Objetivo:

I. Classe abstrata;
II. Interface;
III. Type;
IV. Tipos genéricos;
V. Export e import.

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

> Na POO existem três tipos de classes: classes concretas (todos os métodos possuem corpo), interfaces (nenhum método possui corpo) e classes abstratas (alguns métodos da classe não possuem corpo). Somente uma classe concreta pode ser instanciada usando `new`, as demais podem ser estendidas ou implementadas pelas subclasses.

I. Classe abstrata

Uma classe abstrata é definida usando a palavra reservada `abstract` antes da declaração da classe (marcado em amarelo no exemplo a seguir). A classe abstrata pode ter alguns métodos sem implementação (sem corpo – o corpo é definido pelo par de chaves `{}`).

```
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
```

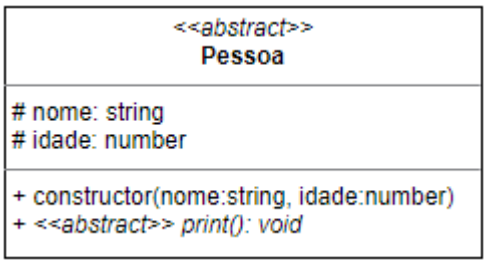
Uma classe abstrata não pode ser instanciada diretamente pelo fato dela poder ter métodos sem corpo. A classe abstrata é projetada para ser estendida e fornecer uma estrutura comum para suas subclasses. No exemplo a seguir a classe `Cliente` estende a classe `Pessoa`, logo ela precisará implementar o método `print`, pois as subclasses são obrigadas a implementar todos os métodos sem implementação da superclasse abstrata.

```
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
```

As classes abstratas fornecem uma maneira de definir uma estrutura comum e comportamentos padrão para suas subclasses, promovendo a reutilização de código e fornecendo uma base sólida para a hierarquia de classes.

No diagrama de classes UML a classe abstrata é representada por itálico e precedido pelo estereótipo *<<abstract>>*. Os métodos abstratos da classe são representados em itálico e, opcionalmente, podem ter o estereótipo *<<abstract>>*.



II. Interface

Uma interface é definida usando a palavra reservada `interface` e todos os métodos não possuem corpo.

A interface precisa ser implementada por uma classe (implementar significa codificar corpo nos métodos sem corpo).

Os membros da interface funcionam como um contrato mínimo a ser seguido pelas classes que implementam a interface. No exemplo a seguir, a classe `Cliente`, obrigatoriamente, precisa ter as propriedades `nome` e `idade` e o método `print` com as assinaturas definidas na 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
```

Como o objeto que está na variável `cli` é do tipo `Cliente`, e o tipo `Cliente` implementa a interface `Pessoa`, então podemos atribuir o objeto que está variável `cli` numa variável `pes`, do tipo `Pessoa`:

```
const pes:Pessoa = cli;
pes.incrementar(); //errado: o tipo Pessoa não possui o método incrementar
pes.print(); //correto: o tipo Pessoa possui o método print
```

No TS, a interface é usada para definir a estrutura de um objeto na notação JSON (JavaScript Object Notation). No exemplo a seguir, a variável `obj` é do tipo `Pessoa`, então, obrigatoriamente, esse objeto terá exatamente as propriedades `nome` e `idade` e o método `print` com as assinaturas definidas na interface.

```
const obj:Pessoa = {
    nome: "Maria",
    idade: 22,
    print: function(){
        console.log(this.nome, this.idade);
    }
};

obj.print();
```

Um objeto que implementa a interface precisa ter exatamente aquilo que foi definido na interface. Veja alguns casos de erro:

Errado: a propriedade `saldo` não existe no tipo `Pessoa`.

```
const obj:Pessoa = {
    nome: "Maria",
    idade: 22,
    saldo: 950,
    print: function(){
```

Errado: a propriedade `idade` foi definida na interface `Pessoa` como sendo do tipo `number`.

```
const obj:Pessoa = {
    nome: "Maria",
    idade: "22",
    print: function(){
        console.log(this.nome, this.idade);
```

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

```
    }
};
```

Errado: a assinatura do método `print` não bate com a assinatura definida na interface `Pessoa`.

Errado: o método `incrementar` não existe na interface `Pessoa`.

