Objetivo:

I. Modificador static;
II. Modificador readonly;
III. Modificadores de visibilidade;
IV. Getters e Setters;
V. Propriedades de parâmetro.

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

> Os objetos na POO são como caixas e os membros (propriedades e métodos) são como tomadas de acesso externo ao interior do objeto. Ao longo dessa aula veremos opções para bloquear o acesso aos membros.

I. Modificador static

O modificador `static` é usado para definir membros (propriedades e métodos) estáticos em uma classe. Membros estáticos são associados à própria classe, em vez de instâncias individuais da classe. Isso significa que eles podem ser acessados diretamente na classe, sem a necessidade de criar uma instância da classe.

Errado: a propriedade `quantidade` não existe no tipo de dado Contador. Desta forma, ela não pode ser acessada pelo objeto `this` ou qualquer outro objeto da classe `Contador`:

```
class Contador {
    static quantidade: number = 0;

    incrementar():void {
//errado: quantidade não existe no objeto this
        this.quantidade++;
    }
}

const cont = new Contador();
// errado: quantidade não existe no objeto cont
console.log(cont.quantidade);
```

Certo: a propriedade `quantidade` só pode ser acessada usando a classe `Contador`. Observe que é possível acessar uma propriedade `static` antes mesmo de ter sido criado algum objeto da classe:

```
class Contador {
    static quantidade: number = 0;

    incrementar():void {
        // certo: quantidade existe na classe
        Contador.quantidade++;
    }
}

//acessada antes de ter sido criado algum objeto
console.log(Contador.quantidade);

const cont = new Contador();
// certo: quantidade existe na classe
console.log(Contador.quantidade);
```

Mesmo dentro do construtor é necessário usar a classe para acessar uma propriedade estática:

```
class Contador {
```

```
    static quantidade: number = 0;

    constructor(){
        Contador.quantidade = 12;
    }
}
```

Os métodos estáticos também precisam ser acessados através da classe. No exemplo a seguir, o método somar só pode ser acessado através da classe:

```
class Operar {
    static somar(x: number, y: number): void {
        console.log("Soma:", x+y);
    }
}

// correto: acessado através da classe
Operar.somar(10,5);
const op = new Operar();
// errado: não pode ser acessado através do objeto
op.somar(2,3);
```

Um membro `static` não pode ser chamado pela instância/objeto. Ele só pode ser invocado diretamente pela classe, pelo motivo do membro pertencer a classe (matriz) e não ao objeto (cópia).

Os membros estáticos são úteis para armazenar informações ou métodos que são compartilhadas por todas as instâncias de uma classe ou para definir métodos utilitários.

Uma classe pode ter vários métodos, mas as vezes precisamos somente de um deles, nesse contexto não seria bom ter de instanciar a classe para consumir um método. Para resolver esse problema, podemos ter apenas métodos estáticos na classe, desta forma, consumimos o método sem instanciar. Uma classe que possui apenas métodos estáticos é chamada de classe utilitária.

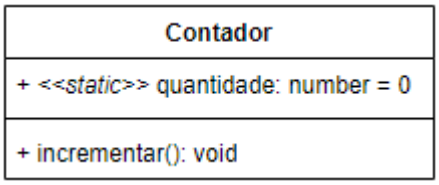
Os membros estáticos só estarão disponíveis na subclasse através do nome da classe. No exemplo a seguir a propriedade `quantidade` está disponível usando a superclasse (`Contador.quantidade`) ou subclasse (`Pessoa.quantidade`).

```
class Contador {
    static quantidade: number = 0;

    incrementar():void {
        Contador.quantidade++;
    }
}

class Pessoa extends Contador {
```

```
    nome: string;

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

const a = new Pessoa("Ana");
const b = new Pessoa("Pedro");
// errado: quantidade não existe no objeto do tipo Pessoa
//console.log(a.quantidade);
// certo: quantidade existe na classe Pessoa
console.log(Pessoa.quantidade);
// certo: quantidade existe na classe Contador
console.log(Contador.quantidade);
```

No diagrama de classes UML o modificador `static` é representado pelo sublinhado ou pelo estereótipo *<<static>>* à esquerda do nome do membro. Na ferramenta draw.io o static é representado pelo estereótipo *<<static>>*. No exemplo ao lado a propriedade `quantidade` é estática.

No exemplo ao lado a propriedade `quantidade` é inicializada com o valor zero. Na UML a inicialização de uma propriedade é representada pelo sinal de atribuição seguido pelo valor (`= 0`, neste caso).



