**TYPESCRIPT**

O TypeScript é um **superset** do JavaScript criado por **Anders Hejlsberg** em 2012. Podemos dizer o TS representa o JavaScript com superpoderes, já que ele adiciona algumas funcionalidades á linguagem.

O ponto forte do TypeScript é sua **tipagem estática.** Isso significa que os desenvolvedores devem especificar o tipo de dado ao criar uma variável, função ou objeto em seus códigos.

Além disso, todo código TypeScript, quando executado, é compilado para JavaScript. Outro fato interessante é que o TypeScript é mantido pela **Microsoft**, que sempre dá suporte e atualização ao superset.

**PORQUE UTILIZAR TYPESCRIPT?**

**1. Adiciona confiabilidade ao programa:** por especificarmos o tipo de dado ao criarmos variáveis, nós deixamos o código mais confiável e consistente.

**2. Adiciona novas funcionalidades:** podemos fazer uso de muitas funcionalidades úteis como Generics, Interfaces e outros.

**3. Verificar erros antes de executar o código:** o TypeScript ajuda a evitar erros e previne problemas de sintaxe e lógica, pois ele fornece uma verificação de tipo de dado durante o desenvolvimento. Isso poupa o tempo do programador ao debugar o código.

**INSTALAÇÃO E COMANDOS**

Para instalar: **npm install typescript -g**

Para executar uma vez: **tsc nomeDoArquivo.ts**

Para executar sempre que houver alterações: **tsc nomeDoArquivo.ts -w**

Para inicializar o arquivo de configurações: **tsc --init**

**PRINCIPAIS RECURSOS DO TYPESCRIPT - BÁSICOS**

**1. TYPE (DECLARAÇÃO DE TIPOS):** esse é o principal recurso do TypeScript. Você pode declarar o tipo de variáveis, parâmetros e retornos de funções, dentre outros.

let some: *any* = 12 // any aceita qualquer tipo

const firstName: *string* = 'Gabriel'

const age: *number* = 17

const preferedNumbers: *number*[] = [12, 14, 16]

const parentsName: *string*[] = ['Marcia', 'Edcarlos']

const cats: *object*[] = [{name: 'Brisa', color: 'White'}]

function sum(*a*: *number*, *b*: *number*): *number* { // função retorna um number

    return *a* + *b*

}

function sayHelloTo(*name*: *string*): *void*{ // função não retorna nada

console.log(`Olá, ${*name*}!`))

}

**1.1. INFERENCIA X ANNOTATION:** esses dois termos se referem ao modo como declaramos as variáveis. Quando é por **inferência**, declaramos a variável sem especificar o tipo dela, mas este está **implícito** e o TypeScript vai cuidar para que ele não seja modificado. Já na **annotation**, nós declaramos o tipo da variável **explicitamente.**

**Inferência = tipo é implícito**

let y = 12

**Annotation = tipo é explicito**

let z: *number* = 13

**2. TUPLAS:** podemos dizer que tuplas são **arrays mais específicos.** Você pode declarar o tipo de dado que deve estar em cada elemento do array. Veja no exemplo a seguir:

const myTuple: [*boolean*, *number*, *number*[]] = [true, 2, [4, 5]]

**3. OBJECT LITERALS:** podemos dizer que object literals são **objetos mais específicos**. Você pode declarar o tipo de dado que cada chave de um objeto receberá como valor. Veja no exemplo a seguir:

const user: { name: *string*, age: *number*, brazilian: *boolean* } = {

    name: 'Gabriel',

    age: 17,

    brazilian: true

}

**4. UNION TYPE:** usamos union type quando permitirmos que determinado elemento possua mais de um tipo de dado. Veja no exemplo a seguir:

let id: *string* | *number* = 12 // id aceita uma string ou um number

**5. TYPE ALIAS:** permite criar um union type para que este possa ser reutilizado em outras declarações. Veja no exemplo a seguir:

type *idType* = *string* | *number*

let userId: *idType* = '12'

let productId: *idType* = 6

**6. LITERAL TYPES:** recurso para **definir tipos que aceitam apenas valores específicos.** Veja só um exemplo de literal type:

let activeOrPending: 'ativo' | 'pendente'

activeOrPending = 'ativo'

