saudacao(nome: string): number {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

Usando como exemplo a função tipada mostrada anteriormente, eu modifiquei a tipagem do valor de retorno de “string” para “number” e imediatamente o TypeScript já me acusou erro no “return”. Essa notificação em “tempo de desenvolvimento” nos ajuda muito a evitar problemas em produção com o software já rodando.

Por isso é importante para nós buscarmos realizar a tipagem de nossos códigos.

```
Type 'string' is not assignable to type 'number'. ts(2322)

fu View Problem (Alt+F8) No quick fixes available
} return `Olá, ${nome}!`;

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

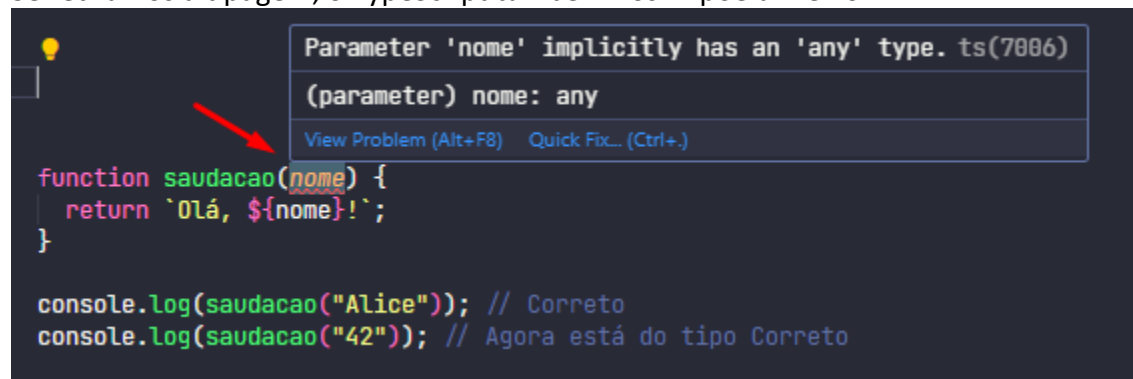
Levando o mouse no “return” podemos ver a mensagem referente ao erro que estamos tomando ali devido a função estar esperando um “number” de retorno.


```
function saudacao(nome: string): string {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

Retornando para “string” no tipo de retorno da função, observamos que o TypeScript para de acusar erro no “return”.

Se retiramos a tipagem, o TypeScript também nos impõe um erro.



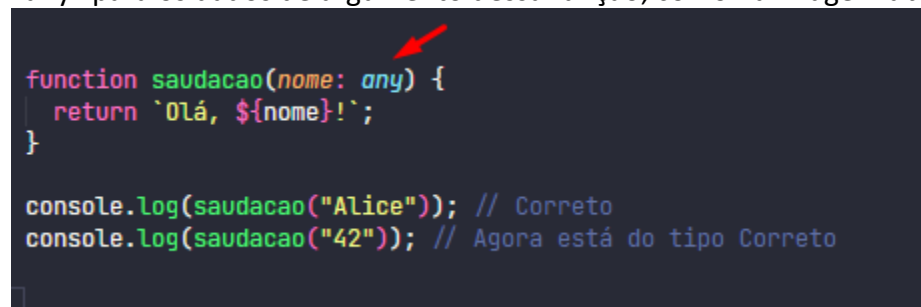
The screenshot shows a code editor with the following code:

```
function saudacao(nome) {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

A red arrow points to the parameter `nome` in the function signature. A tooltip is displayed above it with the text: "Parameter 'nome' implicitly has an 'any' type. ts(7006)". Below the tooltip, it says "(parameter) nome: any" and provides links for "View Problem (Alt+F8)" and "Quick Fix... (Ctrl+.)".

Uma forma de contornar uma situação dessas, **que não é muito recomendada** é informar o tipo “any” para os dados de argumento dessa função, como na imagem abaixo.



The screenshot shows a code editor with the following code:

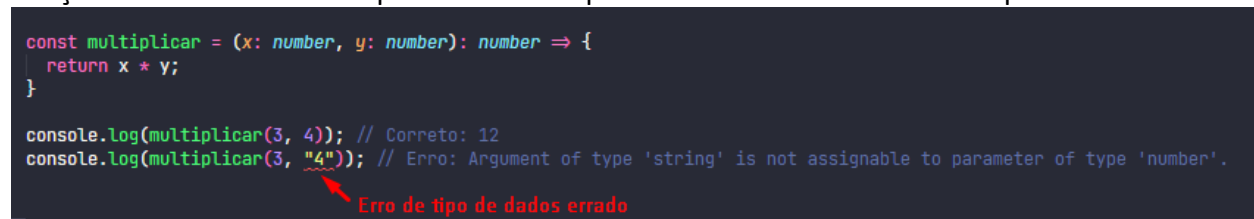
```
function saudacao(nome: any) {
  return `Olá, ${nome}!`;
}