II. Modificador readonly

No TS, o modificador `readonly` é usado para declarar propriedades de uma classe como somente leitura. Isso significa que o valor de uma propriedade `readonly` não pode ser alterado fora do construtor. No exemplo a seguir as atribuições marcadas em amarelo estão erradas pelo fato delas ocorrerem fora do construtor.

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

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

    alterar():void {
        // erro de compilação: a propriedade nome não pode ser alterada
        this.nome = "Aninha";
    }
}
```

```
const p = new Pessoa("Ana",18);
p.idade++;
// erro de compilação: a propriedade nome não pode ser alterada
p.nome = "Ana Maria";
// correto: nome está sendo acessado para leitura
console.log(p.nome);
```

Mesmo se a propriedade tiver sido inicializada na declaração (assim como está marcado em ciano). Ela pode ser alterada somente no construtor.

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

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

    alterar(): void {
        // errado: a propriedade nome não pode ser alterada
        this.nome = "Aninha";
    }
}
```

O modificador `readonly` também pode ser aplicado nos parâmetros do construtor para definir e inicializar as propriedades. No exemplo a seguir o modificar `readonly`, à direita do parâmetro nome no construtor, tem o papel de criar a propriedade nome na classe e atribuir o valor passado como parâmetro.

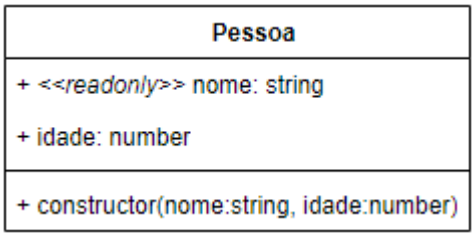
```
class Pessoa {
    idade: number;

    constructor(readonly nome: string, idade: number) {
        // essa atribuição não é incorreta, mas é desnecessária
        // this.nome = nome;
        this.idade = idade;
    }
}

const p = new Pessoa("Ana", 18);
p.idade++;
console.log(p);
```

Observe que o objeto p possui as propriedades nome e idade:

```
PS C:\Desktop\exemplo> npm start
> exemplo@1.0.0 start
> ts-node ./src/index
Pessoa { nome: 'Ana', idade: 19 }
```

O modificador `readonly` é útil quando desejamos garantir que o valor de uma propriedade seja definido apenas uma vez, geralmente durante a inicialização da classe, e não possa ser alterado posteriormente.

O diagrama de classes UML não possui uma representação específica para o modificador `readonly`. No entanto, é possível utilizar o estereótipo `<<readonly>>` à esquerda da propriedade para indicar somente leitura.



III. Modificadores de visibilidade

Os modificadores de visibilidade desempenham papel importante na POO ao controlar o acesso aos membros (propriedades e métodos) de uma classe. Eles ajudam a estabelecer o encapsulamento e a modularidade do código, fornecendo restrições sobre como os membros podem ser acessados e manipulados.

Os modificadores de visibilidade também são chamados de modificadores de acessibilidade, pois na prática eles restringem o acesso aos membros da classe.

No TS, existem três modificadores de visibilidade que podem ser aplicados a membros de uma classe: `public`, `private` e `protected`. Os tipos de visibilidade são usados para restringir o acesso externo aos membros da classe. Esse é o principal recurso usado para tornar os objetos "caixas opacas" e liberar "tomadas de acesso".

1. `public`: é o modificador de visibilidade padrão, ou seja, se nenhum modificador for especificado, o membro é considerado public. Os membros públicos podem ser acessados de qualquer lugar, tanto dentro da classe quanto por meio de instâncias da classe e dentro de subclasses;
2. `private`: o modificador private restringe o acesso aos membros apenas dentro da própria classe onde eles são definidos. Membros privados não podem ser acessados fora da classe ou por meio de instâncias da classe. No exemplo a seguir as propriedades `nome` e `idade` não podem ser acessadas, tanto para leitura quanto para escrita, através do objeto que está na variável `p`:

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

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

    public print():void {
        // certo: os membros privados podem ser acessados de dentro da própria classe
        console.log(this.nome, this.idade);
    }
}

const p = new Pessoa("Ana",18);
// errado: a propriedade nome não pode ser acessada a partir do objeto
p.nome = "Ana Maria"; // não pode para escrita
console.log(p.nome); // não pode para leitura
```

O modificador `private` também restringe o acesso nas subclasses. No exemplo a seguir, o acesso às propriedades `nome` e `idade` são proibidos no método `imprimir` da subclasse `Cliente`.

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

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

    public print(): void {
        console.log(this.nome, this.idade);
    }
}

class Cliente extends Pessoa {
    saldo: number;

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

    imprimir(): void {
        super.print();
        // errado: as propriedades nome e idade não podem ser acessados na subclasse
        console.log(this.nome, this.idade);
        console.log(this.saldo);
    }
}


const c = new Cliente("Ana", 18, 950);
c.imprimir();
// errado: a propriedade nome não pode ser acessada a partir do objeto
console.log(c.nome);
```

3. `protected`: O modificador protected permite que os membros sejam acessados na própria classe, bem como nas subclasses que herdam da classe onde os membros são definidos. Os membros protegidos não podem ser acessados fora da classe ou por meio de instâncias da classe, ou seja, o modificador protected é um meio termo entre private e public. No exemplo a seguir, o acesso às propriedades `nome` e `idade` são permitidos no método `imprimir` da subclasse `Cliente`, mas são proibidos no objeto `c`.

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

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

        public print(): void {
            console.log(this.nome, this.idade);
        }
    }

    class Cliente extends Pessoa {
        saldo: number;

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

        imprimir(): void {
            super.print();
            // certo: as propriedades nome e idade podem ser acessados
            // na subclasse por elas serem protected
            console.log(this.nome, this.idade);
            console.log(this.saldo);
        }
    }


    const c = new Cliente("Ana", 18, 950);
    c.imprimir();
    // errado: a propriedade nome não pode ser acessada a partir do objeto
    // por ela ser protected
    console.log(c.nome);
```

Principais papéis dos modificadores de visibilidade na POO:

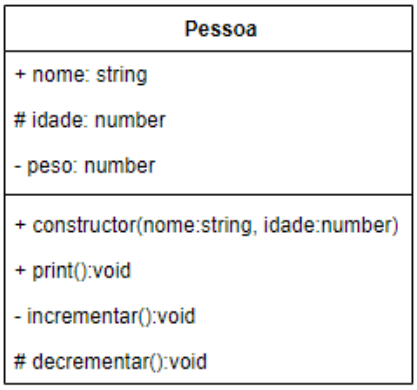
- Encapsulamento: os modificadores de visibilidade permitem ocultar a implementação interna de uma classe, expondo apenas a interface pública. Isso evita que partes externas do código acessem diretamente os detalhes internos da classe, garantindo a coesão e reduzindo o acoplamento entre diferentes partes do sistema;
- Proteção de dados: os modificadores de visibilidade permitem controlar o acesso aos dados de uma classe. Podemos definir membros como privados ou protegidos para restringir o acesso direto aos dados internos da classe. Isso ajuda a prevenir alterações não autorizadas e garante que os dados sejam manipulados de acordo com a lógica e as regras definidas pela classe;
- Abstração: os modificadores de visibilidade auxiliam na criação de uma abstração adequada, fornecendo uma interface clara e consistente para os usuários da classe. Os membros públicos representam a interface pública da classe, definindo as operações que podem ser realizadas com os objetos da classe. Os detalhes internos da

implementação, que não são relevantes para os usuários externos da classe, podem ser mantidos como privados ou protegidos;

- Herança e polimorfismo: os modificadores de visibilidade são usados em conjunto com a herança e o polimorfismo para definir a visibilidade dos membros em classes derivadas. Os membros protegidos permitem que as subclasses acessem e herdem esses membros, mantendo-os ocultos para outras partes do código. Isso permite a extensão e personalização do comportamento da classe base nas subclasses.

No diagrama de classes UML os modificadores public, private e protected são representados, respectivamente, pelos símbolos +, – e # à esquerda do membro da classe. No exemplo a seguir:

- O construtor, a propriedade nome e o método print são públicos;
- O atributo peso e o método incrementar são privados;
- O atributo idade e o método decrementar são protegidos.



IV. Getters e Setters

Os getters e setters são recursos que permitem controlar o acesso e a atribuição de valores às propriedades de uma classe. Eles permitem definir comportamentos personalizados para a leitura (getter) e escrita (setter) de propriedades, em vez de simplesmente obter ou atribuir valores diretamente.

Geralmente, usamos definir a propriedade como privada e fornecemos os acessos `get` e `set` como público. No exemplo a seguir, o método getter retorna o valor atual da propriedade _nome, enquanto o setter recebe um valor como parâmetro e faz a atribuição desse valor à propriedade _nome.