**7. QUESTION MARK (?):** também conhecido como ponto de interrogação, o question mark é utilizado quando definimos algo **opcional**. Veja o exemplo.

function greetings(*name*: *string*, *greet*?: *string*): *void*{

    if (*greet*){

        console.log(`Olá, ${*greet*} ${*name*}.`)

    } else {

        console.log(`Olá ${*name*}.`)

    }

} // greet é um parâmetro opcional

**8. NARROWING:** não é bem um recurso do TypeScript, mas sim do JavaScript. Porém, ele é muito usado no TypeScript pois está diretamente relacionado ao tipo de dado dos elementos. Com narrowing, podemos **checar o tipo de dado de algum elemento** utilizando o **typeof.**

function doSomething(*info*: *number* | *boolean*){

    if (typeof *info* === 'number'){

        console.log('Foi number')

    } else{

console.log('Não foi number')

}

}

**PRINCIPAIS RECURSOS DO TYPESCRIPT – INTERMED.**

**9. ENUM:** este é um recurso do TypeScript que nos permite **armazenar um conjunto de valores** de maneira legível. Eles são úteis quando você tem um conjunto fixo de valores relacionados que são mais expressivos quando referenciados por nome em vez de valores numéricos. Veja o exemplo a seguir:

enum *WeekDay* { // enum de dias da semana

    Domingo = 1,

    Segunda = 2,

    Terça = 3,

    Quarta = 4,

    Quinta = 5,

    Sexta = 6,

    Sábado = 7

}

const today = *WeekDay*.Quarta // 4

**10. INTERFACE:** uma interface **modela** um objeto. Ela dita como um objeto deve ser, quais serão suas chaves e o tipo de dado dos valores de cada chave. A interface garante a consistência dos dados dentro do objeto. Posteriormente, um objeto que segue os padrões de uma interface pode recebe-la como tipo. Veja o exemplo a seguir:

interface *IUser* {

    name: *string*,

    age: *number*

}

const gabriel: *IUser* = {

    name: 'Gabriel',

    age: 17

}

A propriedade de uma interface também pode receber a palavra reservada **readonly**, que determinará que aquela propriedade é somente leitura (uma vez declarada, não pode ser alterada).

**PRINCIPAIS RECURSOS DO TYPESCRIPT - AVANÇADO**

**11. GENERICS:** este recurso nos permite **trabalhar com diferentes tipos de dados sem perder a informação sobre esses dados**. Em outras palavras, generics são flexíveis em relação ao tipo de dado que manipulam. Com eles, podemos criar componentes (funções, classes ou interfaces) que podem ser reutilizados com diferentes tipos, mantendo a segurança dos tipos durante a execução do código. Veja o seguinte exemplo:

function showItem<*T*>(*arr*: *T*[]){ // generic que armazena o tipo genérico

*arr*.forEach(*i* => console.log(*i*))

}

showItem([1, 2, 3]) // função que recebeu arr do tipo number

showItem(['Oi', 'Tchau', 'Olá']) // função que recebeu arr do tipo string

showItem([true, 'Oi']) // função que recebeu arr de tipo boolean e string

**12. DECORATORS:** Decorators em TypeScript são uma característica que permite **adicionar metadados a classes, métodos, propriedades e parâmetros** de uma forma declarativa. Eles são uma **maneira de estender ou modificar o comportamento de uma classe, método ou propriedade,** geralmente aplicando lógica adicional ou adicionando funcionalidades extras.

A teoria por trás dos decorators envolve o uso de funções que são chamadas durante a fase de definição, antes que o código seja realmente executado. Essas funções recebem informações sobre a declaração decorada, como o construtor da classe, o nome do método, o tipo de propriedade, etc., e podem manipular essas informações de acordo com a lógica definida na função do decorator. Veja um exemplo a seguir:

function logarMetodo(*target*: *any*, *nomeMetodo*: *string*, *descriptor*: *any*){

    const metodoOriginal = *descriptor*.value

*descriptor*.value = function(...*args*: *any*[]){

        console.log(`Método ${*nomeMetodo*} iniciando com args ${*args*}`)

        const resultado = metodoOriginal.apply(*this*, *args*)

        console.log(`Método ${*nomeMetodo*} feito. Result: ${resultado}`)

        return resultado

    }

    return *descriptor*

}

class ExemploClasse{

    @*logarMetodo*

    soma(*a*: *number*, *b*: *number*): *number*{

        return *a* + *b*

    }

}

const exemplo = **new** ExemploClasse()

const resultado = exemplo.soma(5, 3)

