**TYPESCRIPT**

O TypeScript oferece todos os recursos do JavaScript e uma camada adicional sobre eles: o sistema de tipos *TypeScript*. Seu uso traz algumas vantagens, como:

* Feedback mais rápido de erros.
* Processo de refatoração mais fácil.
* Autocomplete da linguagem, muito boa no VSCode
* Poder adotar gradualmente TypeScript em uma base de código.

No curso TypeScript parte 1, criamos um arquivo tsconfig.json. Esse arquivo faz com que os arquivos em TS sejam compilados para JS:

{

    "compilerOptions": {

        "outDir": "dist/js",

        "target": "ES6",

"noEmitOnError": true,

"noImplicitAny": true,

"removeComments": true,

"strictNullChecks": true

    },

    "include": ["app/\*\*/\*"]

}

* **outDir** indica para onde os arquivos compilados indicados pela propriedade *include* serão salvos. Nesse caso, os arquivos compilados para JS serão salvos dentro da pasta *dist*.
* **target** indica para o compilador tsc (TypeScript Compiler) para qual versão do JavaScript o código escrito em TypeScript deve ser compilado. Isso significa que o resultado final será arquivos JS sem qualquer referência para a sintaxe do TypeScript. Todo código TS precisa ser compilado para uma sintaxe compatível.
* **noEmitOnError: true** faz com que, enquanto exista um erro no TS, não será compilado um arquivo JS. Caso seja *false,* será compilado mesmo com erros.
* **noImpicitAny: true** tem o intuito de desativar o tipo implícito *any* passando essa configuração especial. Isso fará com que o compilador emita um erro em todos os locais que o tipo *any* foi adotado implicitamente. Por padrão, o *any* é assumido automaticamente pelo TS quando não definimos o tipo das nossas variáveis.
* **include** indica para o compilador onde estão os arquivos “.ts” que ele deve compilar. Nesse caso, os arquivos TS estão dentro da pasta *app*
* **removeComments: true** faz com que os comentários inseridos no código TS não seja compilado para o código em JS, deixando o arquivo mais enxuto.

**npm run server** digite essa instrução no Node em seu terminal para inicia-lo e fazer com que abra a página do browser. Essa página ficara sendo atualizada automaticamente de acordo com suas alterações no código sem precisar ficar recarregando manualmente.

**npm run start** tem o mesmo objetivo do *npm run server* porém fará com que as alterações no código fonte nos arquivos TS sejam automaticamente compiladas para os arquivos JS e também atualizadas no navegador.

**private** diferentemente do JS, onde usamos a tralha (#) para declarar um parâmetro como privado, no TS utilizamos o modificador *private*.

* **Public** tem o intuito de atribuir ao parâmetro a permissão para ser modificado. Quando não informamos nada, subentende-se que o parâmetro é *public.*

**Tipagem Estática:** Tipar sempre as propriedades da classe, os métodos e os parâmetros dos métodos, evitando assim o *any*. Quando não retornar nada, colocar o *void.* Ex:

export class Negociacao {

    private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

    constructor (data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor;

    }

    get data(): Date {

        return this.\_data;

    }

    get quantidade(): number {

        return this.\_quantidade;

    }

    get valor(): number {

        return this.\_valor;

    }

    get volume(): number {

        return this.\_quantidade \* this.\_valor;

    }

}

**Atalho para declarar no próprio constructor:** Se no construtor da sua classe você coloca o modificador *private*, explicita isso, ou *public*, isso indica para o TS que ele vai por debaixo dos panos criar uma propriedade da classe que contenha o mesmo nome do seu, com os parâmetros do construtor, e por debaixo dos panos vai fazer a atribuição, pegar esse valor e jogar para você. Ex:

**ANTES**

export class Negociacao {

    private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

    constructor (data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor; }

    get data(): Date {

        return this.\_data; }

    get quantidade(): number {

        return this.\_quantidade; }

    get valor(): number {

        return this.\_valor; }

    get volume(): number {

        return this.\_quantidade \* this.\_valor;

    }

}

**DEPOIS: TORNANDO O CÓDIGO MAIS ENXUTO**

export class Negociacao {

    constructor (

        private \_data: Date,

        private \_quantidade: number,

        private \_valor: number

        ) {}

    get data(): Date {

        return this.\_data;

    }

    get quantidade(): number {

        return this.\_quantidade;

    }

    get valor(): number {

        return this.\_valor;

    }

    get volume(): number {

        return this.\_quantidade \* this.\_valor;

    }

}

**Simplificando a declaração do tipo de um Array e o ReadyonlyArray:**

**ANTES**

import { Negociacao } from "./negociacao.js";

export class Negociacoes{

    private negociacoes: Array<Negociacao> = [];

    adiciona(negociacao: Negociacao){

        this.negociacoes.push(negociacao);

    }

lista(): ReadonlyArray<Negociacao>{

        return [...this.negociacoes];

    }

}

**DEPOIS**

import { Negociacao } from "./negociacao.js";

export class Negociacoes{

    private negociacoes: Negociacao[] = [];

    adiciona(negociacao: Negociacao){

        this.negociacoes.push(negociacao);

    }

lista(): readonly Negociacao[] {

        return [...this.negociacoes];

    }

}

Tópicos:

* Interfaces
* Generics
* Enums
* Tuples

As principais diferenças entre o tipo normal e as interfaces é que um tipo não pode ser reaberto para adicionar novas propriedades, enquanto que a interface é sempre extensível.