```
const obj:Pessoa = {
    nome: "Maria",
    idade: 22,
    print: function(entrada:string){
        console.log(this.nome, this.idade);
    }
};
```

```
const obj:Pessoa = {
    nome: "Maria",
    idade: 22,
    print: function(){
        console.log(this.nome, this.idade);
    },
    incrementar: function() {
        this.idade++;
    }
};
```

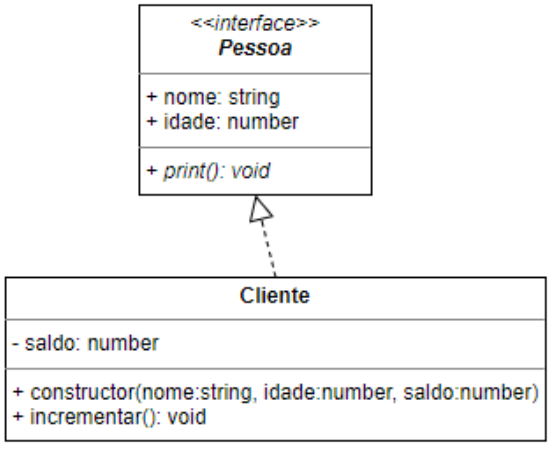
As interfaces não possuem modificadores de visibilidade, pois elas são usadas para definir contratos e estruturas de objetos, não para implementar comportamento. Desta forma, a classe deve fornecer uma implementação pública para todos os membros da interface. Isso significa que os membros da interface serão implicitamente considerados públicos na classe que a implementa.

> Interfaces são ferramentas poderosas na POO para definir contratos e estruturas comuns entre objetos. Elas promovem a interoperabilidade entre diferentes partes do código, nos permitindo especificar requisitos mínimos. Além disso, facilita a verificação de conformidade durante a fase de desenvolvimento e ajuda a garantir uma maior consistência no código.

No diagrama de classes UML a interface é representada por itálico e precedido pelo estereótipo *<<interface>>*. Os métodos são representados em itálico, pois eles são abstratos.

O relacionamento de implementação é representado por uma seta (flecha) vazada e por uma linha pontilhada.

Observação: o relacionamento de herança é representado pela mesma flecha, porém a linha é contínua.



III. Type

No TS, tanto `type` quanto `interface` são usados para definir estruturas de tipos. Em muitos casos, `type` e `interface` podem ser usados de forma intercambiável, e a escolha depende da preferência do programador.

A `interface` `Pessoa` pode ser reescrita usando o `type` `People`:

```
type People = {
    nome: string;
    idade: number;
    print(): void;
};

class Cliente implements People {
    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 obj:People = {
    nome: "Maria",
    idade: 22,
    print: function(){
        console.log(this.nome, this.idade);
    }
};
```

IV. Tipos genéricos

Os tipos genéricos na POO referem-se a uma técnica que permite criar componentes (como classes, funções e interfaces) que podem ser reutilizados com diferentes tipos de dados. Essa abordagem promove a flexibilidade e a reutilização de código, permitindo que os desenvolvedores escrevam implementações genéricas que funcionem com vários tipos específicos.

Os tipos genéricos são definidos usando parâmetros de tipo que serão substituídos por tipos concretos quando a classe, função ou interface for utilizada. O parâmetro de tipo é definido entre os colchetes angulares <>. Geralmente, use-se a letra T, mas pode ser qualquer outra letra ou nome.

No exemplo a seguir a função `concatenar` define um tipo genérico que será substituído por um tipo concreto em cada chamada da função. Na chamada `concatenar(nros,18)` o tipo `T` será substituído por `number` e na chamada `concatenar(nomes,"Carlos")` o tipo `T` será substituído por `string`.

