**MOD 26 – TYPESCRIPT**

**26.1 – CONHEÇA O TYPESCRIPT**

Gpt e slides: **TypeScript** é uma linguagem de programação desenvolvida pela Microsoft em 2012, sua principal tarefa é adicionar a **sintaxe de tipagem** ao JavaScript. É uma linguagem de código aberto que transpila para JavaScript, o que significa que todo código TypeScript é convertido para JavaScript antes de ser executado em qualquer ambiente que suporte JavaScript. O resultado do TypeScript será um código JavaScript, por isso recebe também o nome de **superset**.

**Principais características:**

* **Tipagem estática**: TypeScript permite definir **tipos** para variáveis, parâmetros de função, e retornos, o que ajuda a evitar erros comuns de tempo de execução e melhora a previsibilidade do código.
* **Suporte a ES6/ES7**: TypeScript suporta recursos modernos do JavaScript (como classes, módulos, e async/await), além de oferecer compatibilidade com versões anteriores.
* **Compilação**: TypeScript precisa ser compilado para JavaScript antes de ser executado, e o compilador verifica erros de tipo e outros problemas durante essa etapa.
* **Integração com ferramentas de desenvolvimento**: Ele se integra bem com editores de código como Visual Studio Code, oferecendo funcionalidades como autocompletar, navegação entre arquivos e refatoração de código.

**Benefícios:**

* **Detecção antecipada de erros**: Como os tipos são verificados em tempo de compilação, muitos erros podem ser detectados antes de o código ser executado.
* **Manutenção mais fácil**: Com tipos explícitos, o código se torna mais claro, facilitando a compreensão e manutenção, especialmente em projetos grandes.
* **Grande comunidade e suporte**: TypeScript é amplamente adotado, com uma comunidade ativa e uma vasta gama de bibliotecas e ferramentas disponíveis.

Em resumo, TypeScript aprimora o JavaScript adicionando tipos estáticos, melhorando a produtividade do desenvolvedor e a qualidade do código.

Slide: **Tipagem fraca**: No JavaScript podemos começar uma variável string e depois, ao evoluir o programa, atribuir um número a essa variável, o que pode resultar em falhas quando o código for executado. Com o TypeScript resolvemos esse problema, podemos dizer que uma variável receberá apenas números, todo o processo do TypeScript é feito em tempo de codificação e build (construção do projeto). Se houver algum problema no código não teremos a geração do JS final.

Aula: **INSTALANDO O TYPESCRIPT**

Vamos iniciar instalando o typescript no terminal do vscode: **npm install --global typescript**

Dps: **npx tsc --init** (criou o tsconfig.json)

Em seguida, vamos criar uma pasta **src** e o arquivo **teste.ts**

Para criarmos um arquivo **JS** diretamente do arquivo **TS**, digitamos no terminal: **npx tsc** (a cada alteração no arquivo **TS** tem que digitar novamente o **npx tsc** para atualizar o **JS**.)

Para resolvermos isso digitamos: **npx tsc --watch**

**26.2 - ANALISE OS TIPOS BÁSICOS**

Slide:

No TypeScript possuímos alguns tipos básicos, primitivos, são eles: **boolean**, **number**, **string**, **array**, **readonlyArray**, **tuplas**, **union** e **any**. Os arrays podem ser declarados através dos colchetes ou das palavras reservadas Array e readonlyArray:

**const nomes: string[] = []**

**const nomes2: Array<string> = []**

**const nomes3: readonlyArray<string> = []**

A única diferença que temos é: ao utilizar o **readonlyArray**, onde os métodos para manipulação do array serão removidos, por exemplo, não poderemos adicionar um novo item com push.

No JavaScript podemos ter num mesmo vários tipos de dados. Isso não é possível no TypeScript, onde temos que definir do que o array será composto, porém temos um tipo de dado que permite um comportamento parecido, a **tupla**.

