

Revisão P3

Node.js e NPM:

- O Node.js permite executar código JavaScript fora do navegador, abrindo portas para a criação de aplicações do lado do servidor, como APIs e aplicações web.
- O NPM (Node Package Manager) é crucial para gerenciar as dependências (bibliotecas) do projeto, facilitando a instalação, atualização e compartilhamento de código.

TypeScript como Superset do JavaScript:

- TypeScript é uma linguagem que adiciona tipagem estática ao JavaScript, proporcionando maior robustez e detecção de erros em tempo de desenvolvimento.
- A tipagem estática ajuda a evitar erros comuns, como atribuição de valores a variáveis de tipos incompatíveis.
- TypeScript oferece recursos avançados de POO, como classes, interfaces, herança, encapsulamento, polimorfismo, entre outros.

Estrutura de Projetos, Compilação e Execução:

- Um projeto Node.js com TypeScript geralmente segue uma estrutura de pastas bem definida, com diretórios como src (código fonte), public (arquivos estáticos), node_modules (dependências).
- O arquivo package.json é fundamental para gerenciar as configurações, dependências e scripts do projeto.
- O compilador TypeScript (tsc) converte o código TypeScript em JavaScript, permitindo que seja executado pelo Node.js.
- Ferramentas como ts-node facilitam a execução direta do código TypeScript sem a necessidade de compilação prévia.

Classes, Objetos e Herança:

- Classes servem como modelos para a criação de objetos, definindo suas propriedades (atributos) e métodos (comportamentos).

Exemplo de uma classe:

```
class Pessoa { // nome da classe, mesmo nome arquivo
  nome:string; // atributo
  idade:number; // atributo
  constructor(nome:string, idade:number){ // método construtor e
                                          // seus parâmetros
    this.nome = nome; // atributo recebe parâmetro
    this.idade = idade; // atributo recebe parâmetro
  }
  imprimir(){ // método imprimir
    console.log(`${this.nome} possui ${this.idade} anos`);
  }
}

const pessoa = new Pessoa("Ana", 21); // instanciando um objeto
```

- O construtor é um método especial usado para inicializar os objetos quando eles são criados.

- A herança permite criar classes (subclasses) que herdam características e comportamentos de classes existentes (superclasses), promovendo a reutilização de código e a organização hierárquica.

Exemplo Herança:

```
class Pessoa {
    nome:string = "";
    idade:number = 0;
    constructor(nome:string, idade:number) {
        this.nome = nome;
        this.idade = idade;
    }
}

class Cliente extends Pessoa { // classe Cliente herda classe Pessoa
    saldo: number;
    constructor(nome:string, idade:number, saldo:number) {
        // super é o construtor da classe herdada/estendida
        // por este, motivo temos de passar os parâmetros
        // do construtor da classe base
        super(nome,idade);
        this.saldo = saldo;
    }
    print():void {
        console.log(`${this.nome} - ${this.idade} - ${this.saldo}`);
    }
}

const c = new Cliente("Ana",18,980); // instanciando objeto Cliente
c.print();
```

Conceitos Importantes da POO:

- **Polimorfismo:** Capacidade de objetos de diferentes classes responderem ao mesmo método de maneiras diferentes, proporcionando flexibilidade e extensibilidade.
- **Sobrescrita:** Redefinição de um método herdado de uma superclasse em uma subclasse, adaptando o comportamento para a classe específica.

Exemplo Sobrescrita:

```
class A {
    nome: string;
    constructor(nome:string) {
        this.nome = nome.toUpperCase();
    }
    print():void {
        console.log("Classe A:", this.nome);
    }
}

class B extends A {
    // sobreescreve a propriedade nome da classe A
    nome: string;
    constructor(nome:string) {
        super(nome);
        this.nome = nome.toLowerCase();
    }
    // sobreescreve o método print da classe A
    print():void {
```

```
        console.log("Classe B:", this.nome);
    }
    imprimir():void{
        // chama o método print da superclasse
        super.print();
    }
}
const a = new A("Tipo A");
a.print();
const b = new B("Tipo B");
b.print();
b.imprimir();
```

- **Sobrecarga:** Criação de múltiplos métodos com o mesmo nome, mas com diferentes parâmetros, proporcionando diferentes formas de usar um método.

Exemplo de sobrecarga:

```
class Teste {
    somar(a: number, b: number): number;
    somar(a: string, b: string, c: string): string;
    somar(a: string, b: string): string;
    somar(a: any, b: any, c?: any): any {
        if (c !== undefined) {
            return a + b + c;
        } else {
            return a + b;
        }
    }
}

const t = new Teste();
// usa a assinatura somar(a: number, b: number): number
console.log(t.somar(2, 3));
// usa a assinatura somar(a: string, b: string, c: string): string;
console.log(t.somar("x", "y", "z"));
// usa a assinatura somar(a: string, b: string): string;
console.log(t.somar("x", "y"));
```

Classes Abstratas e Interfaces:

- Classes abstratas fornecem uma estrutura base para subclasses, mas não podem ser instanciadas diretamente, contendo métodos abstratos que devem ser implementados pelas subclasses.