// Método soma iniciando com os argumentos 5,3

// Método soma concluído. Resultado: 8

**CURSO TYPESCRIPT HCODE**

**TIPOS DE DADOS**

Um tipo de dado representa o formato, ou natureza, de uma informação. Por exemplo, se a informação será um número, texto, etc. Um dos princípios do TypeScript é a definição dos tipos que as variáveis vão possuir, nos ajudando a ter uma maior controle sobre essas informações.

**String:** cadeia de caracteres (“Gabriel”, “Teste”, “olá”)

**Number:** qualquer tipo de número. (1, 2, 3)

**Union Types:** união de dois ou mais tipos. (string | number | undefined)

**Boolean:** verdadeiro ou falso. (true or false)

**Array:** estrutura de dados baseada em índices, onde os elementos podem possuir um tipo específico de dado. (string[ ], number[ ], (string | number)[ ]).

**Tuple:** determina o tipo de cada elemento de um array em sua respectiva ordem. ( [string, number, string] )

**Object:** coleção dinâmica de pares chave/valor. (

{

name: “Gabriel”,

age: 18

}

)

**Enum:** tipo de dado que permite definir um conjunto de valores nomeados constantes. (

enum Permission {

ADMIN = “admin”,

USER = “user”,

READONLY = “readonly”

}

)

**Null:** tipo que indica ausência de valor, geralmente usado dentro de um union type. (string | null, number | null)

**Undefined:** tipo que indica algo existente, mas sem valor definido, também muito usado dentro de um union type. (string | undefined, number | undefined).

**Any:** aceita qualquer tipo de dado.

**Unknown:** tipo de dado desconhecido. A diferença entre any e unknown é que any permite que você faça qualquer coisa com uma variável sem verificar o tipo, enquanto unknown exige que você verifique o tipo antes de usá-la, tornando-o mais seguro.

**Void:** indica ausência de retorno de uma função (não retorna nenhum valor).

**Never:** tipo de dado que representa valores que nunca acontecem. É usado em situações onde uma função ou expressão não pode retornar um valor, como em funções que lançam exceções ou entram em um loop infinito.

**Afirmações de tipo:** forma de informar ao compilador que você tem mais conhecimento sobre o tipo de uma variável do que o próprio TypeScript. Isso permite que você trate uma variável de um tipo como sendo de outro tipo. É semelhante ao "casting" em outras linguagens. (“teste” **as** string).

**POO (Programação Orientada a Objetos)**

A POO é um **paradigma de programação** que consiste em criar as instâncias de nossos projetos levando em conta o conceito dos **objetos** na vida real. Esse estilo de programação surgiu na década de 1960 e continua sendo popular até hoje.

Quando pensamos em objetos do mundo real, percebemos que todos tem algo em comum: eles possuem **características** e **ações.** Por exemplo, um celular possui características como cor, marca, memória e câmera, mas também possui ações, como mandar mensagens, ver filmes, olhar redes sociais, etc.

Para construir um objeto, na maioria dos casos, é necessário possuir um **modelo** que nos ajudará a entender como o objeto está e o que ele fará. Por exemplo, para construir uma casa, é necessária uma **planta.** Trazendo para a programação, esse “modelo” é chamado de **classe,** ela é como a receita, a planta, e define as características e ações que nosso objeto possuirá. **O objeto é a instância de uma classe**.Essas características e ações também possuem nomes específicos dentro do paradigma da orientação a objetos. As características são **atributos**, e as ações são **métodos.**

**Quatro pilares – Orientação a Objetos**

Os quatro pilares da orientação a objetos são: **herança, encapsulamento, polimorfismo e abstração.**

**Herança:** Permite que uma classe herde recursos de outra classe (extends).

**Encapsulamento:** Ajuda a proteger a integridade das informações da classe. Existem quatro modificadores de acesso: **public, protected, private** e **readonly (apenas atributo),**

* **Publico:** qualquer um pode acessar (padrão).
* **Protected:** apenas aquela classe e suas filhas podem acessar.
* **Private:** apenas aquela classe pode acessar.
* **Readonly:** permite apenas a leitura de um atributo, impedindo sua modificação,

**Polimorfismo:** qualidade ou estado de ser capaz de assumir diferentes formas. Esse conceito é aplicado quando uma classe filha cria um mesmo atributo ou método de sua classe pai, mas implementando um novo comportamento ou lógica, sobrescrevendo-o.