**TUPLA**: Para criar uma tupla, utilizamos os colchetes informando os tipos que serão aceitos:

**const lista: [string, boolean, number] = [“ana”, true, 22];**

Assim temos uma estrutura similar ao de um <array>. Também podemos informar um nome para cada índice, como se fossem campos:

**const lista: [nome: string, estaEstudando: boolean, idade: number] = [“ana”, true, 22];**

**UNION TYPE:** Apesar da forte tipagem que o TypeScript nos fornece, podemos aceitar mais um tipo em uma variável, utilizando o Union Type**:**

**let idade: number | string;**

**idade = 20**

**idade = “20 anos”**

**ANY**: O TypeScript fornece um tipo fraco, chamado **any** (do inglês: qualquer), onde podemos atribuir qualquer tipo de dado. Mas este recurso deve ser usado em último caso:

**let minhaVariavel: any;**

**minhaVariavel = [1, 2, 3, 4]**

**minhaVariavel = “gian”**

**minhaVariavel = {}**

**minhaVariavel = true**

**aula**: criar o arquivo tipos\_basicos.ts

//usamos as mesmas palavras reservadas do js: **let e const**

let estaChovendo: boolean = false

estaChovendo = true // alterando a variável para true, começou a chover

let idade: number = 33

let altura: number = 1.58 //percebe-se que no TS não ha diferença entre o número do tipo inteiro ou fracionado

// Arrays: Não podemos ter um array com vários tipos, como acontecia no js.

const nacionalidade: string = 'brasileira'

//um array de string! apenas strings!

const colegas: string[] = ['lucas', 'pietro', 'beatriz']

//outra forma de tipar as arrays para strings:

const tecnologia: Array<string> = ['html', 'css', 'js']

//um **array de numeros apenas de leitura**:

const notas: ReadonlyArray<number> = [5, 6, 7, 8]

colegas.push(); //usamos o push() para add um ou mais elementos a array.

tecnologia.push();

// notas.push();   //no array apenas de leitura n conseguimos atribuir o push. MAS conseguimos atribuir os métodos modernos do ES6+

*gpt: O método push() em arrays no JavaScript é utilizado para adicionar um ou mais elementos ao final de um array. Quando você usa o método push(), o array original é modificado, e os novos elementos são adicionados ao final*

*const colegas: string[] = ['lucas', 'pietro', 'beatriz'];*

*colegas.push('ana');*

*const colegas: string[] = ['lucas', 'pietro', 'beatriz'];*

//As **TUPLAS** permitem usar vários tipos na array

const lista: [nome:string, estaEstudando:boolean, idade:number ] = ['denise', true, 33]

// agora faremos com o tipo **UNION TYPES**: uniao de tipos! com o simbolo **|** add um novo tipo, no caso o string

let idadeDaAna: number | string = '25 anos';

//o tipo **any** (qq) aceita qualquer coisa, mas usar com cautela, pois foge do proposito do TS.

let dadosDaApi: any;

dadosDaApi = true

dadosDaApi = [1, 2, 3]

dadosDaApi = 'palavras'

dadosDaApi = 10

**26.3 - APLIQUE TIPAGENS EM FUNÇÕES**

As funções também possuem tipos, em seus argumentos e em seu retorno:

**let somaDoisNumeros(n1: number, n2: number): number {** //o retorno tbm sera number

**return n1 + n2;**

**}**

No exemplo acima, estamos recebendo dois argumentos, ambos do tipo **number** e, após o fechamento dos parênteses, incluímos o tipo do retorno que também será um **number.**

Aula: Criar o arquivo funções.ts

function calculaArea (base: number , altura: number): number {

    return base \* altura;

}

//ou como uma arrow function ( => )

const calculaArea2 = (base: number , altura: number): number =>  base \* altura;

//qnd estiver trabalhando com o operador REST onde vc n conhece todos os argumentos q a funcao pode receber.

