**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**

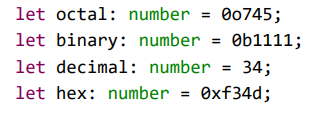
**Boolean:**

Assim como qualquer linguagem possui dois valores true e false:



**Number:**

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



**String:**

Assim como em outras linguagens no TypeScript strings são textos:



Podemos também utilizar de templates string para concatenar valores:



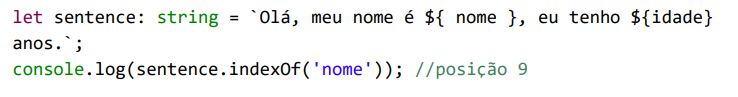
**Lenght:**

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).

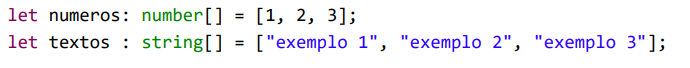
**IndexOf:**

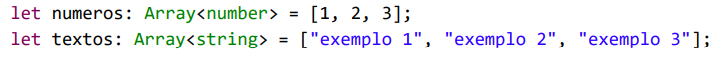
Esse método nos permite localizar a posição de uma string em outra string:



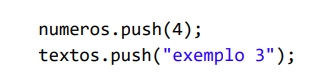
**Array:**

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:





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



**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>:





**Tuple:**

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



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



Assim como no array utilizamos .push para adicionar valores:

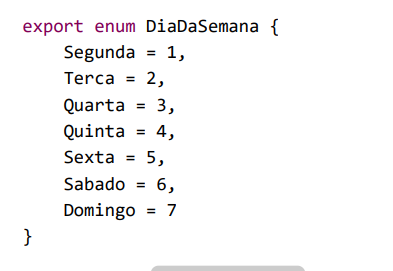


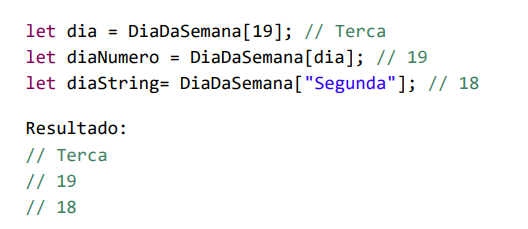
Assim como Array podemos também utilizar Readonly.

**Enum:**

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

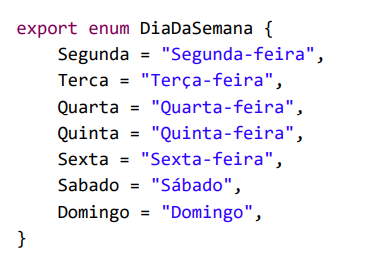
**Number->** Os enums numéricos armazenam variáveis com valores numéricos, nos podemos declarar um valor inicial:

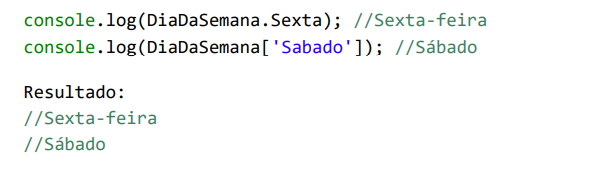




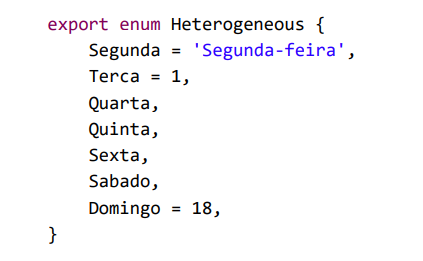
Caso não passemos um valor em uma variável, basta entendermos que o valor dessa variável é o valor da variável anterior +1 e caso a primeira variável seja sem valor o seu valor será 0.

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



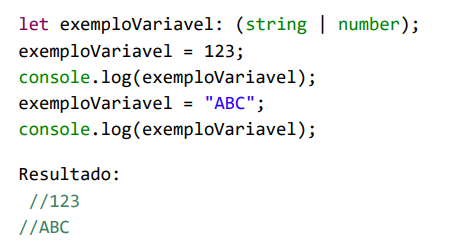


**Heterogeneous->** Esses podem usar tanto String quanto numérico:



**Union:**

Permite combinar um ou mais tipos:

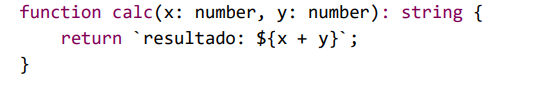


**Any:**

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

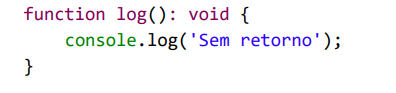
**Tipando funções:**

O TS também nos permite tipar o retorno das funções:



**Void:**

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



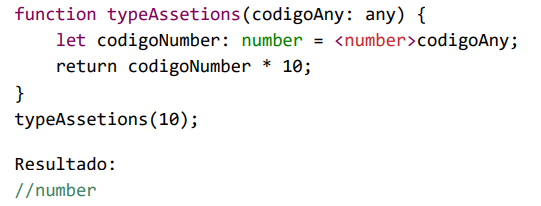
**Never:**

O type never significa que algo nunca deve ocorrer:



**Type assertions:**

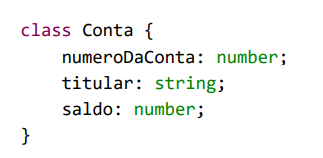
O type assertions funciona como o cast de outras linguagens, basicamente serve para alterarmos o tipo primitivo de uma variável existente que já possui outro tipo primitivo, basta acrescentarmos <tipoPrimitivo> antes da variável:



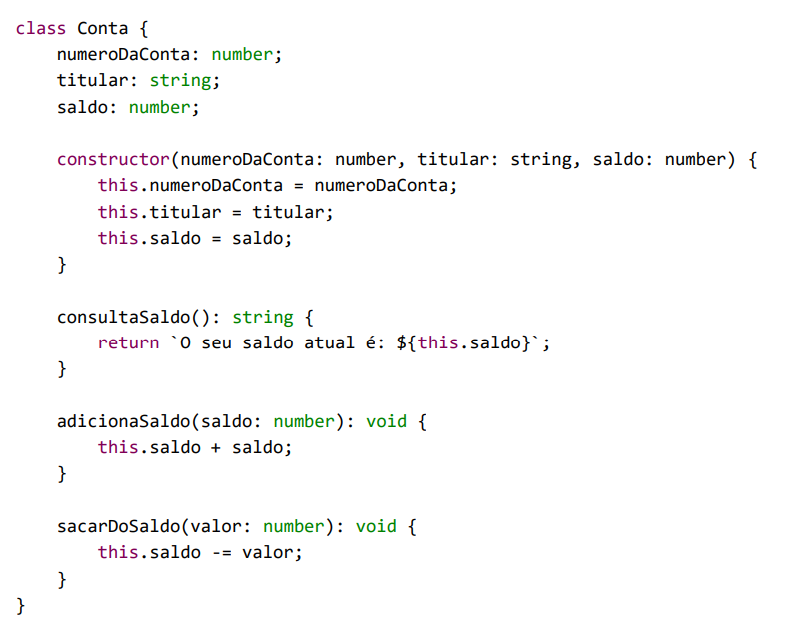
**POO**

**Classes:**

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



**Métodos:**

****

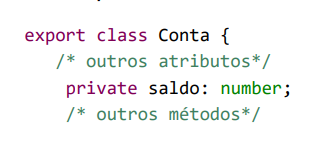
**Encapsulamento:**

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

**->Public.**

**->Protected.**

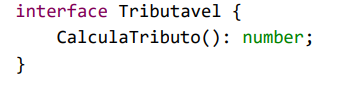
**->Private.**

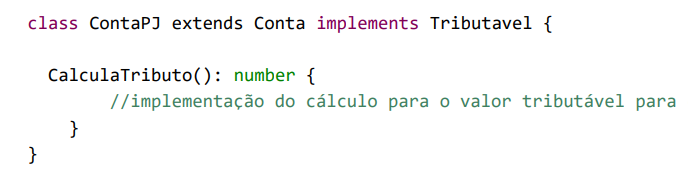
****

**Herança:**

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

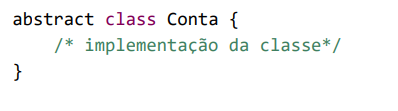






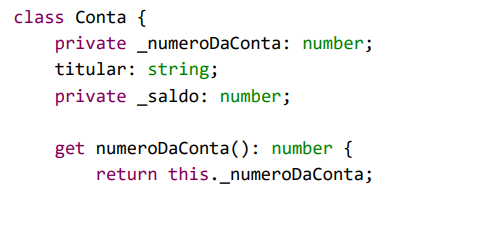
**Classe abstrata:**

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



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

**Getters e Setters**

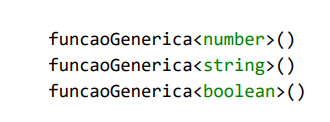
****

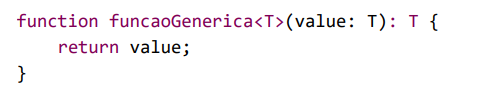
**Generics**

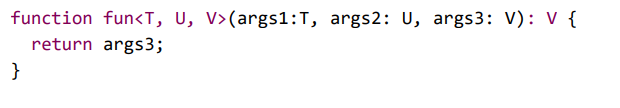
**Função genérica:**

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

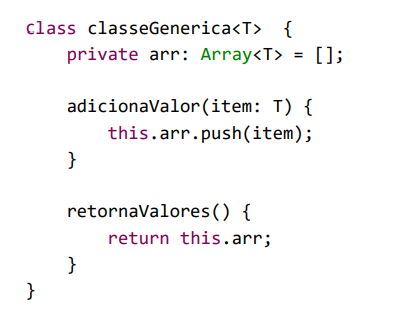


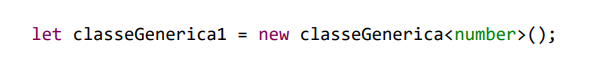




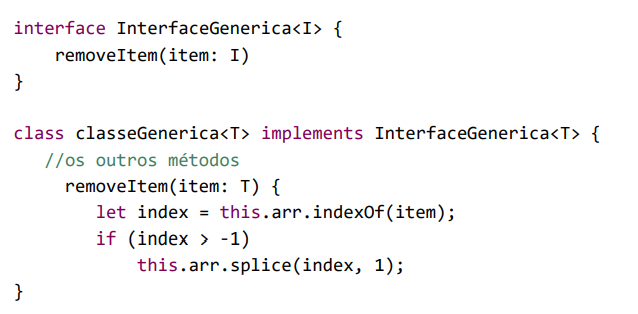


**Classe genérica:**

****

****

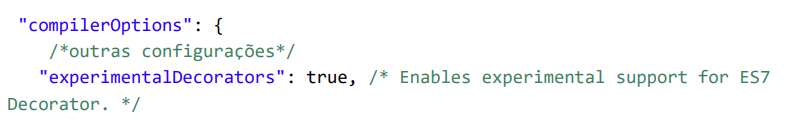