```
function concatenar<T>(vet:T[], item:T){
    vet.push(item);
```

```
}

const nros:number[] = [];
concatenar(nros,18);
concatenar(nros,11);
console.log(nros);

const nomes:string[] = [];
concatenar(nomes,"Carlos");
concatenar(nomes,"Paulo");
console.log(nomes);
```

No exemplo a seguir a classe define um tipo genérico. O tipo genérico será substituído ao instanciar a classe, no objeto criado usando

```
new Lista<string>()
```

o tipo T será substituído por `string` e no objeto

```
new Lista<number>()
```

o tipo T será substituído por `number`.

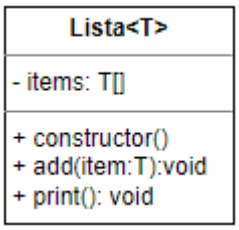
```
class Lista<T> {
    private itens: T[];

    constructor() {
        this.itens = [];
    }

    add(item: T): void {
        this.itens.push(item);
    }

    print(): void {
        for (let i = 0; i < this.itens.length; i++) {
            console.log(this.itens[i]);
        }
    }
}

const nomes = new Lista<string>();
nomes.add("Ana");
nomes.add("Pedro");
nomes.add("Maria");
nomes.print();
const nros = new Lista<number>();
nros.add(12);
nros.add(9);
nros.print();
```

No exemplo a seguir a interface define um tipo genérico que será substituído pelo tipo definido na implementação (marcado em amarelo).

```
interface Ponto<T>{
    x:T;
    y:T;
}

const p:Ponto<number> = {
    x: 1,
    y: 2
};

const q:Ponto<string> = {
    x: "1",
    y: "2"
};
```

Os tipos genéricos são úteis quando precisamos escrever código que é independente de algum tipo de dado específico, permitindo maior reutilização e flexibilidade. Eles nos permitem criar estruturas e algoritmos genéricos que podem ser usados com vários tipos de dados, mantendo a segurança de tipo durante a compilação do TS.

A UML não possui uma notação específica para tipos genéricos, pois a UML é focada na estrutura e nos relacionamentos entre as classes do que nos detalhes da implementação específica da linguagem. Desta forma, a representação exata pode variar dependendo da convenção utilizada ou do propósito do diagrama. No entanto, é comum adicionar um parâmetro de tipo genérico após o nome da classe, indicando que a classe é genérica e pode ser parametrizada com um tipo específico.



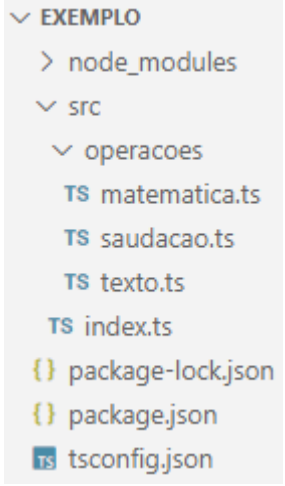
V. Export e import

No contexto do TS um módulo é um arquivo que exporta alguma entidade (variável, função, classe, interface ou tipo). Os pacotes são pastas, na estrutura de arquivos do projeto, que possuem algum módulo que exporta alguma entidade.

A exportação e importação é um recurso importante na modularização e reutilização de código. Dado que o código pode ser organizado em pastas e módulos e consumidos em diferentes partes do programa.

A seguir tem-se três formas de exportar e importar entidades. Considere nas explicações o pacote `operacoes` com os módulos `matematica`, `saudacao` e `texto`.

Estrutura de pastas e arquivos:



Módulo matematica: exemplo de exportação por default

```
export default function somar(x:number, y:number):number {
    return x + y;
}
```

Módulo texto: exemplo de exportação nomeada

```
export function concatenar(x:string, y:string):string {
    return x + y;
}

export const carro = "Uno";
```

Módulo saudacao: exemplo de exportação agrupada

```
function msg(): void {
    console.log("olá");
}

function resposta(): void {
    console.log("Boa noite");
}

export { msg, resposta };
```

Módulo index:

```
import somar from "./operacoes/matematica";
import add from "./operacoes/matematica";
import { concatenar, carro } from "./operacoes/texto";
import { msg, resposta } from "./operacoes/saudacao";

console.log(add(5,11));
console.log(somar(2,3));
console.log(concatenar("o","i"));
console.log(carro);
msg();
resposta();
```

1. Exportação por padrão: um arquivo pode ter apenas uma entidade exportada por padrão. Usa-se o termo `export default` antes da entidade a ser exportada. O módulo `matematica` exporta por default a função `somar`.

   A importação de uma entidade exportada por default pode ter qualquer nome na importação. No exemplo a seguir o mesmo recurso foi chamado de `somar` e `add`:

   ```
   import somar from "./operacoes/matematica";
   import add from "./operacoes/matematica";
   ```

2. Exportação nomeada: o termo export vem antes de cada entidade exportada. O módulo `texto` exporta a função `concatenar` e a variável `carro`.

A importação requer que o recurso importado tenha o nome exato da exportação e seja desestruturado, ou seja, fique entre chaves. Os recursos podem ser importados agrupados:

```
import { concatenar, carro } from "./operacoes/texto";
```

ou podem ser importados separadamente:

```
import { concatenar } from "./operacoes/texto";
import { carro } from "./operacoes/texto";
```

3. Exportação agrupada: as entidades exportadas são estruturadas em um objeto JSON. O módulo `saudacao` exporta as funções `msg` e `resposta`.
   A importação requer que o recurso importado tenha o nome exato da exportação e seja desestruturado, ou seja, fique entre chaves. Os recursos podem ser importados agrupados:

```
import { msg, resposta } from "./operacoes/saudacao";
```

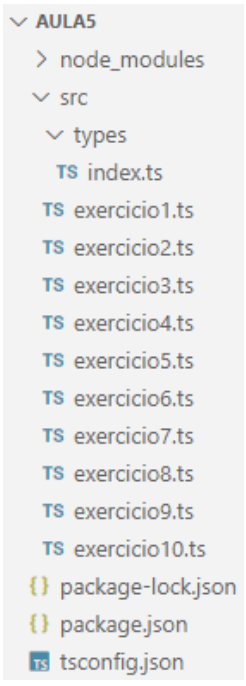
ou podem ser importados separadamente:

```
import { resposta } from "./operacoes/saudacao";
import { msg } from "./operacoes/saudacao";
```

Um módulo pode ter somente um recurso exportado por default (por padrão), mas pode ter vários exportados de forma nomeada ou agrupada.

Exercícios

Observação: crie um único projeto para fazer todos os exercícios, assim como você fez nas aulas anterior. Depois, crie a pasta types e coloque o código a seguir na arquivo index.ts:




```
export type Ponto = {
    x: number;
    y: number;
};

export interface Retangulo {
    infEsquerdo: Ponto;
    supDireito: Ponto;
};

export abstract class Livro {
    constructor(protected titulo:string, protected ano:number){
    }

    abstract print():void;
}

export class Pilha<T> {
    private items: T[] = [];
```

```
    push(item:T):void {
        this.items.push(item);
    }

    pop():T|undefined {
        return this.items.pop();
    }
}

interface Automovel {
    velocidade: number;
    coeficiente: number;
    // retorna a distância percorrida ao frear o automóvel
    // cálculo da distância percorrida ao frear um automóvel: D = V²/250μ
    // https://www.resumoescolar.com.br/fisica/calculando-a-frenagem-de-um-automovel/
    frenagem(): number;
}

export interface Carro extends Automovel {
    consumo: number;
    // retorna a quantidade de combustível gasta para percorrer uma distância
    // gasto = distancia/consumo
    gasto(distancia:number):number;
}

export abstract class Imc {
    constructor(protected peso: number, protected altura: number){
    }

    getImc(){
        return this.peso / this.altura**2;
    }

    abstract classificacao():string;
}

// exportado por padrão
export default interface Geometria {
    area():number;

    perimetro():number;
}
```

Exercício 1: No arquivo exercicio1.ts crie um objeto JSON do tipo Ponto, usando o tipo definido no pacote types. Na sequência, imprima este objeto no terminal.

Requisito: será necessário importar o tipo Ponto no arquivo exercicio1.ts.

Exemplo de saída:

```
PS D:\aula5> npm run um
> aula5@1.0.0 um
> ts-node ./src/exercicio1
Objeto: { x: 10, y: 20 }
```

Exercício 2: No arquivo exercicio2.ts crie um objeto JSON do tipo Retangulo, usando o tipo definido no pacote types. Na sequência, imprima este objeto no terminal.

Requisito: será necessário importar o tipo Retangulo no arquivo exercicio2.ts.