```
class Pessoa {
    private _nome: string;

    constructor(_nome: string) {
        this._nome = _nome;
    }

    get nome(): string {
        return this._nome;
    }

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

const p = new Pessoa("Ana");
// Observe que os getters e setters são acessados como propriedades, ou seja,
// não requerem parênteses, como métodos.
```

```
// o set está sendo invocado para atribuir valor na propriedade _nome
p.nome = "Ana Maria";
// o et está sendo invocado para ler a propriedade _nome
console.log(p.nome);
```

Observação: os acessos `get` e `set` não podem ter o mesmo nome da propriedade.

> Os getters e setters nos permitem adicionar lógica personalizada ao acesso e atribuição de valores a propriedades de uma classe. Isso é útil quando precisamos realizar validações, formatações ou outras manipulações antes de armazenar ou retornar um valor. Além disso, os getters e setters permitem manter um maior controle sobre o acesso aos dados internos da classe e facilitam a manutenção e evolução do código.

V. Propriedades de parâmetro

No TS, as propriedades de parâmetro (parameter properties, em inglês) são uma sintaxe simplificada para definir e inicializar propriedades de uma classe diretamente nos parâmetros do construtor. Essa abordagem economiza tempo e reduz a repetição de código, tornando mais fácil declarar e atribuir valores às propriedades de uma classe em um único local.

No exemplo a seguir as propriedades nome, idade e genero foram definidas e inicializadas nos parâmetros do construtor. Uma vantagem é não ter de inicializar as propriedades no corpo do construtor.

```
class Pessoa {
    //as propriedades são definidas nos parâmetros do construtor
    constructor(public nome:string, private idade:number, readonly genero:string){
        // as propriedades são inicializadas nos parâmetros
    }

    public print():void {
        console.log(this.nome, this.idade, this.genero);
    }
}

const p = new Pessoa("Ana",18,"F");
p.print();
```

As propriedades podem ser criadas e inicializadas usando os modificadores de visibilidade ou `readonly` antes dos parâmetros.

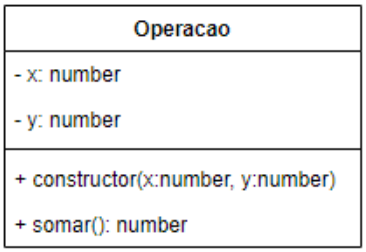
Para mais detalhes acesse https://www.typescriptlang.org/docs/handbook/2/classes.html#parameter-properties.

Exercícios

Veja o vídeo se tiver dúvidas nos exercícios: https://youtu.be/9sjEp_ohPMg

Observação: crie um único projeto para fazer todos os exercícios. Dica, use os passos da Aula 3 para criar o projeto.

Exercício 1: Aplicar na classe Operacao os modificadores de visibilidade/acessibilidade representados no diagrama UML ao lado.




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

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

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

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

Exemplo de saída:

```
PS D:\aula4> npm run um
> aula4@1.0.0 um
> ts-node ./src/exercicio1
Somar: 20
```

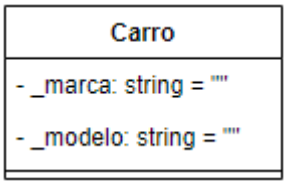
Exercício 2: Alterar o código da classe Operacao, do Exercício 1, para que as propriedades sejam definidas e inicializadas no construtor.

Exemplo de saída:

```
PS D:\aula4> npm run dois
> aula4@1.0.0 dois
> ts-node ./src/exercicio2
Somar: 20
```

Exercício 3: Codificar a classe Carro e adicionar os acessos get e set nas propriedades. Utilize o código a seguir para testar:

```
const carro = new Carro();
carro.marca = "VW";
carro.modelo = "Gol";
console.log(carro);
```



Exemplo de saída:

```
PS D:\aula4> npm run tres
> aula4@1.0.0 tres
> ts-node ./src/exercicio3
Carro { _marca: 'VW', _modelo: 'Gol' }
```

Exercício 4: Alterar o código da classe Carro, do Exercício 3, para que as propriedades sejam definidas e inicializadas no construtor. Retirar os acessos set das propriedades, deixe apenas os acessos get.

Exemplo de saída:

```
PS D:\aula4> npm run quatro
> aula4@1.0.0 quatro
> ts-node ./src/exercicio4
Carro { _marca: 'VW', _modelo: 'Gol' }
```

Exercício 5: Criar um array do tipo Carro – use o tipo de dado definido no Exercício 4 – e adicione 3 objetos do tipo Carro nesse array. Na sequência, imprima o conteúdo do array no terminal, assim como mostrado ao lado.

Exemplo de saída:

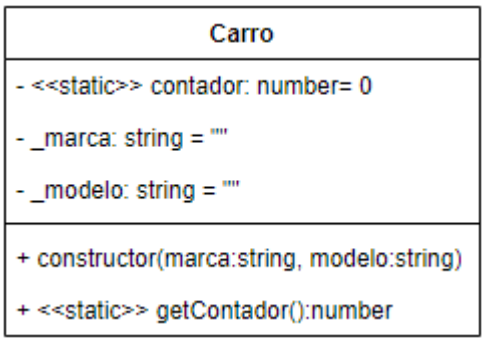
```
PS D:\aula4> npm run cinco
> aula4@1.0.0 cinco
> ts-node ./src/exercicio5
VW Gol
Fiat Uno
GM Corsa
```

Exercício 6: Alterar a classe Carro, do Exercício 4, para ter a propriedade contador e o método getContador, ambos representados no diagrama a seguir.

Incrementar no construtor a propriedade contador a cada nova instância criada. Desta forma, a propriedade contador terá a quantidade de objetos criados.

Completar o código a seguir para exibir no terminal a quantidade de objetos do tipo Carro que já foram criados.

```
const a = new Carro("VW","Gol");
const b = new Carro("Fiat","Uno");
const c = new Carro("GM","Corsa");
```



Exemplo de saída:

```
PS D:\aula4> npm run seis
> aula4@1.0.0 seis
> ts-node ./src/exercicio6
Quantidade: 3
```

Exercício 7: Criar um objeto do tipo Cliente e imprimir no terminal as suas propriedades. Observação: a classe Cliente não poderá ser alterada.

```
class Cliente {
    private constructor(private nome:string, private idade:number){
    }

    public static criar(nome:string, idade:number):Cliente {
        return new Cliente(nome,idade);
    }

    print(){
        console.log(this.nome, this.idade);
    }
```

Exemplo de saída:

```
PS D:\aula4> npm run sete
> aula4@1.0.0 sete
> ts-node ./src/exercicio7
Maria 21
```

```
}
```

Exercício 8: Codificar o corpo da classe Calcular. Requisitos:

- A classe Matematica não poderá ser alterada;
- A soma e subtração deverá ser calculada nos métodos da classe Matematica;
- O código em amarelo não poderá ser alterado.

Exemplo de saída:

```
PS D:\aula4> npm run oito
> aula4@1.0.0 oito
> ts-node ./src/exercicio8
Soma: 8
Diferença: 2
```

```
class Matematica {
    constructor(private a: number, private b: number) {
    }

    protected somar() {
        console.log("Soma:", this.a + this.b);
    }

    protected subtrair() {
        console.log("Diferença:", this.a - this.b);
    }
}

class Calcular extends Matematica {
}

const calc = new Calcular(5,3);
calc.somar();
calc.subtrair();
```

Exercício 9: Adicionar no código a seguir algum mecanismo para contar a quantidade de geometrias criadas, ou seja, contar a quantidade de instâncias dos tipos Retangulo e Circulo. No exemplo ao lado foram criadas duas geometrias.

Exemplo de saída:

```
PS D:\aula4> npm run nove
> aula4@1.0.0 nove
> ts-node ./src/exercicio9
Retângulo: 50
Círculo: 78.53981633974483
Quantidade de geometrias: 2
```

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

class Retangulo extends Geometria {
    constructor(private base:number, private altura:number){
        super();
    }

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

```
    }
}

class Circulo extends Geometria {
    constructor(private raio: number) {
        super();
    }

    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 10: Fazer o código a seguir exibir um número aleatório no terminal.

Requisitos:

- A classe Número não poderá ser alterada;
- O código em amarelo não poderá ser alterado.

Exemplo de saída:

```
PS D:\aula4> npm run dez
> aula4@1.0.0 dez
> ts-node ./src/exercicio10
Valor: 56
```

```
class Numero {
    protected aleatorio():number {
        return Math.floor(Math.random()*100);
    }
}

class Categoria extends Numero {
}

class Teste extends Categoria {
}

const teste = new Teste();
console.log("Valor:", teste.aleatorio() );
```