Objetivo:

I. Herança;
II. Sobrescrita;
III. Sobrecarga;
IV. Polimorfismo.

Observação: antes de começar, crie um projeto para reproduzir os exemplos. Dica, use os passos da Aula 2 para criar o projeto de exemplo.

I. Herança

A herança é um recurso da POO (Programação Orientada a Objetos) que permite que uma classe herde propriedades e métodos de outra classe. Isso permite criar uma hierarquia de classes, onde as subclasses herdam o comportamento e as características da superclasse.

Na programação, a herança é codificada usando a palavra reservada `extends` (estender) seguida pelo nome da classe a ser herdada. No exemplo a seguir a instrução

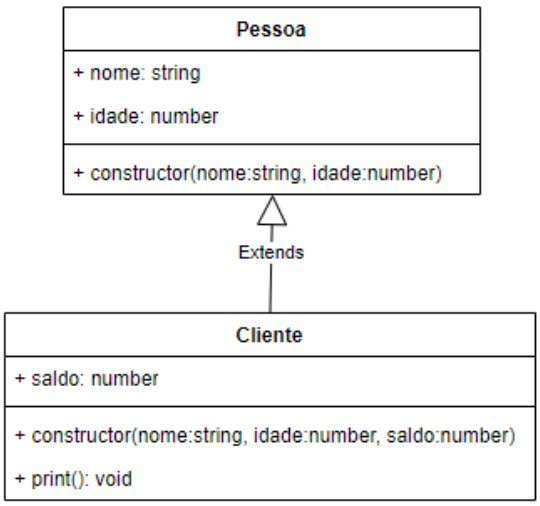
```
class Cliente extends Pessoa
```

é usada para definir que a classe `Cliente` herda/estende da classe `Pessoa`.

```
class Pessoa {
    nome:string = "";
    idade:number = 0;

    constructor(nome:string, idade:number){
        this.nome = nome;
        this.idade = idade;
    }
}

class Cliente extends 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}`);
  }
}
```

Na UML a herança é representada por uma seta vazada.

A subclasse é também chamada de classe filha e a superclasse é também chamada de classe base ou classe pai.



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

No exemplo anterior, a existência da classe `Pessoa` independe da existência da classe `Cliente`, ou seja,

```
new Pessoa("Pedro",20)
```

construirá na memória do computador somente um objeto do tipo `Pessoa`. Porém, a construção de um objeto `Cliente`

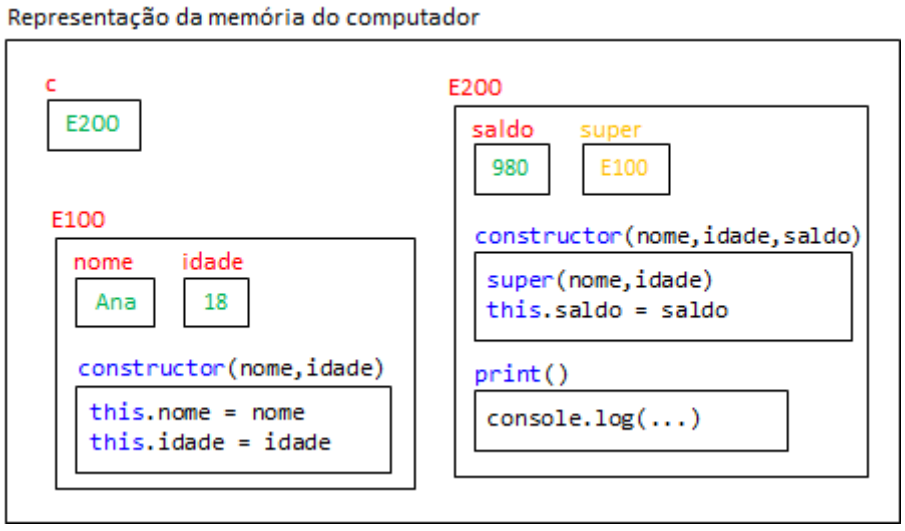
```
new Cliente("Ana",18,980)
```

requer antes a construção de um objeto `Pessoa`. A seguir tem-se os passos da representação de um objeto `Cliente` na memória do computador:

1. A 1ª instrução dentro do construtor da subclasse precisa ser a chamada do construtor da superclasse – `super()`. Desta forma, será alocado na memória um objeto do tipo `Pessoa`. Na ilustração a seguir, esse objeto está no endereço E100. A seguir tem-se a ordem de inicialização da superclasse:
    a. As propriedades da superclasse são inicializadas com os valores definidos na criação das propriedades. Nesse exemplo são os valores marcados em amarelo:

    ```
    nome: string = "";
    idade: number = 0;
    ```

    b. O construtor da superclasse é executado: as instruções `this.nome = nome` e `this.idade = idade` atribuem os valores nas propriedades do objeto E100.
2. Somente após construir o objeto da superclasse que será construído o objeto da subclasse. Na ilustração a seguir, esse objeto está no endereço E200. Dentro do objeto da subclasse, a palavra reservada `super` possui a referência para a superclasse, aqui representada pelo endereço E100. A seguir tem-se a ordem de inicialização da subclasse:
    a. As propriedades da subclasse são inicializadas. Nesse exemplo a propriedade `saldo` não será inicializada pelo fato dela não ter recebido valor na criação;
    b. O construtor da subclasse é executado: a instrução `this.saldo = saldo` atribui valor na propriedade do objeto E200.



Observação: é importante ressaltar que a representação exata de objetos e superclasses na memória do computador podem variar entre as linguagens de programação. Cada linguagem de programação e ambiente de execução têm suas próprias estruturas de dados e mecanismos para gerenciar a alocação de memória.

Possíveis causas de erro no construtor da subclasse:

- Errado pelo fato de não ter a chamada do construtor da superclasse:

```
constructor(nome:string, idade:number, saldo:number){
    this.saldo = saldo;
}
```

- Errado pelo fato do objeto `this` ser chamado antes do `super`:

```
constructor(nome:string, idade:number, saldo:number){
    this.saldo = saldo;
    super(nome, idade);
}
```

Considerações sobre o objeto `super`:

- A palavra reservada `super` é usada na subclasse para referenciar o seu objeto da superclasse. Porém, `super.nome` resultará em `undefined`, pois as propriedades da superclasse são consideradas propriedades do objeto `this`, logo não estão disponíveis pelo objeto `super`:

```
print(): void {
    console.log(`${super.nome} - ${super.idade} - ${this.saldo}`);
}
```

- No exemplo a seguir adicionou-se o método `imprimir` na superclasse. Para chamar esse método na subclasse é necessário preceder o nome do método pela palavra reservada `super` ou `this`. Pelo princípio da herança o objeto `this` possui as propriedades e métodos da superclasse:

```
class Pessoa {
    nome: string;
    idade: number;

    constructor(nome: string, idade: number) {
        this.nome = nome;
        this.idade = idade;
    }

    imprimir():void {
        console.log(`${this.nome} - ${this.idade}`);
    }
}

class Cliente extends Pessoa {
    saldo: number;
```

Exemplo de saída:

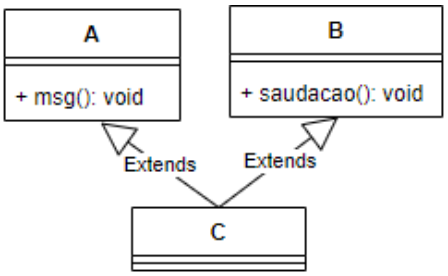
```
PS C:\Desktop\exemplo> npm start
> exemplo@1.0.0 start
> ts-node ./src/index
Ana - 18
Ana - 18
Ana - 18 - 980
```

```
    constructor(nome: string, idade: number, saldo: number) {
        super(nome, idade);
        this.saldo = saldo;
    }

    print(): void {
        this.imprimir();
        super.imprimir();
        console.log(`${this.nome} - ${this.idade} - ${this.saldo}`);
    }
}

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

Observações:

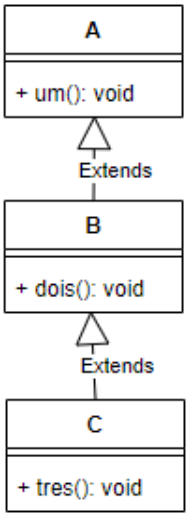
- A superclasse não tem a capacidade de saber quais são as classes filhas;
- A linguagem TS não suporta herança múltipla, isto é, uma subclasse herdar de mais de uma superclasse. O código a seguir apresentará erro, pois a classe `C` está herdando das classes `A` e `B`:

```
class A {
    msg():void {
        console.log("bom dia");
    }
}

class B {
    saudacao():void {
        console.log("boa noite");
    }
}

class C extends A,B {
}
```



- As classes podem herdar hierarquicamente. No exemplo a seguir os objetos da classe C possuem os métodos das classes `A` e `B`. Isso acontece pelo fato dos métodos da classe `A` serem herdados pela `B` e estes, por sua vez, são herdados pela classe `C`:

```
class A {
    um(){
        console.log("bom dia");
    }
}

class B extends A {
    dois(){
        console.log("boa tarde");
    }
}

class C extends B {
    tres(){
        console.log("boa noite");
    }
}

const ce = new C();
ce.um();
ce.dois();
ce.tres();
```



O principal papel da herança na POO é possibilitar a reutilização de código. O programador pode utilizar comportamentos já codificados ao programar novas funcionalidades. Imagine a necessidade de codificar uma classe `Aluno`, poderíamos estender da classe `Pessoa` e evitar ter de programar as propriedades e métodos existentes na classe `Pessoa`.

II. Sobrescrita

A sobrescrita (overriding, em inglês) é um conceito da POO que permite que uma subclasse substitua uma propriedade ou método da classe base com sua própria implementação. Isso significa que a classe filha fornece uma implementação diferente para um método já definido na classe pai.

No exemplo a seguir, a classe `B` sobrescreve a propriedade `nome` e o método `print` da classe `A`. Desta forma, não é possível acessar o método `print` da superclasse a partir do objeto da classe `B`, pois a instrução `b.print()` chamará o método `print` da classe `B`.

Para chamar o método `print` da superclasse é necessário que a subclasse tenha outro método. Neste exemplo, temos o método `imprimir`.

```
class A {
    nome: string;

    constructor(nome:string){
        this.nome = nome.toUpperCase();
    }

    print():void {
        console.log("Classe A:", this.nome);
```

Exemplo de saída:

```
PS C:\Desktop\exemplo> npm start
> exemplo@1.0.0 start
> ts-node ./src/index
Classe A: TIPO A
Classe B: tipo b
Classe A: tipo b
```

```
    }
}

class B extends A {
    // sobrescreve a propriedade nome da classe A
    nome: string;

    constructor(nome:string){
        super(nome);
        this.nome = nome.toLowerCase();
    }

    // sobrescreve 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();
```

As principais características da sobrescrita são:

- Deve haver uma relação de herança: a sobrescrita ocorre entre uma classe pai e uma classe filha que herda o método. No exemplo anterior a classe `B` herdou o método `print` da classe `A`;
- Assinatura do método deve ser a mesma: a classe filha deve declarar um método com a mesma assinatura (nome e parâmetros) do método da classe pai que está sendo sobrescrito. Isso garante que o método na classe filha substitua o método na classe pai. No exemplo anterior foi definido o método `print` na classe `B` com a mesma assinatura do método `print` da superclasse;
- Acesso ao método da classe pai: dentro da implementação do método na classe filha, é possível acessar o método da classe pai usando a palavra reservada `super`. Isso permite que a classe filha chame o método da classe pai antes ou depois de adicionar sua própria lógica. No exemplo anterior a instrução `super.print()`, no método `imprimir` da classe `B`, é usado para chamar o método `print` da superclasse.

> A assinatura de um método é formada pelo nome do método e parâmetros. Se dois métodos possuem o mesmo nome e ordem dos parâmetros, então eles possuem a mesma assinatura.

III. Sobrecarga

A sobrecarga (overloading, em inglês) é um conceito da POO que permite que uma classe tenha vários métodos com o mesmo nome, mas com assinaturas diferentes. Isso significa que uma classe pode ter métodos com o mesmo nome, mas com diferentes parâmetros ou tipos de retorno.

No TS a sobrecarga do método é definida criando várias assinaturas para o método com o mesmo nome, mas com diferentes parâmetros ou tipos de retorno. A implementação real do método deve ser a última assinatura definida (no exemplo a seguir a implementação real está marcada em amarelo). No exemplo a seguir o TS selecionará a assinatura correspondente com base nos argumentos passados:

```
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"));
```

A sobrecarga de métodos no TS é uma maneira útil de definir diferentes comportamentos para um mesmo nome de método, dependendo dos parâmetros fornecidos.

Os construtores também podem ser sobrecarregados:

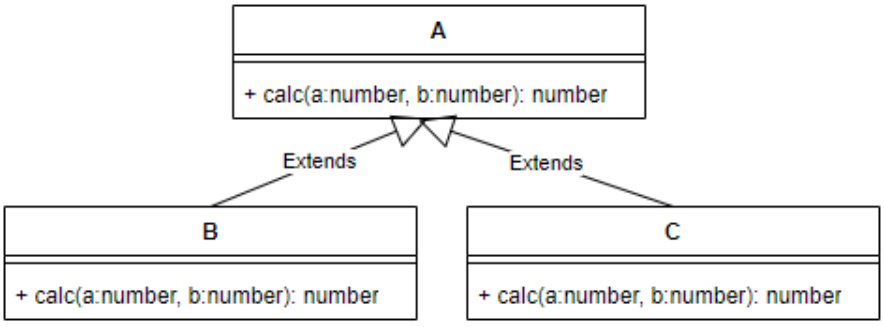
```
class Operacao {
    x: any;
    y: any;

    constructor(x: number, y: number);
    constructor(x: string, y: string);
    constructor(x: any, y: any) {
        this.x = x;
        this.y = y;
    }

    somar(): any {
        return this.x + this.y;
    }
```

```
}

// usa a assinatura Operacao(number,number)
const a = new Operacao(5, 3);
console.log("x + y:", a.somar());
// usa a assinatura Operacao(string,string)
const b = new Operacao("4", "1");
console.log("x + y:", b.somar());
```

IV. Polimorfismo

O termo polimorfismo é originário do grego e significa muitas formas. Na POO o polimorfismo permite que classes mais abstratas (no exemplo a seguir seria a classe A) representem o comportamento de classes concretas (classes B e C), em outras palavras, permite escrever programas que processam objetos que compartilham a mesma superclasse em uma hierarquia de classes como se todos fossem objetos da superclasse.

No exemplo a seguir fez-se a chamada do método `calc` duas vezes usando os parâmetros `4,5`, mas em cada caso está sendo chamado o método de um objeto diferente – este conceito é o polimorfismo, onde a mesma instrução possui execuções distintas.




```
class A {
    calc(a:number, b:number):number {
        return a + b;
    }

    teste():number {
        return 1;
    }
}

class B extends A {
    calc(a:number, b:number):number {
        return a - b;
    }

}

class C extends A {
    calc(a:number, b:number):number {
        return a * b;
```