**Abstração:** conceito que permite criar classes abstratas, que representam atributos e métodos que podem ser utilizadas em mais de um contexto. Essa prática nos ajuda a reaproveitar melhor o código e também a deixá-lo mais leve.

**Módulos**

Um **módulo é basicamente um arquivo separado que armazena uma parte do nosso código**. Ao fazer a importação de um módulo, podemos reaproveitar esse código em vários arquivos.

Alguns benefícios do uso de módulos são:

* Melhor organização do código.
* Melhor reutilização do código.
* Evita conflito nos identificadores das variáveis (cada módulo tem seu próprio escopo)

Ao iniciar um projeto em TypeScript, pode ser que tenhamos poucos arquivos em um único diretório. Mas, conforme o projeto cresce, o número de arquivos também aumenta, o que pode deixar nosso projeto desorganizado e confuso. Aí que entra a necessidade de usar módulos.

**ESModule:**

**import** [o-que-queremos-importar] **from** [nome-do-modulo]

**export** [o-que-queremos-compartilhar]

**CommonJS:**

**module.exports = {**[o-que-queremos-compartilhar]**}**

**const** [o-que-queremos-importar] = **require(**[nome-do-modulo]**)**

**Namespaces**

**Namespace é um recurso específico do TypeScript que nos permite organizar melhor os arquivos em nossos projetos**. O objetivo de um namespace é bem semelhante ao objetivo de um módulo.

“A partir do ECMAScript 2015, os módulos são parte nativa da linguagem JavaScript e por isso são suportados por todos os mecanismos compatíveis. Assim, para novos projetos, os módulos são o mecanismo de organização de código recomendado” – Documentação do TypeScript

**Mixins**

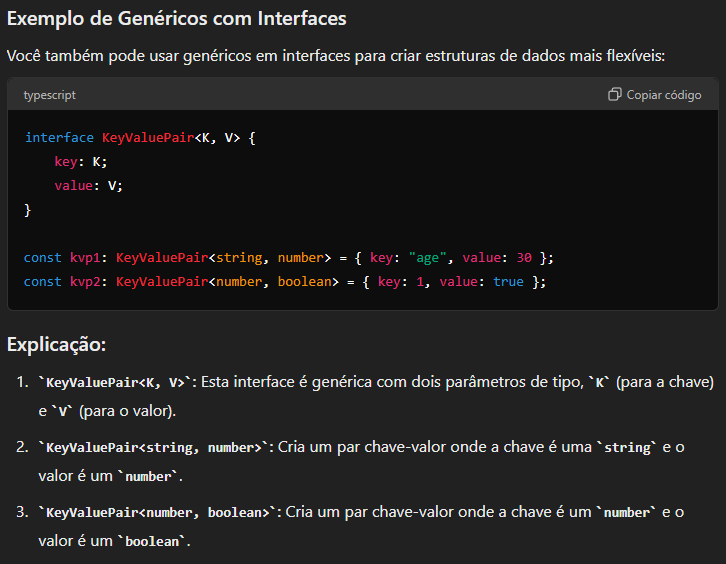
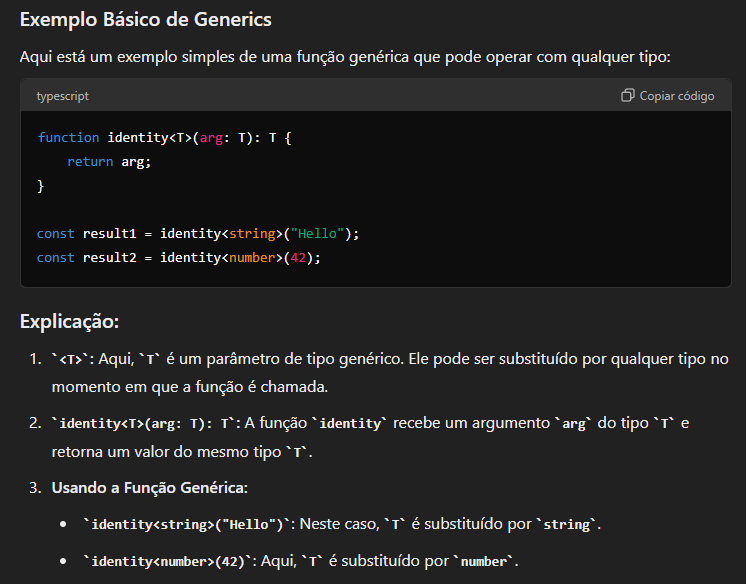
**Mixins em TypeScript permitem combinar várias classes em uma única classe, permitindo reutilizar comportamento sem herança múltipla direta,** que não é suportada em TypeScript ou JavaScript.

