TypeScript

Introdução

O TypeScript é um pre-processador de códigos JavaScript open source desenvolvido e mantido pela Microsoft.

O objetivo do TypeScript é aplicar a orientação a objetos ao JavaScript, oferecendo classes, tipagens, interfaces, Generics etc.

Os navegadores não interpretam TypeScript por isso é necessario transpilar o código em ECMAScript.

Instalação

Para instalar o TypeScript utilizamos o **node.js,** para verificar a versão instalada do TypeScript usamos o seguinte comando:

tsc –v

Para instalar o TypeScript usamos o seguinte comando:

npm install -g typescript

Entendendo o compilador

O compilador TypeScript é altamente configurável. Ele nos permite definir o local onde estão os arquivos **.ts** dentro do nosso projeto, o diretório de destino dos arquivos transpilados, a versão ECMAScript que será utilizada, o nível de restrição do verificador de tipos e até se o compilador deve permitir arquivos JavaScript.

Cada uma das opções de configuração pode ser passada para um arquivo chamado **tsconfig.json**. Para quem não conhece, esse é o principal arquivo de configuração do TypeScript.

Para criar esse arquivo dentro de um novo projeto, basta executar o comando tsc --init . Isso vai criar um arquivo na raiz do seu projeto com algumas configurações padrões, como target, module entre outras.

Devemos entender que o transpilador possui diversas configurações em seu escopo por exemplo **noEmitOnError** essa propriedade por exemplo define se o compilador não vai compilar o arquivo se houver um erro, por padrão ele vem false, ou seja por padrão ele vai compilar.

Para compilarmos um arquivo .ts usamos o seguinte comando:

tsc nomedoarquivo.ts

Para compilarmos todos os arquivos presente em uma pasta usamos o seguinte comando:

tsc -w

Para termos uma visão geral de todas as configurações do compilador é importante lermos a documentação através do seguinte link:

https://www.typescriptlang.org/docs/handbook/compiler-options.html

Conhecendo os Types

Var, Let e Const

Antes de entendermos os types, temos que entender o que são os Var, Let e Const.

Var:

As variáveis do tipo Var eram o único tipo de variável disponível antes do ECMAScript 5, basicamente são variáveis de escopo, ou seja, são definidas no escopo e podem ser acessadas por qualquer lugar do programa, ou seja, basicamente quando definimos um var dentro de um if por exemplo na execução ela será levada para o topo do escopo e poderá ser acessada por todo o código.

Let:

As variáveis do tipo Let são variáveis de blocos, ou seja, quando definimos uma variável do tipo Let estamos dizendo que essa variável só poderá ser acessada no bloco em que foi definida.

Const:

As variáveis do tipo Const tem seu valor inalterado, ou seja, seu estado é imutável, assim como o var é uma variável de escopo.

Types

Boolean:

Assim como qualquer linguagem possui dois valores true e false:

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

Number:

No TypeScript todos os tipos numéricos são definidos pelo tipo Number:

```
let octal: number = 0o745;
let binary: number = 0b1111;
let decimal: number = 34;
let hex: number = 0xf34d;
```

String:

Assim como em outras linguagens no TypeScript strings são textos:

```
let cor: string = "verde";
```


Podemos também utilizar de templates string para concatenar valores:

```
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${idade} anos.
```

Lenght:

Quando trabalhamos com strings as vezes precisamos manipular as mesmas, para isso temos alguns comandos que nos permitem fazer isso:

Usando de .lenght nós teremos o tamanho da string (a quantidade de caracteres).

IndexOf:

Esse método nos permite localizar a posição de uma string em outra string:

```
let sentence: string = `Olá, meu nome é ${ nome }, eu tenho ${idade}
anos.`;
console.log(sentence.indexOf('nome')); //posição 9
```

Array:

Assim como em outras linguagens usamos [ ] para declararmos um array, com a diferença que um array possui um tamanho dinâmico não sendo preciso declarar o tamanho do array no início dele:

```
let numeros: number[] = [1, 2, 3];
let textos : string[] = ["exemplo 1", "exemplo 2", "exemplo 3"];

let numeros: Array<number> = [1, 2, 3];
let textos: Array<string> = ["exemplo 1", "exemplo 2", "exemplo 3"];
```

Para adicionarmos um item ao array já criado utilizamos o comando .push:

```
numeros.push(4);
textos.push("exemplo 3");
```

ReadonlyArray:

O ReadonlyArray<T> é um array que nos permite somente leitura. Ele remove todos os métodos de alteração de um array, como push , pop etc.