```
    }
}

let op = new B();
console.log(op.calc(4,5)); //chama o método calc da classe B
console.log(op.teste());  //chama o método teste da classe A
op = new C();
console.log(op.calc(4,5)); //chama o método calc da classe C
console.log(op.teste());  //chama o método teste da class A
```

A sobrescrita é uma das principais técnicas utilizadas na POO para alcançar o polimorfismo, onde um objeto pode se comportar de maneira diferente com base no tipo específico da classe a que pertence.

Exercícios

Veja o vídeo se tiver dúvidas nos exercícios: https://youtu.be/Ff_Fo538i6Q

Instruções para criar o projeto e fazer os exercícios:

1. Crie a pasta `aula3` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
2. Abra a pasta no VS Code e acesse o terminal do VS Code;
3. No terminal, execute o comando `npm init -y` para criar o arquivo package.json:
4. No terminal, execute o comando `npm i -D ts-node typescript` para instalar os pacotes `ts-node` e `typescript` como dependências de desenvolvimento;
5. Crie uma pasta de nome `src` na raiz do projeto e crie os arquivos dos exercícios nela – assim como é mostrado na figura ao lado;
6. Adicione na propriedade `scripts` do `package.json` os comandos para executar cada exercício. Como exemplo, aqui estão os comandos para executar os dois primeiros exercícios:


```
{
  "name": "aula3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "um": "ts-node ./src/exercicio1",
    "dois": "ts-node ./src/exercicio2"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.9.1",
    "typescript": "^5.1.6"
  }
}
```

Observação: se o comando `ts-node` não funcionar, então use o programa `npx` para executar o comando `ts-node`:

```
"scripts": {
  "um": "npx ts-node ./src/exercicio1",
  "dois": "npx ts-node ./src/exercicio2"
},
```

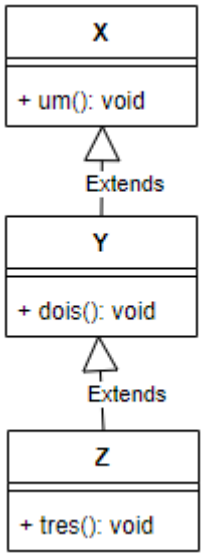
Exercício 1: Completar o código a seguir para produzir o resultado mostrado ao lado. O método somar da classe Calcular deverá usar a operação de soma implementada na superclasse.

```
class Operacao {
    somar(a:number,b:number):number {
        return a + b;
    }
}

class Calcular extends Operacao {
    x: number;
    y: number;

    constructor(x:number, y:number) {
       //completar aqui
    }

    somar(): number {
       //completar aqui
    }
}

const c = new Calcular(5,15);
console.log("Somar:", c.somar());
```

Exemplo de saída:

```
PS D:\aula3> npm run um
> aula3@1.0.0 um
> ts-node ./src/exercicio1
Somar: 20
```

Exercício 2: Codificar as classes representadas no diagrama UML de classes. Codificar em cada método as seguintes instruções:



No corpo do método um:

```
console.log("um");
```

No corpo do método dois. Incluir também a chamada do método um:

```
console.log("dois");
```

No corpo do método tres. Incluir também a chamada do método dois:

```
console.log("três");
```

Utilize o objeto a seguir para testar o código e produzir o resultado mostrado ao lado.

```
const z = new Z();
z.tres();
```

Exemplo de saída:

```
PS D:\aula3> npm run dois
> aula3@1.0.0 dois
> ts-node ./src/exercicio2
um
dois
três
```

Exercício 3: Completar o código dos métodos incrementar para o resultado dar 6 ao chamar o método incrementar do objeto que está na variável teste.

```
class Um {
    nro: number = 0;

    incrementar(): void {
        this.nro++;
    }
}

class Dois extends Um {
    incrementar(): void {
        this.nro += 2;
    }
}

class Tres extends Dois {
    incrementar(): void {
        this.nro += 3;
    }
}

const teste = new Tres();
teste.incrementar();
console.log("Nro:", teste.nro);
```

Exemplo de saída:

```
PS D:\aula3> npm run tres
> aula3@1.0.0 tres
> ts-node ./src/exercicio3
Nro: 6
```

Exercício 4: Utilizando a sobrecarga de construtores. Complete o código a seguir adicionando assinaturas de construtores para atender as instanciações marcadas em amarelo.

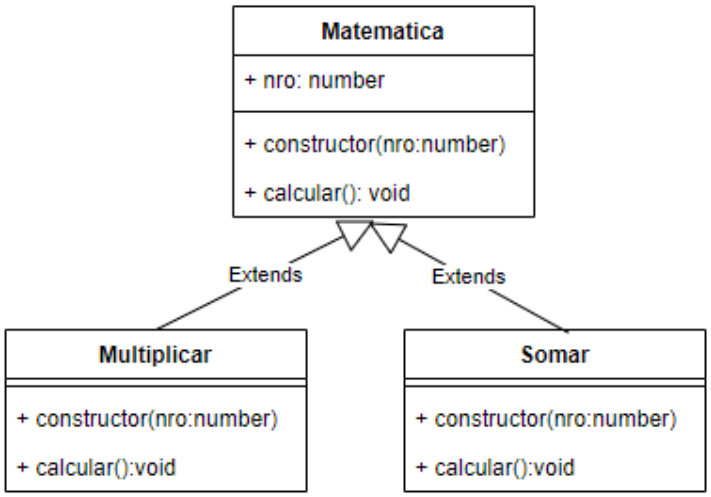
Observação: só é permitido adicionar construtores na classe Carro.

```
class Carro {
    marca?: string;
    modelo?: string;
    ano?: number;

    print(): void {
        console.log(this);
    }
}

const g = new Carro("VW","Gol",2010);
g.print();
const f = new Carro("Fiat", "Uno");
f.print();
const v = new Carro();
v.print();
```

Exemplo de saída:

```
PS D:\aula3> npm run quatro
> aula3@1.0.0 quatro
> ts-node ./src/exercicio4
Carro { marca: 'VW', modelo: 'Gol', ano: 2010 }
Carro { marca: 'Fiat', modelo: 'Uno' }
Carro {}
```

Exercício 5: O relacionamento entre as classes gera um polimorfismo, onde os objetos das subclasses sobrescrevem o comportamento do método calcular da superclasse. Completar o código a seguir codificando os construtores das classes.



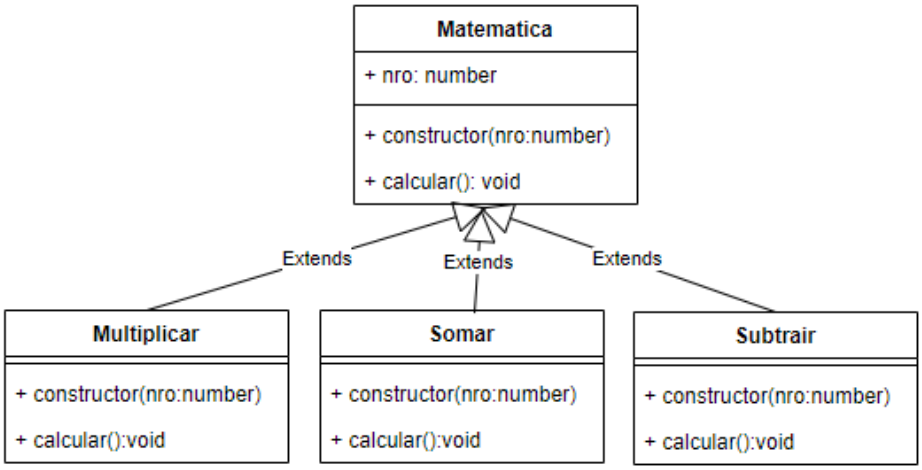

```
class Matematica {
    nro: number;

    calcular():void{}
}
```

Exemplo de saída:

```
PS D:\aula3> npm run cinco
> aula3@1.0.0 cinco
> ts-node ./src/exercicio5
5 + 0 = 5
5 + 1 = 6
5 + 2 = 7
5 + 3 = 8
5 + 4 = 9
5 + 5 = 10
5 + 6 = 11
5 + 7 = 12
5 + 8 = 13
5 + 9 = 14
5 + 10 = 15
5 * 0 = 0
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

```
class Multiplicar extends Matematica {
    calcular(): void {
        for (let i = 0; i <= 10; i++) {
            console.log(`${this.nro} * ${i} = ${this.nro * i}`);
        }
    }
}

class Somar extends Matematica {
    calcular(): void {
        for (let i = 0; i <= 10; i++) {
            console.log(`${this.nro} + ${i} = ${this.nro + i}`);
        }
    }
}

const a:Matematica = new Somar(5);
a.calcular();
const b:Matematica = new Multiplicar(5);
b.calcular();
```

Exercício 6: Adicionar a classe Subtrair no código do Exercício 5.



