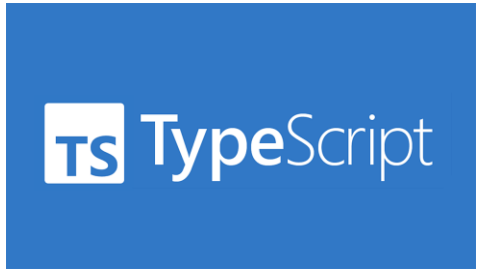
Tecnologia Web III



TypeScript

- TypeScript é a linguagem fortemente tipada para desenvolvimento de aplicativos Angular. É um superset de JavaScript com suporte de segurança de tipos em tempo de design, além de disponibilizar ferramentas de desenvolvimento.

- Os navegadores não podem executar TypeScript diretamente. O Typescript deve ser "transpilado" para JavaScript usando o compilador ***tsc***.

Instalação do TypeScript

Para instalar o pacote TypeScript como uma dependência em seu projeto utilize npm

```
npm i typescript
```

Existem várias opções disponíveis que o npm dependendo onde deseja que o TypeScript seja instalado.

`npm i -g typescript` para instalar globalmente o pacote TypeScript

`npm i -D typescript` para instalar o pacote TypeScript como uma dependência de desenvolvimento

`tsc soma.ts` para compilar um arquivo .ts

Tipos de Dados

- Os tipos de dados primitivos do TypeScript, são por exemplo number, string, boolean, array, object

```
let numero: number = 10;

let texto: string = "Olá, mundo";

let ativo: boolean = true;

let lista: number[] = [1, 2, 3];

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

Declaração de variáveis

- Em TypeScript, que é um superset tipado de JavaScript que compila para JavaScript puro, a declaração de variáveis é semelhante ao JavaScript, mas com a adição de tipos estáticos:

1. Usando let ou const:

```
// Declara uma variável do tipo number com o valor 10
let minhaVariavel: number = 10;
// Declara uma constante do tipo string
const minhaConstante: string = "Olá, mundo!";
```

2. Inferência de tipo: Em TypeScript, pode deixar o compilador inferir o tipo da variável com base no valor atribuído:

```
let numero = 5; // TypeScript infere que 'numero' é do tipo number
let texto = "Hello";
```

3. Tipos explícitos: Você também pode declarar variáveis sem atribuir um valor inicial, especificando o tipo explicitamente:

```
// Declara uma variável do tipo number sem atribuir um valor inicial
let meuNumero: number;
 // Declara uma variável do tipo sem atribuir um valor inicial
let minhaString: string; string
```

Uso do ***var*** em TypeScript

Existem algumas razões pelas quais é recomendado evitar o uso da palavra-chave var em favor de ***let*** e ***const*** em TypeScript. Aqui estão algumas razões importantes:

- Escopo de Bloco vs. Escopo de Função:
  - Em JavaScript puro, var tem escopo de função ou global, o que pode levar a bugs difíceis de identificar, especialmente em loops e condicionais.
  - Em TypeScript, ***var*** possui escopo de bloco, no entanto, o uso de ***let*** e ***const*** é mais explícito e claro em relação ao escopo de bloco, o que ajuda a evitar erros.
- Hoisting:
  - Variáveis declaradas com var ***são*** içadas (hoisted) para o topo do escopo em que foram declaradas, o que pode causar comportamentos inesperados.
  - ***let*** e ***const*** não são içadas da mesma forma, o que ajuda a evitar problemas relacionados ao hoisting.

Uso do ***var*** em TypeScript

- Reatribuição e Redeclaração:
  - Com var, é possível redeclarar a mesma variável no mesmo escopo, o que pode levar a confusão e erros no código.
  - let permite reatribuição, mas não redeclaração no mesmo escopo, tornando o código mais seguro.
  - const impede tanto a reatribuição quanto a redeclaração da mesma variável no mesmo escopo.
- Legibilidade e Manutenção:
  - O uso de let e const torna o código mais legível, uma vez que você pode entender facilmente o escopo e a imutabilidade das variáveis.
  - Código mais claro e legível facilita a manutenção e colaboração em equipes.

Funções em TypeScript

- As funções em TypeScript podem ter parâmetros tipados e valores de retorno específicos, garantindo consistência e facilitando a detecção de erros durante o desenvolvimento.