//o operador REST nos devolve um array de valores

// Void - Indica que o retorno da função é vazio;

// Rest - Podemos utilizar o Rest no TS e explicitar seu tipo para evitar "any";

function somar(...numeros: number[]): **void** {

    console.log(numeros)

}

// Union Type - Desta forma a função pode retornar uma "String" ou um Número;

function teste(): string | number {

    if (10 > 5) {

        return 'Dez maior que cinco'

    } else {

        return 5

    }

}

const resultadoDeTeste = teste()

**26.4 - CRIE CASTING (a moldagem)**

**Palavra reservada “as” :**

Apesar da forte tipagem que o TypeScript nos fornece, podemos fazer com que um tipo seja entendido como outro tipo no compilador do TypeScript. Por exemplo, podemos tratar um número como se fosse uma string e para isso, utilizamos a palavra reservada **as**:

**const nome: string = 10 as string** *(trate o 10 como uma string)*

O exemplo acima é uma má prática, afinal um nome não é um número. Esse recurso deve ser utilizado com cautela e apenas em casos onde não conhecemos o tipo do dado que vamos receber, o que pode acontecer em integrações a APIs.

AULA: Criar o arquivo casting.ts

namespace casting { /\* necessário para o programa n usar a let idade de outra aba dessa pasta\*/

let idade: any = 25; /\* nesse caso, 25 pode ser de qq tipo (any) \*/

(idade as number).toFixed(); /\* usamos o AS para dizer: trate a idade como (as) do tipo número \*/

(idade as string).length;

(idade as string[]).forEach(x => { /\* convertendo idade para uma array, mas nao funciona qnd roda\*/

console.log(x);

})

let nome: string = 35 as unknown as string;

/\* aq estamos dizendo que 35 esta sendo convertido em um tipo desconhecido e dps convertido em uma string\*/

}

*Gpt: EXEMPLO DE USO COM TOFIXED() - toFixed esta disponivel apenas tipos numericos*

*let num = 5.6789; //entre parêntese colocamos a qnt de casas decimais que queremos.*

*console.log(num.toFixed(2)); // "5.68" 2 casas decimais*

*console.log(num.toFixed(0)); // "6"*

*console.log(num.toFixed(4)); // "5.6789"*

**26.5 - CRIE TIPOS CUSTOMIZADOS (muito usado na pratica)**

Slide: Crie os seus próprios tipos: Além dos tipos básicos que o TypeScript possui, podemos criar nossos próprios tipos, considere:

const profissional = {

nome: “Sabrina”,

idade: 30,

profissao: “designer”

}

A partir dessa estrutura já podemos criar um tipo: **Profissional.** O tipo customizado ficaria:

type **Profissional** = {

nome: string;

idade: number;

profissao: string;

}const dev1: Profissional = {

nome: “bruno”,

idade: 27,

profissional: “dev front-end”,

salario: 5000 // n será válido pois essa prop não faz parte do tipo Profissional

}

**Aula:** Criar o arquivo tipos\_customizados.ts

type **aluno** = {       /\* construindo um tipo customizado: ALUNO \*/

    nome: string;

    cursos?: string[];   /\* a ? é para deixar essa propriedade n sendo obrigatorio qnd for add um novo aluno\*/

    idade: number;

}

const alunos: aluno[] = [

    {

        nome: "Carlos",

        cursos: ["Front-end", "Ux/Ui"],

        idade: 27,

    },

    {

        nome: "Ana",

        cursos: ["Front-end", "Python"],

        idade: 23,

    },

]

alunos.push({   /\* add um novo objeto a const aluno

    nome: "Julia",

    cursos: ["Arquitetura"],

    idade: 29

})

const novoAluno: aluno = {   // Ou pode-se add assim também na array! /\* o novo aluno foi valido \*/

    nome: "Lucas", /\* n precisa ter a propriedade cursos pq deixamos ela com uma '?' no tipo ALUNO que criamos \*/

    idade: 32,

}

alunos.push(novoAluno);   /\* add o novoAluno para a array\*/

function exibeAluno(aluno: aluno) {   /\* um aluno do tipo aluno\*/

    console.log(aluno.nome)

}

console.log(alunos);   /\* exibir toda a array aluno com todas as propriedades \*/

**26.6 - EXPLORE A ORIENTAÇÃO A OBJETOS**

**Slide: Modificadores de acesso**: A programação orientada a objetos no TypeScript também se faz presente, e ganhou recursos novos como o modificador de acesso **protected**. Além disso uma mudança grande em relação ao JavaScript é que no TypeScript é necessário **declarar as propriedades antes de inicializá-las no construtor.**

Os modificadores de acesso (3) são os responsáveis pelo encapsulamento de um membro em uma classe. No **JavaScript** possuímos os modificadores:

• **público**: padrão, acessível na classe, nas herdeiras e na instância;

• **privado**: acessível apenas na classe;

• **estático**: acessível na classe.