```
PS D:\aula5> npm run dois
> aula5@1.0.0 dois
> ts-node ./src/exercicio2
Objeto: { infEsquerdo: { x: 1, y: 2 }, supDireito: { x: 3, y: 4 } }
```

Exercício 3: Colocar o código a seguir no arquivo exercicio3.ts e definir no pacote types um tipo de dado a ser usado na variável poligono.

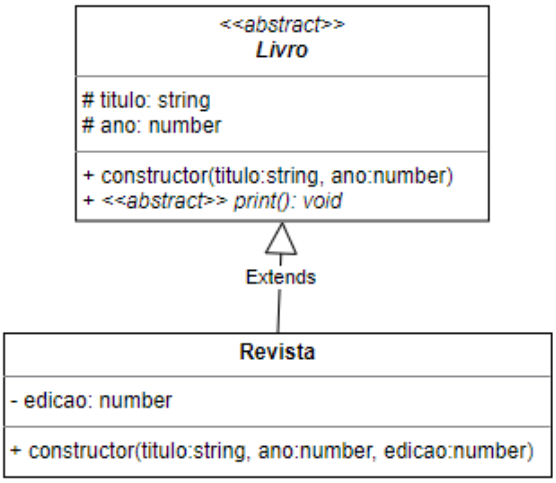
```
const poligono = {
    nome: "propriedade",
    editavel: false,
    pontos: [{x:1, y:2},{x:3, y:4},{x:2,y:5}]
};
```

Observação: nos exercícios anteriores foi dado o tipo e pediu-se para você criar o objeto JSON, aqui é o oposto.

Exercício 4: A classe Livro está codificada no pacote types. Codificar no arquivo exercicio4.ts a classe Revista. Na sequência, crie um objeto do tipo Revista e chame o seu método print.

Exemplo de saída:

```
PS D:\aula5> npm run quatro
> aula5@1.0.0 quatro
> ts-node ./src/exercicio4
Superinteressante: O futuro da IA 2023 448
```



Exercício 5: A classe Pilha está codificada no pacote types. Codificar no arquivo exercicio5.ts um objeto do tipo Pilha e colocar nele os nomes Ana, Pedro, Luiz e Maria (nesta sequência). Após colocar os nomes, utilize uma estrutura de repetição while para remover os nomes e imprimir no terminal.

Exemplo de saída:

```
PS D:\aula5> npm run cinco
> aula5@1.0.0 cinco
> ts-node ./src/exercicio5
Maria
Luiz
Pedro
Ana
```

Exercício 6: A classe Pilha e o tipo Ponto estão codificados no pacote types. Codificar no arquivo exercicio6.ts um objeto do tipo Pilha e colocar nele 4 objetos do tipo Ponto. Após colocar os objetos na pilha, utilize uma estrutura de repetição while para remover os objetos e imprimir no terminal.

Exemplo de saída:

```
PS D:\aula5> npm run seis
> aula5@1.0.0 seis
> ts-node ./src/exercicio6
{ x: 4, y: 5 }
{ x: 3, y: 4 }
{ x: 2, y: 3 }
{ x: 1, y: 2 }
```

Exercício 7: A interface Carro está definida no pacote types. Codificar no arquivo exercicio7.ts uma classe de nome Uno que implementa a interface Carro. Utilize o objeto a seguir para testar o código.

```
const uno = new Uno(12.5, 90, 1);
console.log(`Gasto para percorrer 100km: ${uno.gasto(100)} litros`);
console.log(`Distância percorrida ao frear o carro: ${uno.frenagem()} metros`);
```

Exemplo de saída:

```
PS D:\aula5> npm run sete
> aula5@1.0.0 sete
> ts-node ./src/exercicio7
Gasto para percorrer 100km: 8 litros
Distância percorrida ao frear o carro: 32.4 metros
```

Exercício 8: A classe abstrata Imc está definida no pacote types. Codificar no arquivo exercicio8.ts uma classe de nome Pessoa que estende da classe Imc. Utilize o objeto a seguir para testar o código.