```
function soma(a: number, b: number): number {
    return a + b;
}
```

ArrowFunctions

- As Arrow Function lembram bastante as expressões lambda da linguagem C# e seus dois benefícios principais são: Serem menos verbosas do que as funções tradicionais. O valor do 'this' é definido a partir das funções onde foram definidas, não sendo necessário usar o método bind()

Por que escrever:

```
function() {
        return 42;
}
```

quando pode escrever apenas

```
() => 42?
```

```
// sem arrow function
let relatorio = {
    gerarPdf: function() {
    // código aqui...
    },
    gerarRelatorio: function(){
        document.addEventListener('click', function(e) {
            this.gerarPdf();
        }.bind(this));
    }
}

// com arrow function
let relatorio = {
    gerarPdf: function() {
        // codigo aqui ...
    },
    gerarRelatorio: function() {
        document.addEventListener('click', (e) =>
        this.gerarPdf());
   }
}
```

Interfaces em TypeScript

- As interfaces no TypeScript são utilizadas para definir a forma de objetos, permitindo a criação de contratos que especificam quais propriedades um objeto deve ter.

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

function saudar(pessoa: Pessoa) {
    console.log(`Olá, ${pessoa.nome}!`);
}
```

Classes em TypeScript

- As classes em TypeScript permitem a criação de estruturas orientadas a objetos com propriedades e métodos. Elas suportam herança, encapsulamento e polimorfismo

```
class Animal {
    nome: string;

    constructor(nome: string) {
        this.nome = nome;
    }

    emitirSom() {
        console.log("Som do
animal");
    }
}

let cachorro = new Animal("Rex");
cachorro.emitirSom();
```

Interfaces x Classes

Interface

Definição:

- Interfaces são contratos que definem a estrutura de um objeto, especificando quais propriedades e métodos devem estar presentes em um objeto que implementa a interface.

Implementação:

- As interfaces são implementadas por classes ou objetos que devem seguir a estrutura definida pela interface, garantindo consistência na forma dos objetos.

Herança Múltipla:

- As interfaces em TypeScript suportam herança múltipla, o que significa que uma classe pode implementar várias interfaces ao mesmo tempo.

Extensibilidade:

- As interfaces podem ser estendidas para adicionar novas propriedades ou métodos, permitindo uma maior flexibilidade na definição de contratos.

Class

Definição:

- Classes são estruturas de código que encapsulam dados (propriedades) e comportamentos (métodos) em um único objeto. Elas são usadas para criar instâncias de objetos.

Instância:

- As classes são usadas para criar instâncias de objetos com base no molde definido pela classe. Cada instância é um objeto independente.

Herança:

- As classes em TypeScript suportam herança (não suportam diretamente herança múltipla), permitindo que uma classe herde propriedades e métodos de outra classe pai.

Métodos Construtores:

- As classes podem ter um método construtor para inicializar propriedades quando uma instância é criada.

Módulos e Importações

- Os módulos no TypeScript permitem organizar o código em unidades independentes e reutilizáveis.
- As importações e exportações são usadas para compartilhar funcionalidades entre arquivos.

```
// arquivo1.ts
export function dobrarNumero(numero: number):
number {
    return numero * 2;
}

// arquivo2.ts
import { dobrarNumero } from './arquivo1';

let numeroDobrado = dobrarNumero(5);
console.log(numeroDobrado); // Saída: 10
```

Manipulação de arrays (Tipagem Estática)

JavaScript

- Em JavaScript, os arrays podem conter elementos de diferentes tipos e não há verificação de tipos em tempo de compilação.