No **TypeScript** tivemos a introdução do modificador **protegido (4)**, que já é presente em outras linguagens de programação. Os modificadores no TypeScript são:

• público: padrão, acessível na classe, nas herdeiras e na instância;

• privado: acessível apenas dentro da propria classe, é o mais restrito;

• estático: acessível na classe, e não pelas instâncias;

**• protegido: acessível na classe e nas herdeiras dessa classe.**

**Gpt sobre NOVIDADES CAMPOS PRIVADOS:**

A partir do ECMAScript 2022, o JavaScript introduziu um recurso para criar propriedades realmente privadas, usando o símbolo **#** antes do nome da propriedade:

class Pessoa {

#nome; // Propriedade realmente privada

constructor(nome) {

this.#nome = nome;

}

get nome() {

return this.#nome;

}

set nome(novoNome) {

if (novoNome.length > 0) {

this.#nome = novoNome;

} else {

console.log("Nome inválido");

}

}

}

const pessoa = new Pessoa("João");

console.log(pessoa.#nome); // Erro: não é possível acessar #nome diretamente

**Aula:** Criar o arquivo oo.ts

class Pessoa {

    nome: string;   //na class construtora precisamos declarar a prop antes!!!

    renda?: number; // **?** Define a variável como opcional;

    constructor(nome: string, renda?: number) {   //sempre vir primeiro a obrigatoria, dps a opcional

        this.nome = nome;

        this.renda = renda;

    }

    dizOla(): string {

        return `${this.nome} disse oi`;

    }

}

class ContaBancaria { // usando os modificadores de acesso: private, public, protected, static.

    protected saldo: number = 0;   // **Protected** - Só pode ser acessado onde foi criado e pelas herdeiras;

    public numeroConta: number; // **Public** - Acessível em todas as instâncias e á padrao, nao precisa por!

    constructor(numeroDaConta: number) {

        this.numeroConta = numeroDaConta;

    }

    static retornaNumeroDoBanco() {   // **Static** - Pertence somente à classe ContaBancaria , logo não pode ser acessado por fora;

        return 125;

    }

    private getSaldo() {        // **Private** - Somente acessível dentro da classe em que foi criada, nem as herdeiras acessam  (mais restritivo);

        return this.saldo;       //get para acessar o saldo

    }

    depositar(valor: number) {   /\*era setSaldo. set para modificar o saldo \*/

        this.saldo += valor;   /\* += p manter o valor atual e add o valor \*/

    }

}

class ContaBancariaPessoaFisica **extends** ContaBancaria {  // ContaBancariaPessoaFisica é herdeira da ContaBancaria

    depositar(valor: number): void {   //foi feito no autocomplet do vscode void é vazio !

        this.saldo = valor \* 2;   /\* acessou SALDO pq o campo protegido permite q as herdeiras de acessem \*/

    }

}

//instancia da conta bancaria

const contaDoPedro = new ContaBancariaPessoaFisica(123456)

// contaDoPedro.

ContaBancaria.retornaNumeroDoBanco();

**GPT:** No TypeScript, os modificadores de acesso são usados para controlar a visibilidade de membros de classes (propriedades e métodos). Existem três modificadores principais:

1. **Publico:**

* É o padrão. Membros marcados como public podem ser acessados de qualquer lugar, tanto dentro quanto fora da classe.

class Pessoa {

public nome: string;

constructor(nome: string) {

this.nome = nome;

}

public cumprimentar() {

console.log(`Olá, meu nome é ${this.nome}.`);

}

}

const pessoa = new Pessoa('João'); **//INSTANCIA**

console.log(pessoa.nome); // joao - Acessível fora da classe

pessoa.cumprimentar(); // ola, meu nome é joao - Acessível fora da classe

1. **Private:**

* Membros marcados como private só podem ser **acessados dentro da própria classe.** Não podem ser acessados fora da classe nem por subclasses.