Exemplo de classe abstrata:

```
abstract class Pessoa {
    protected nome: string;
    protected idade: number;
    constructor(nome: string, idade: number) {
        this.nome = nome;
        this.idade = idade;
    }
    // um método abstrato não possui corpo
    public abstract print(): void;
}
// errado: uma classe abstrata não pode ser instanciada usando new
const p = new Pessoa("Ana", 18);
```

```
p.print(); // errado: o método print não possui corpo

class Cliente extends Pessoa {
    private saldo:number;
    constructor(nome:string, idade:number, saldo:number){
        super(nome,idade);
        this.saldo = saldo;
    }
    public print(): void {
        console.log(this.nome, this.idade, this.saldo);
    }
}
// certo: a classe Cliente é concreta
const c = new Cliente("Ana", 18, 950);
c.print(); // certo: o método print possui implementação na classe
Cliente
```

- Interfaces definem contratos que as classes devem seguir, especificando os membros que devem ser implementados, garantindo a compatibilidade entre objetos.

Exemplo Interface:

```
interface Pessoa {
    nome: string;
    idade: number;
    print(): void;
}

class Cliente implements Pessoa {
    private saldo:number;
    public nome:string;
    public idade:number;
    constructor(nome:string, idade:number, saldo:number){
        this.nome = nome;
        this.idade = idade;
        this.saldo = saldo;
    }
    public print(): void {
        console.log(this.nome, this.idade, this.saldo);
    }
    public incrementar(): void {
        this.idade++;
    }
}

const cli = new Cliente("Ana", 18, 950);
cli.incrementar(); //correto: o tipo Cliente possui o método incrementar
cli.print(); //correto: o tipo Cliente possui o método print
```

Tratamento de Exceções:

- O tratamento de exceções permite lidar com erros que ocorrem durante a execução do código, evitando que o programa seja interrompido abruptamente.
- O bloco try...catch é usado para capturar e tratar as exceções, enquanto o bloco finally garante que certas ações sejam executadas independentemente da ocorrência de erros.

Exemplo try/catch/finally

```
try {
    const resultado = imc(70, 0);
    // esta instrução não será executada se for lançada uma exceção
    console.log("Resultado:", resultado);
} catch (e:any) { // o erro lançado será recebido no parâmetro e
    // esta instrução será executada se for lançada uma exceção
    // a propriedade message possui a mensagem do objeto Error
    console.log("Exceção:", e.message);
} finally {
    console.log("Passa por aqui");
}
```

Array de Objetos:

```
class Carro {
    fabricante:string;
    modelo: string;
    ano:number;
    constructor(fabricante:string, modelo: string, ano:number){
        this.fabricante = fabricante;
        this.modelo = modelo;
        this.ano = ano;
    }
}

export default Carro;

import Carro from "./Carro";

var carros: Array<Carro> = [];
var carro = new Carro("Fiat", "Uno", 2010);
carros.push(carro);
var carro = new Carro("Chevrolet", "Onix", 2016);
carros.push(carro);
var carro = new Carro("Hyundai", "HB20", 2024);
carros.push(carro);

carros.forEach(carro => {
    console.log("<< CARRO >>");
    console.log("Fabricante:", carro.fabricante);
    console.log("Modelo:", carro.modelo);
    console.log("Ano:", carro.ano);
})
```

Saída:

```
<< CARRO >>
Fabricante: Fiat
Modelo: Uno
Ano: 2010
<< CARRO >>
Fabricante: Chevrolet
Modelo: Onix
Ano: 2016
```

```
<< CARRO >>
Fabricante: Hyundai
Modelo: HB20
Ano: 2024
```

Máscara de saída (impressão)

```
var cep = "12309500";
console.log(cepMask(cep));

function cepMask(v: string | undefined) {
  if (v == undefined) {
    return
  }
  let r = v.replace(/\D/g, "");
  r = r.replace(/^0/, "");
  if (r.length == 8) {
    r = r.replace(/^(\d{5})(\d{3}).*/, "$1-$2");
  }
  return r;
}
```

Saída: 12309-500

Regex Validação:

```
let regex = /^[0-9]{10,11}$/;

var value = "12345678910";

console.log(regex.test(value)); // true - válido

var value = "1234567891a";

console.log(regex.test(value)); // false - inválido

var value = "123456789";

console.log(regex.test(value)); // false - inválido
```

Formatação de valores:

```
var nota = 8.3;

console.log("Nota:", nota.toFixed(2));

var saldo = 1553.83;

console.log("Saldo:", saldo.toLocaleString('pt-BR', { style: 'currency',
currency: 'BRL' }));
```

Saída:

```
Nota: 8.30
Saldo: R$ 1.553,83
```