console.log(saudacao("Alice")); // Correto
console.log(saudacao("42")); // Agora está do tipo Correto
```

A red arrow points to the `any` type annotation for the parameter `nome`.

TIPAGEM DE FUNÇÕES ANÔNIMAS

Funções anônimas também podem ter seus parâmetros e valor de retorno tipados.



The screenshot shows a code editor with the following code:

```
const multiplicar = (x: number, y: number): number => {
  return x * y;
}

console.log(multiplicar(3, 4)); // Correto: 12
console.log(multiplicar(3, "4")); // Erro: Argument of type 'string' is not assignable to parameter of type 'number'.
```

A red arrow points to the string `"4"` in the second call to `multiplicar`. Below the arrow, the text "Erro de tipo de dados errado" is written in red.

TIPAGEM DE PARÂMETROS OPCIONAIS

Em TypeScript, você pode definir parâmetros opcionais usando o símbolo “?”. Isso indica que o argumento pode ser fornecido ou não.

```
function saudacao(nome: string, saudacaoPersonalizada?: string): string {
  if (saudacaoPersonalizada) {
    return `${saudacaoPersonalizada}, ${nome}!`;
  } else {
    return `Olá, ${nome}!`;
  }
}

console.log(saudacao("Alice")); // Correto: Olá, Alice!
console.log(saudacao("Alice", "Bom dia")); // Correto: Bom dia, Alice!
```

No exemplo acima, podemos observar que a função está preparada para receber ou não o valor de “saudacaoPersonalizada” por causa do uso do “?”. E quando não é passado nenhum valor para esse parâmetro, automaticamente a função se adapta retornando uma saudação “padrão”.

PARÂMETROS PADRÃO

Podemos fornecer valores padrão para os parâmetros de uma função. Se o argumento não for fornecido, o valor padrão será usado.

```
function saudacaoPadrao(nome: string, saudacao: string = "Olá"): string {
  return `${saudacao}, ${nome}!`;
}

console.log(saudacaoPadrao("Alice")); // Correto: Olá, Alice!
console.log(saudacaoPadrao("Alice", "Bom dia")); // Correto: Bom dia, Alice!
```

Observe que no exemplo acima é definido um valor padrão para o parâmetro “saudação” e quando não passamos valor para essa variável no parâmetro da função ele automaticamente recebe o valor “Olá”.

Inclusive temos aqui até uma oportunidade de observar uma refatoração no código, se observarmos esse código do exemplo acima reduz para 1 linha o conteúdo da função mostrada mais anteriormente que usava mais linhas para retornar uma “saudação padrão” caso não fosse informado um valor especial para a saudação.

PARÂMETROS REST

TypeScript permite definir um número variável de argumentos usando a sintaxe

```
function criarUsuario(
  nome: string,
  idade: number,
  ...contatos: string[]
): { nome: string; idade: number; contatos: string[] } {
  return {
    nome,
    idade,
    contatos,
  };
}

const usuario = criarUsuario("Flávio", 25, "flavio@teste.com", "555-1234");
console.log(usuario); // { nome: 'Flávio', idade: 25, contatos: ['flavio@teste.com', '555-1234'] }
```

6. Tipagem de objetos

Em TypeScript, a tipagem de objetos permite definir a estrutura de um objeto, especificando quais propriedades ele deve ter e quais tipos de valores essas propriedades podem armazenar. Isso ajuda a garantir que os objetos sejam usados de forma consistente e evita erros comuns relacionados a propriedades ausentes ou tipos incorretos.

```
let pessoa: { nome: string; idade: number };

pessoa = { nome: "Alice", idade: 30 }; // Correto
pessoa = { nome: "Bob" }; // Erro: falta a propriedade 'idade'
pessoa = { nome: "Charlie", idade: "trinta" }; // Erro: 'idade' deve ser um número
```

Observe no exemplo acima a importância de criar um objeto tipado. Ao tentarmos criar uma pessoa sem informar a idade o TypeScript acusa um erro.

Novamente ao tentar criar uma pessoa informando um valor “string” onde era esperado um valor do tipo “number”, também obtemos um erro.

INTERFACES

Interfaces são muito importantes na hora de definir tipos de objetos. Elas permitem reutilizar definições de tipos em várias partes do código.