class Pessoa {

private idade: number;

constructor(idade: number) {

this.idade = idade;

}

public mostrarIdade() {

console.log(`Tenho ${this.idade} anos.`);

}

}

const pessoa = new Pessoa(30);

// console.log(pessoa.idade); **// Erro: 'idade' é privado e não pode ser acessado fora da classe**

pessoa.mostrarIdade(); // Tenho 30 anos - Acessível fora da classe

* + - 1. **Protegido**:
* Membros marcados como **protected** podem ser acessados dentro da classe e por classes derivadas (subclasses), mas não fora das classes.

class Animal {

protected especie: string;

constructor(especie: string) {

this.especie = especie;

}

protected mover() {

console.log(`${this.especie} está se movendo.`);

}

}

class Cachorro extends Animal { //cachorro herda de animal

public latir() {

console.log(`${this.especie} está latindo.`);

this.mover(); // Acessível na subclasse

}

}

const cachorro = new Cachorro('Cão');

// console.log(cachorro.especie); // Erro: 'especie' é protegido e não pode ser acessado fora da classe ou fora da subclasse

cachorro.latir(); // Acessível fora da classe

1. **Estático**

* **Acesso pela classe**: Membros estáticos são acessíveis diretamente pela classe, e não pelas instâncias.

class Matematica {

**static** PI: number = 3.14;

**static** calcularCircunferencia(raio: number): number {

return 2 \* this.PI \* raio;

}

}

console.log(Matematica.PI); // 3.14 Acessando a propriedade stática

console.log(Matematica.calcularCircunferencia(5)); // 31.400000000000002 Acessando o método estático

const matematica = new Matematica();

// console.log(matematica.PI); // Erro: 'PI' não é acessível através de uma instância

**Outro exemplo de estático:**

class Contador {

static contagem: number = 0;

constructor() {

Contador.contagem++;

}

static mostrarContagem() {

console.log(`Total de instâncias: ${this.contagem}`);

}

}

const c1 = new Contador();

const c2 = new Contador();

Contador.mostrarContagem(); // Mostra 'Total de instâncias: 2'

No ex, a propriedade contagem e o método **mostrarContagem** pertencem à classe **Contador**, não às instâncias individuais. Cada vez que uma nova instância de Contador é criada, o valor de contagem é incrementado. Como resultado, o método mostrarContagem exibe o número total de instâncias criadas, independentemente de qual instância está sendo utilizada.

**FUNÇÃO GET E SET**

**(Gpt)**

**Função get**

* **Objetivo:** A função get é usada para **acessar** o valor de uma propriedade de um objeto.
* **Uso:** É utilizada quando você deseja obter o valor de uma propriedade de forma segura, possivelmente aplicando lógica adicional ao retorno do valor.
* **Sintaxe:** A função get é definida **dentro** de uma classe, usando a palavra-chave get seguida pelo nome da propriedade. Dentro do método, você retorna o valor desejado.

**class Pessoa {**

**constructor(nome) {**

**this.\_nome = nome; //USOU O \_ ENTAO É PRIVADO?**

**}**

**get nome() { //QUER ACESSAR O VALOR DE NOME**

**return this.\_nome;**

**}**

**}**

**const pessoa = new Pessoa("João"); // INSTANCIA**

**console.log(pessoa.nome); // João**

**Função set**

* **Objetivo:** A função set é usada para **modificar** o valor de uma propriedade de um objeto.
* **Uso:** Ela é utilizada para atribuir um valor a uma propriedade de forma controlada, permitindo validações ou transformações antes de alterar o valor da propriedade.
* **Sintaxe:** A função set é definida **dentro** de uma classe, usando a palavra-chave set seguida pelo nome da propriedade. Dentro do método, você define o valor da propriedade, podendo aplicar lógica adicional, como validação.

**class Pessoa {**

**constructor(nome) {**

**this.\_nome = nome;**

**}**

**get nome() {**

**return this.\_nome;**

**}**

**set nome(novoNome) {**

**if (novoNome.length > 0) {**

**this.\_nome = novoNome;**

**} else {**

**console.log("Nome inválido");**

**}**

**}**

**}**

**const pessoa = new Pessoa("João");**

**pessoa.nome = "Carlos"; // Modifica o nome para "Carlos"**

**console.log(pessoa.nome); // Carlos**

**pessoa.nome = "";**  // Nome inválido

**Resumo**

* **get:** Retorna o valor de uma propriedade.
* **set:** **Modifica** o valor de uma propriedade, com a possibilidade de realizar validações ou transformações antes da atribuição.