TypeScript

- Com a tipagem estática do TypeScript, é possível definir o tipo dos elementos do array, permitindo uma verificação de tipos em tempo de compilação e evitando erros relacionados à tipagem.

```
// Definição de um array de números em JavaScript
let numeros = [1, 2, 3, 4, 5];
```

```
// Definição de um array de números em TypeScript
let numeros: number[] = [1, 2, 3, 4, 5];
```

Manipulação de arrays (Iteração Segura)

JavaScript

- Em JavaScript, é possível iterar sobre um array usando loops ***for***, ***for...of***, ***forEach***, entre outros, mas não há garantias sobre os tipos dos elementos.

```
// Iteração sobre um array em JavaScript
numeros.forEach(num => {
    console.log(num);
});
```

TypeScript

- Com TypeScript, ao definir o tipo dos elementos do array, a iteração sobre o array é mais segura, pois o compilador pode detectar possíveis erros de tipos durante a compilação.

```
// Iteração segura sobre um array em TypeScript
numeros.forEach(num => {
    console.log(num);
});
```

Manipulação de arrays (Métodos de Array)

JavaScript

- JavaScript oferece uma variedade de métodos de array como map, filter, reduce, find, forEach, entre outros, para manipulação e iteração de arrays.

```
// Utilizando métodos de array em JavaScript
let dobrados = numeros.map(num =>num*2);
let pares = numeros.filter(num =>num%2===0);
```

TypeScript

- TypeScript herda todos os métodos nativos de array do JavaScript e fornece suporte para tipos genéricos, o que torna a manipulação de arrays mais robusta e segura em termos de tipos.

```
// Utilizando métodos de array em TypeScript
let dobrados: number[] = numeros.map(num => num*2);
let pares: number[]=numeros.filter(num=>num%2===0);
```

Manipulação de arrays (Tipos Genéricos)

TypeScript

- TypeScript suporta o uso de tipos genéricos em arrays, permitindo a definição de arrays com tipos específicos ou parâmetros genéricos que podem ser inferidos automaticamente.

Em JavaScript, devido à tipagem dinâmica, não há necessidade de declarar explicitamente o tipo dos elementos de um array.

```
// Função para imprimir um array em JavaScript
// sem tipos genéricos
function imprimeArray(arr) {
    arr.forEach(item => {
        console.log(item);
    });
}
// Utilizando a função sem tipos genéricos
imprimeArray([1, 2, 3]);
imprimeArray(["a", "b", "c"]);
```

```
// Definição de um array genérico em TypeScript
function imprimeArray<T>(arr: T[]): void {
    arr.forEach(item => {
        console.log(item);
    });
}
// Utilizando a função com tipos genéricos
imprimeArray<number>([1, 2, 3]);
imprimeArray<string>(["a", "b", "c"]);
```

Métodos de Manipulação de Arrays em TypeScript

- O método ***map()*** cria um novo array com os resultados da chamada de uma função para cada elemento do array.
- Ele não altera o array original, mas retorna um novo array com os resultados das operações aplicadas a cada elemento.

```
let numeros: number[] = [1, 2, 3, 4, 5];
let dobrados: number[] = numeros.map(num => num * 2);
console.log(dobrados); // Saída: [2, 4, 6, 8, 10]
```

Métodos de Manipulação de Arrays em TypeScript

- O método ***filter()*** cria um novo array com todos os elementos que passaram no teste implementado pela função fornecida.
- Ele não altera o array original, mas retorna um novo array contendo apenas os elementos que satisfazem a condição especificada.

```
let numeros: number[] = [1, 2, 3, 4, 5];
let pares: number[] = numeros.filter(num=>num%2===0);
console.log(pares); // Saída: [2, 4]
```

Métodos de Manipulação de Arrays em TypeScript

- O método ***reduce()*** executa uma função redutora em cada elemento do array, resultando em um único valor de retorno.
- Ele percorre o array da esquerda para a direita, acumulando o resultado da função para cada elemento.
- Ao omitir o valor inicial (“0”), o total iniciará por padrão do primeiro item do array.

```
let numeros: number[] = [1, 2, 3, 4, 5];
let soma: number = numeros.reduce((acu, val) => acu
+ val, 0);
console.log(soma); // Saída: 15
```

Métodos de Manipulação de Arrays em TypeScript

- O método ***forEach()*** executa uma função para cada elemento do array, sem retornar um novo array.
- Ele é utilizado para iterar sobre os elementos do array e executar uma determinada operação para cada um deles, como exibir no console ou modificar os elementos.

```
let frutas: string[] = ["Maçã", "Banana", "Laranja"];
frutas.forEach(fruta => {
    console.log(fruta);
});
// Saída:
// Maçã
// Banana
// Laranja
```

Exercícios sobre Métodos de Manipulação de Arrays em TypeScript

1) Instruções: Dado um array de strings representando nomes, utilize o método map() para criar um novo array com os nomes em maiúsculas.