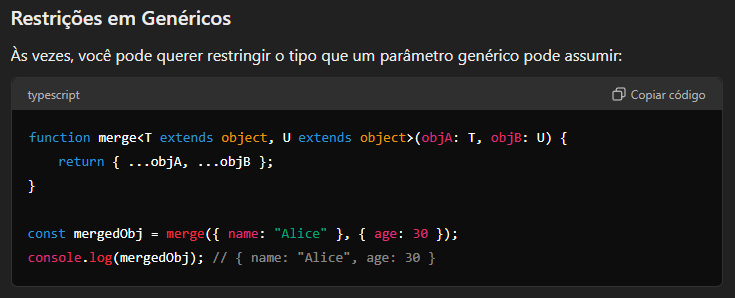
Para fazer uso dos mixins, basta usar a função **applyMixins** passando como argumento as respectivas classes. Essa função está disponível na documentação do TypeScript

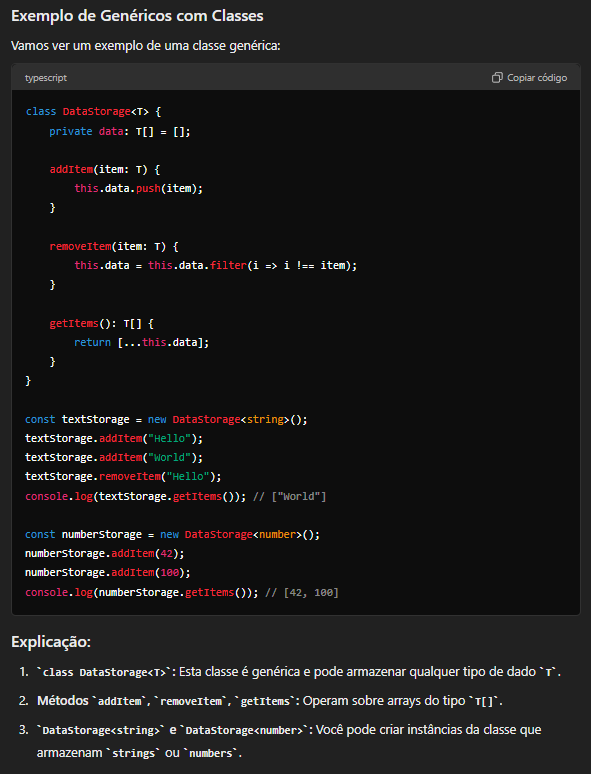
**Generics < >**

**Generics é uma maneira de adaptar nosso código a vários tipos de dados.** Esse conceito nos ajuda a reaproveitar melhor nosso código e torná-lo flexível a diversas situações.

Com generics, podemos criar componentes, funções, classes e interfaces que são especificados apenas quando o componente é utilizado. Isso proporciona maior flexibilidade e reusabilidade no seu código.







**Decorators**

**Decorators é um recurso que nos permite realizar modificações em partes de nosso código no momento da sua execução**. Podemos dizer que eles são como uma anotação adioional ao código que permite uma modificação em seu comportamento.

Um decorator é basicamente uma função que é chamada em tempo de execução e que realizar uma modificação na estrutura onde foi anotada.