Exercício 7: Modificar o código das classes a seguir para fazer uso da herança e aproveitar o código da superclasse.

```
class Veiculo {
    valor: number;

    constructor(valor:number){
        this.valor = valor;
    }

    print():void {
```

Exemplo de saída:

```
PS D:\aula3> npm run sete
> aula3@1.0.0 sete
> ts-node ./src/exercicio7
GM - 12500
Mangalarga - 4500
```

```
        console.log(`${this.valor}`);
    }
}

class Automovel {
    fabricante: string;
    valor: number;

    constructor(fabricante:string, valor:number){
        this.fabricante = fabricante;
        this.valor = valor;
    }

    print():void {
        console.log(`${this.fabricante} - ${this.valor}`);
    }
}

class Cavalo {
    raca: string;
    valor: number;

    constructor(raca:string, valor:number){
        this.raca = raca;
        this.valor = valor;
    }

    print():void {
        console.log(`${this.raca} - ${this.valor}`);
    }
}

const auto = new Automovel("GM", 12500);
auto.print();
const cavalo = new Cavalo("Mangalarga", 4500);
cavalo.print();
```

Exercício 8: Adicionar no código a herança para as classes fazerem uso do polimorfismo. Observe que a variável geom é do tipo genérico Geometria.

```
class Geometria {
    area():number {
        return 0;
    }
}

class Retangulo  {
    base: number;
```

Exemplo de saída:

```
PS D:\aula3> npm run oito
> aula3@1.0.0 oito
> ts-node ./src/exercicio8
Retângulo: 50
Círculo: 78.53981633974483
```

```
    altura: number;

    constructor(base:number, altura:number){
        this.base = base;
        this.altura = altura;
    }

    area():number {
        return this.base * this.altura;
    }
}

class Circulo {
    raio: number;

    constructor(raio: number) {
        this.raio = raio;
    }

    area(): number {
        return Math.PI * this.raio ** 2;
    }
}

let geom:Geometria = new Retangulo(10,5);
console.log("Retângulo:", geom.area());
geom = new Circulo(5);
console.log("Círculo:", geom.area());
```

Exercício 9: Adicionar a classe Quadrado no código do Exercício 8.

A área do quadrado é `lado x lado`.

```
let geom:Geometria = new Retangulo(10,5);
console.log("Retângulo:", geom.area());
geom = new Circulo(5);
console.log("Círculo:", geom.area());
geom = new Quadrado(5); // lado do quadrado
console.log("Quadrado:", geom.area());
```

Exemplo de saída:

```
PS D:\aula3> npm run nove
> aula3@1.0.0 nove
> ts-node ./src/exercicio9
Retângulo: 50
Círculo: 78.53981633974483
Quadrado: 25
```

Exercício 10: No TS, não é possível herdar diretamente da classe string, pois string (s minúsculo) é um tipo primitivo e não uma classe. Aqui, criamos a classe Texto que estende String (S maiúsculo). A classe Texto possui o comportamento disponível na classe String.

```
class Texto extends String {
    primeira():string {
        return this.charAt(0);
    }
```

Exemplo de saída:

```
PS D:\aula3> npm run dez
> aula3@1.0.0 dez
> ts-node ./src/exercicio10
Quantidade: 9
Primeira: B
Última: e
Minúscula: boa noite
```

```
    ultima():string {
        return this.charAt(this.length-1);
    }
}
```

Crie um objeto do tipo Texto fornecendo “Boa noite”, na sequência imprima no console:

- A quantidade de letras do texto “Boa noite”;
- A primeira letra do texto. Use o método primeira;
- A última letra do texto. Use o método ultima;
- O texto fornecido em minúsculo.