Na versão 3.4 do TypeScript foi adicionado uma nova sintaxe para o ReadonlyArray: Readonly<T>:

```
let numerosDaMega: ReadonlyArray<number> = [8, 5, 5, 11, 4, 28];

 let numerosDaMega: Readonly<number[]> = [8, 5, 5, 11, 4, 28];
```

Tuple:

As tuplas são uma estrutura de dados parecida com array, porém diferente do array as tuplas podem utilizar diversos tipos de dados:

```
let list: [string, number, string] =  ['string', 1, 'string 2'];
```

No TypeScript 4.0 foi adicionado a possibilidade de darmos nomes para os tipos:

```
let list: [nome: string, idade: number, email: string] = ['Bill Gates',
65, 'bill@teste.com'];
```

Assim como no array utilizamos .push para adicionar valores:

```
let list: [string, number] = ['Bill Gates', 1];
list.push('Steve', 2);
```

Assim como Array podemos também utilizar Readonly.

Enum:

O enum nos permite declarar um conjunto de valores/constantes predefinidos. Existem três formas de se trabalhar com enum:

Number-> Os enums numéricos armazenam variáveis com valores numéricos, nos podemos declarar um valor inicial:

```
export enum DiaDaSemana {
    Segunda = 1,
    Terca = 2,
    Quarta = 3,
    Quinta = 4,
    Sexta = 5,
    Sabado = 6,
    Domingo = 7
}
```



```
let dia = DiaDaSemana[19]; // Terca
let diaNumero = DiaDaSemana[dia]; // 19
let diaString= DiaDaSemana["Segunda"]; // 18

Resultado:
// Terca
// 19
// 18
```

Caso não passemos um valor em uma variável, basta entendermos que o valor dessa variável é o valor da variável anterior +1 e caso a primeira variável seja sem valor o seu valor será 0.

String-> Esses enums obrigatoriamente precisam iniciar com um valor:

```
export enum DiaDaSemana {
    Segunda = "Segunda-feira",
    Terca = "Terça-feira",
    Quarta = "Quarta-feira",
    Quinta = "Quinta-feira",
    Sexta = "Sexta-feira",
    Sabado = "Sábado",
    Domingo = "Domingo",
}
```

```
console.log(DiaDaSemana.Sexta); //Sexta-feira
console.log(DiaDaSemana['Sabado']); //Sábado
```

```
Resultado:
//Sexta-feira
//Sábado
```

Heterogeneous-> Esses podem usar tanto String quanto numérico:

```
export enum Heterogeneous {
    Segunda = 'Segunda-feira',
    Terca = 1,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo = 18,
}
```

Union:

Permite combinar um ou mais tipos:

```
let exemploVariavel: (string | number);
exemploVariavel = 123;
console.log(exemploVariavel);
exemploVariavel = "ABC";
console.log(exemploVariavel);

Resultado:
 //123
//ABC
```

Any:

O any basicamente é um tipo primitivo que aceita todos os outros.

Tipando funções:

O TS também nos permite tipar o retorno das funções:

```
function calc(x: number, y: number): string {
    return `resultado: ${x + y}`;
}
```


Void:

O TS permite o tipo void como retorno de uma função:

```
function log(): void {
    console.log('Sem retorno');
}
```

Never:

O type never significa que algo nunca deve ocorrer:

```
function fail(message: string): never { throw new Error(message); }
```

Type assertions:

O type assertions funciona como o cast de outras linguagens, basicamente serve para alterarmos o tipo primitivo de uma variável existente que já possui outro tipo primitivo, basta acrescentarmos <tipoPrimitivo> antes da variável:

```
function typeAssetions(codigoAny: any) {
    let codigoNumber: number = <number>codigoAny;
    return codigoNumber * 10;
}
typeAssetions(10);
```

Resultado:

```
//number
```

POO

Classes:

Assim como em outras linguagens as classes são estruturas para os objetos, a estrutura é a seguinte:

```
class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;
}
```

Métodos:

```
class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;

    constructor(numeroDaConta: number, titular: string, saldo: number) {
        this.numeroDaConta = numeroDaConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    consultaSaldo(): string {
        return `O seu saldo atual é: ${this.saldo}`;
    }

    adicionaSaldo(saldo: number): void {
        this.saldo + saldo;
    }

    sacarDoSaldo(valor: number): void {
        this.saldo -= valor;
    }
}
```

Encapsulamento:

Assim como em outras linguagens o TS possui o paradigma de encapsulamento, possuindo os mesmos modificadores de acesso:

->Public.

->Protected.