```
interface Pessoa {
  nome: string;
  idade: number;
}

let usuario: Pessoa = { nome: "Alice", idade: 30 }; // Correto
let usuario2: Pessoa = { nome: "Bob" }; // Erro: falta a propriedade 'idade'
```

Observe no exemplo acima, que temos o “usuário” e o “usuario2” sendo definidos do tipo “Pessoa”, com isso temos maior facilidade na hora de declarar uma tipagem de variáveis de objetos.

PROPRIEDADES OPCIONAIS

Em objetos também podemos definir propriedades opcionais usando o símbolo “?”.

```
interface Produto {
  nome: string;
  preco: number;
  descricao?: string; // Propriedade opcional
}

let item: Produto = { nome: "Camiseta", preco: 29.99 }; // Correto
let item2: Produto = { nome: "Calça", preco: 59.99, descricao: "Calça jeans" }; // Correto
```

Conforme exemplo acima, podemos observar que o item descrição é somente informado em um dos casos, pois essa propriedade está como opcional por causa do “?”

PROPRIEDADES DE SOMENTE LEITURA

Você pode definir propriedades de somente leitura em interfaces usando a palavra-chave “readonly”.

```
interface Carro {
  readonly marca: string;
  modelo: string;
  ano: number;
}

let meuCarro: Carro = { marca: "Toyota", modelo: "Corolla", ano: 2020 };
meuCarro.marca = "Honda"; // Erro: não é possível atribuir à propriedade 'marca' porque ela é somente leitura
```

No exemplo acima, depois de criarmos “meuCarro”, nós não conseguimos mudar mais a marca, pois essa propriedade está definida como “readonly”.

7. Interfaces e Types

Em TypeScript, as interfaces e os tipos (types) são duas maneiras de definir a forma de objetos e outras estruturas de dados. Ambos são usados para descrever a forma de um objeto, mas têm algumas diferenças e características únicas.

INTERFACES

As interfaces são usadas para definir a estrutura de um objeto, especificando os nomes e tipos de suas propriedades. Elas são especialmente úteis para garantir que um objeto atenda a um contrato específico.

TYPES (TIPOS)

Os tipos (types) em TypeScript são uma forma mais flexível de definir a forma de dados. Eles podem ser usados para definir tipos complexos que combinam vários tipos primitivos ou outros tipos definidos pelo usuário.

Definindo um Tipo

Para definir um tipo, usamos a palavra-chave `type` seguida do nome do tipo.

```
type Pessoa = {  
  nome: string;  
  idade: number;  
};  
  
let usuario: Pessoa = { nome: "Alice", idade: 30 }; // Correto  
let usuario2: Pessoa = { nome: "Bob" }; // Erro: falta a propriedade 'idade'
```

DIFERENÇAS DE TYPES E INTERFACES

Não existe uma opinião formada dizendo quando usar `type` e quando usar `interface`, isso na maioria das vezes é uma questão de gosto.

Não podemos ter um `type` e uma `interface` com o mesmo nome dentro do mesmo “arquivo/contexto”.

Uma diferença é a possibilidade de criar mais de uma `interface` com o mesmo nome.

Ex:

```
interface Person {  
  name: string  
  age: number  
}  
  
interface Person {  
  sex: 'male' | 'female'  
}
```

No exemplo acima é entendido que a `interface Person` recebe um incremento mais abaixo e quando vamos criar algo do tipo `Person` pode ser adicionado o “sex” também como propriedade.

Um outro ponto é quanto a **Extensibilidade**.

`Interfaces` podem ser estendidas com a palavra-chave `extends`, enquanto tipos usam interseções (&) para combinar tipos.

```
interface Animal {  
  nome: string;  
}  
  
interface Cachorro extends Animal {  
  raca: string;  
}  
  
let meuCachorro: Cachorro = { nome: "Rex", raca: "Labrador" };
```

```
type Animal = { nome: string };  
type Cachorro = Animal & { raca: string };  
  
let meuCachorro2: Cachorro = { nome: "Rex", raca: "Labrador" };
```