Essas funções são muito úteis para encapsular e proteger dados dentro de uma classe, fornecendo um meio para interagir com as propriedades de objetos de forma controlada e segura.

**26.7 - CONHEÇA AS INTERFACES**

Slide: Uma outra nova funcionalidade que temos no TypeScript é a **Interface**. Ela já existe em linguagens de back-end. Uma interface basicamente é um contrato, onde temos que seguir algumas regras para que seja satisfeito. Por exemplo, um veículo, obrigatoriamente precisa ter a capacidade de acelerar e frear. **Na POO, programação orientada a objetos, só é permitido a herança de uma classe, ou seja, não podemos herdar de mais de uma classe, porém com as interfaces podemos implementar várias.**

Escrevemos uma interface assim:

interface **I**Veiculo { //uso o I antes do elemento

acelerar: () => boolean;

frear: () => boolean;

velocidadeMaxima: number;

}

Para implementar a interface **IVeiculo**, precisamos fazer a implementação dos **métodos (frear e acelerar)** e definir uma propriedade **velocidadeMaxima** que contenha um número:

class Carro **implements** IVeiculo {

velocidadeMaxima = 120;

acelerar() {..}

frear() {..}

}

As interfaces só estão disponíveis no typescript !

**Aula:** Criar o arquivo interface.ts.

class Conta {

    numeroDaConta: number;

    saldo: number = 0 ;

    constructor(numeroDaConta: number) {

        this.numeroDaConta = numeroDaConta;

    }

}

class ContaSalario extends Conta {

    depositar( valor: number) {   /\* faz a funcao de SET \*/

        this.saldo += valor;

    }

}

//criando a interface

**interface ITransacional {**

**transferir: (valor:number, destinatario: Conta) => boolean;**

**taxaTransferencia: number;**

**}**

class ContaCorrente extends Conta **implements** ITransacional {

    transferir (valor:number, destinatario: Conta) {

        destinatario.saldo += (valor - this.taxaTransferencia);

        return true

    };

    taxaTransferencia: number = 0;

}

**26.8 - Integre TypeScript com DOM**

**Typescritp no navegador:** O TypeScript também pode ser utilizado para adicionar comportamentos aos sites, afinal o seu produto final é um arquivo JavaScript. Por padrão a linguagem já possui alguns tipos pré-definidos, como **HTMLElement** quando utilizamos um **querySelector**, **getElementById** etc.

**Aula:** Criar o arquivo dom.ts.

const campoNome = document.getElementById('nome');

const formulario = document.getElementsByTagName('form');

const botaoEnviar = document.getElementById('btn-enviar');

botaoEnviar?.addEventListener('click', (e: MouseEvent) => {

})

//o ts add automaticamente a '?', pois assim garante que so vai usar o botao, caso tenha algum conteudo.

MouseEvent => selecionar ele e apertar F12 !

**Criar instâncias da interface**: **Não é possível.** Uma interface não pode ser instanciada diretamente porque ela não possui implementação, apenas declara os métodos que devem ser implementados por classes que a implementam.

**Compartilhar interfaces, como herança**: **É possível.** Interfaces podem herdar de outras interfaces, o que permite a criação de hierarquias de interfaces.

**Utilizar como tipos de dados**: **É possível.** Uma interface pode ser usada como tipo de dado, e qualquer classe que implemente essa interface pode ser tratada como esse tipo.

**Aplicar interfaces para serem implementadas por classes**: **É possível.** Esse é o principal uso das interfaces, que servem como um contrato que as classes devem seguir ao implementar os métodos definidos na interface

CONVERVÃO DE TIPOS:

parseFloat: Converte uma string para um número de ponto flutuante.

parseInt: Converte uma string para um número inteiro.

toString: Converte um número ou outro tipo de dado em uma string.