->Private.

```
export class Conta {
   /* outros atributos*/
    private saldo: number;
    /* outros métodos*/
```

Herança:

Assim como no Java o TS possui herança, porém a herança múltipla é apenas nas interfaces:

```
class ContaPF extends Conta {
    cpf: number;
interface Tributavel {
    CalculaTributo(): number;
}

 class ContaPJ extends Conta implements Tributavel {

   CalculaTributo(): number {
         //implementação do cálculo para o valor tributável para
     }
 }
```

Classe abstrata:

As classes abstratas não permitem realizar qualquer tipo de instância, elas são utilizadas como modelos para outras classes, que são conhecidas como classes concretas:

```
abstract class Conta {
    /* implementação da classe*/
}
```

Essa classe não pode ser instanciada apenas pode ser usada como modelo usando de herança.

Getters e Setters

```
class Conta {
    private _numeroDaConta: number;
    titular: string;
    private _saldo: number;

    get numeroDaConta(): number {
        return this._numeroDaConta;
```

Generics

Função genérica:

Para criar uma função genérica basta adicionar as chaves <T>:

```
function funcaoGenerica<T>() {}


    funcaoGenerica<number>()
    funcaoGenerica<string>()
    funcaoGenerica<boolean>()


 function funcaoGenerica<T>(value: T): T {
     return value;
 }

 function fun<T, U, V>(args1:T, args2: U, args3: V): V {
   return args3;
 }
```

Classe genérica:

```
class classeGenerica<T>  {
    private arr: Array<T> = [];

    adicionaValor(item: T) {
        this.arr.push(item);
    }

    retornaValores() {
        return this.arr;
    }
}


 let classeGenerica1 = new classeGenerica<number>();
```

Interface genérica:

```
interface InterfaceGenerica<I> {
    removeItem(item: I)
}

class classeGenerica<T> implements InterfaceGenerica<T> {
   //os outros métodos
     removeItem(item: T) {
        let index = this.arr.indexOf(item);
        if (index > -1)
            this.arr.splice(index, 1);
}
```

Decorator

Os decorators nos permitem decorar dinamicamente as características de uma classe.

Atualmente é um recurso experimental do TS, por isso temos que ativar o seguinte comando no tsconfig:

```
"compilerOptions": {
   /*outras configurações*/
  "experimentalDecorators": true, /* Enables experimental support for ES7
Decorator. */
```

Um decorator é definido pelo @ e seu nome, exemplo: @NgModule.

Para criarmos um decorator basicamente precisamos 3 propriedades:

Target(alvo): Pode ser um método estático ou uma function construtora de uma classe.

propertyKey(chave): Nome do membro da instância que será utilizada no alvo.

Descriptor(descritor): A propriedade descriptor do membro da instância, chamando o método:

```
Object.getOwnPropertyDescriptor() .
```


Criando um método decorator:

Para criarmos um decorator de um método basicamente temos que criar uma função com as 3 propiedades: Target, propertyKey e Descriptor; e em seguida devemos colocar esse decorator no método que queremos:

```
function analisaSaldo(target: any, key: any, descriptor: any) {
    //implementação
}

@analisaSaldo
consultaSaldo(): string {
    return `O seu saldo atual é: ${this.saldo}`;
}
```

Decorator de propriedade:

Basicamente o decorator de uma propriedade é uma função com apenas o target e o propertyKey:

```
function validaTitular(target: any, propertyKey: any) {
    //implementação
}

@validaTitular
titular: string;
```

Decorator de parâmetro:

Basicamente o decorator terá que ter um **target**, **propertyKey** e um **parameterIndex**, que é o número da posição do parâmetro na função começando com 0:

```
function saldo() {
    return (
        target: any,
        propertyKey: number,
        parameterIndex: number,
    ) => {
        console.log('target', target);
        console.log('property key', propertyKey);
        console.log('parameter index', parameterIndex);
    }
}

adicionaSaldo(@saldo() saldo: number): void {
    this.saldo + saldo;
}
```

Decorator de classe:

O decorator de uma classe deve ser declarado antes da declaração da própria classe, ele irá receber apenas um parâmetro que será o construtor da classe:

```
function log(ctor: any) {
    console.log(ctor)
}

@log
class Conta {}
```

Namespaces:

Namespaces são basicamente o package do TypeScript, é uma forma de organizarmos nossas classes em um determinado grupo:

Normal:

```
class ContaSalario extends Conta {}
class ContaInvestimento extends Conta {}
```

Namespace:

```
namespace Banco {
    export class Conta {
        numeroDaConta: number;
        titular: string;
        saldo: number;
        /* outras implementações */
    }
}

class ContaPF extends Banco.Conta {}
class ContaPJ extends Banco.Conta {}
```


Modules:

Modules são basicamente a forma de se **export** classes e **import** classes:

```
//contas.ts
export class Conta {
    numeroDaConta: number;
    titular: string;
    saldo: number;
    /* outras implementações */
}

//contaInvestimento.ts
import { Conta } from "./Conta";
export class ContaInvestimento extends Conta { }
```