- Array de Nomes: ["Alice", "Bob", "Charlie", "David"]

```
let nomes: string[] = ["Alice", "Bob", "Charlie", "David"];
let nomesMaiusculos: string[] = nomes.map(nome => nome.toUpperCase());
console.log(nomesMaiusculos); // Saída: ["ALICE", "BOB", "CHARLIE", "DAVID"]
```

Exercícios sobre Métodos de Manipulação de Arrays em TypeScript

2) Instruções: Dado um array de números, utilize o método filter() para criar um novo array contendo apenas os números maiores que 10.

- Array de Números: [5, 15, 3, 20, 8, 12]

```
let numeros: number[] = [5, 15, 3, 20, 8, 12];
let maioresQueDez: number[] = numeros.filter(num => num > 10);
console.log(maioresQueDez); // Saída: [15, 20, 12]
```

Exercícios sobre Métodos de Manipulação de Arrays em TypeScript

3) Instruções: Dado um array de números, utilize o método reduce() para calcular a multiplicação de todos os elementos do array.

- Array de Números: [2, 3, 4, 5]

```
let numeros: number[] = [2, 3, 4, 5];
let multiplicacao: number = numeros.reduce((acc, curr) => acc * curr, 1);
console.log(multiplicacao); // Saída: 120
```

Glossário

Hoisting: é um comportamento em JavaScript onde as declarações de variáveis e funções são movidas para o topo do seu escopo durante a fase de compilação, antes da execução do código. Isso significa que, mesmo que declare uma variável ou função em um local específico no código, o interpretador JavaScript irá tratá-la como se tivesse sido declarada no início do seu escopo.

No caso de variáveis declaradas com var, o hoisting faz com que a declaração da variável seja movida para o topo do escopo, enquanto a atribuição da variável permanece no local onde foi definida. Isso pode causar comportamentos inesperados e confusão no código.

Exemplo de hoisting com ***var***:

```
console.log(x); // Retorna 'undefined' em vez de erro
var x = 10;
```

No exemplo acima, devido ao hoisting, a declaração da variável x é movida para o topo do escopo, fazendo com que *console.log(x)* não gere um erro, mas retorne undefined.

Glossário

Transpile (transpilar): é um termo mais genérico usado para descrever o processo de converter código de uma linguagem para outra. No caso do TypeScript, como ele é uma linguagem que tem como alvo o JavaScript, podemos dizer que ele transpila o código TypeScript em código JavaScript.

Código verboso: é aquele que precisa de mais palavras, ou palavras mais longas, do que o necessário para expressar adequadamente a intenção do código. Em códigos verbosos existem muitos símbolos ou símbolos muito longos.

Glossário

Fortemente Tipado: no TypeScript traz benefícios como:

- Detecção de Erros em Tempo de Compilação: O compilador do TypeScript verifica se os tipos estão sendo usados de forma consistente em todo o código, identificando erros de tipo antes da execução.
- Melhor Documentação e Legibilidade: A definição explícita dos tipos torna o código mais legível e auto-documentado, facilitando a compreensão do código por outros desenvolvedores.
- Refatoração Segura: Com a tipagem forte, é mais seguro realizar refatorações no código, pois o compilador pode identificar possíveis problemas de tipo introduzidos durante as alterações.
- Intellisense Aprimorado: Em IDEs compatíveis, como o Visual Studio Code, a tipagem forte do TypeScript permite um suporte mais robusto ao Intellisense, fornecendo sugestões de código mais precisas.
- Código Mais Robusto: A tipagem forte ajuda a evitar erros comuns, como passagem de argumentos incorretos para funções ou operações inválidas em tipos incompatíveis.