**Interface genérica:**

****

**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:



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.

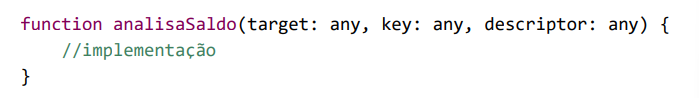
**propertyKey(chave):** Nome do membro da instância que será utilizada no alvo.

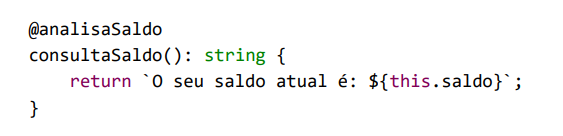
**Descriptor(descritor):** A propriedade descriptor do membro da instância, chamando o método:



**Criando um método decorator:**

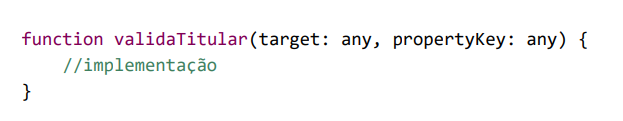
Para criarmos um decorator de um método basicamente temos que criar uma função com as 3 propiedades: Target, propertyKey e Descriptor; e em seguida devemos colocar esse decorator no método que queremos:





**Decorator de propriedade:**

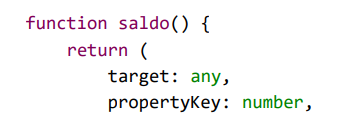
Basicamente o decorator de uma propriedade é uma função com apenas o target e o propertyKey:

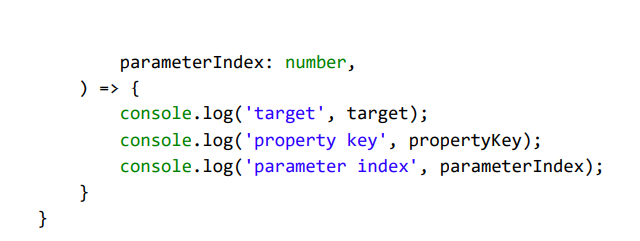


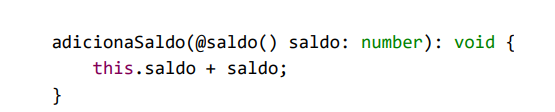


**Decorator de parâmetro:**

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:

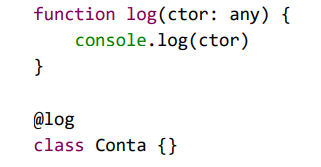






**Decorator de classe:**

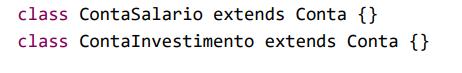
O decorator de uma classe deve ser declarado antes da declaração da própria classe, ele irá receber apenas um parâmetro que será o construtor da classe:



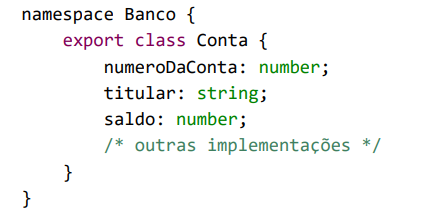
**Namespaces:**

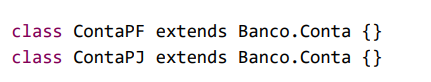
Namespaces são basicamente o package do TypeScript, é uma forma de organizarmos nossas classes em um determinado grupo:

Normal:



Namespace:





**Modules:**

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