```
const pessoa = new Pessoa(70, 1.62);
console.log("IMC:", pessoa.getImc());
console.log("Classificação:", pessoa.classificacao());
```

Utilize as seguintes classificações de IMC:

1. Abaixo do peso: IMC abaixo de 18,5
2. Peso normal: IMC entre 18,5 e 24,9
3. Sobrepeso: IMC entre 25 e 29,9
4. Obesidade grau I: IMC entre 30 e 34,9
5. Obesidade grau II: IMC entre 35 e 39,9
6. Obesidade grau III (obesidade mórbida): IMC igual ou superior a 40

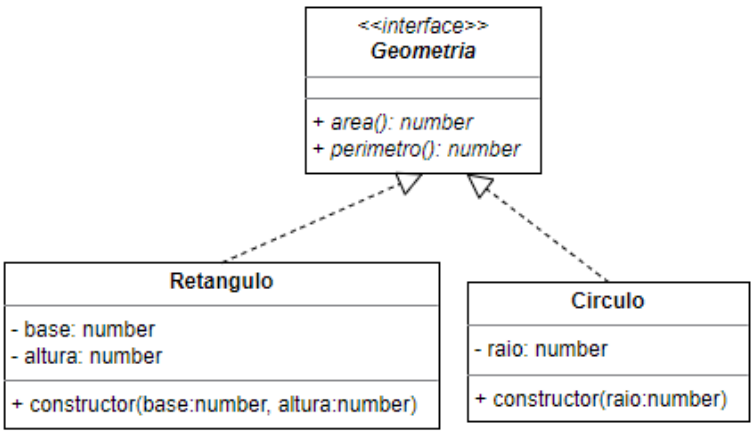
Exemplo de saída:

```
PS D:\aula5> npm run oito
> aula5@1.0.0 oito
> ts-node ./src/exercicio8
IMC: 26.672763298277697
Classificação: Acima do peso (sobrepeso)
```

Exercício 9: A interface Geometria está definida no pacote types – observe que a exportação é por padrão. Codificar no arquivo exercicio9.ts as classes Retangulo e Circulo. Utilize o objeto a seguir para testar o código.

```
let geom: Geometria = new Retangulo(10, 5);
console.log("Área do retângulo:", geom.area());
console.log("Perímetro do retângulo:", geom.perimetro());
geom = new Circulo(5);
console.log("Área do círculo:", geom.area());
```

```
console.log("Perímetro do círculo:", geom.perimetro());
```



Exemplo de saída:

```
PS D:\aula5> npm run nove
> aula5@1.0.0 nove
> ts-node ./src/exercicio9
Área do retângulo: 50
Perímetro do retângulo: 30
Área do círculo: 78.53981633974483
Perímetro do círculo: 31.41592653589793
```

Exercício 10: O código a seguir é usado para ordenar arrays de string ou number. Tornar o código genérico para existir apenas uma função de ordenação. Requisitos:

- As funções stringCompare e numberCompare não podem ser alteradas;
- Criar uma única função para substituir as funções bubleSortString e bubleSortNumber. A quantidade de linhas de código da nova função não poderá ser superior a quantidade de linhas de bubleSortString ou bubleSortNumber.

Exemplo de saída:

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

```
PS D:\aula5> npm run dez
> aula5@1.0.0 dez
> ts-node ./src/exercicio10
[ 1, 2, 3, 6, 8, 9 ]
[ 'abacaxi', 'atemoia', 'banana', 'limão', 'mamão' ]
```

```
function stringCompare(a: string, b: string): boolean {
    return a.localeCompare(b) > 0;
}

function numberCompare(a: number, b: number): boolean {
    return a > b;
}

function bubbleSortString(array: string[]): string[] {
    const n = array.length;
    let trocou = true, temp;

    while( trocou ) {
        trocou = false;
```

```
        for (let i = 0; i < n - 1; i++) {
            if (stringCompare(array[i], array[i + 1])) {
                // troca os itens
                temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
                trocou = true;
            }
        }
    }
    return array;
}

function bubbleSortNumber(array: number[]): number[] {
    const n = array.length;
    let trocou = true, temp;

    while( trocou ) {
        trocou = false;

        for (let i = 0; i < n - 1; i++) {
            if (numberCompare(array[i], array[i + 1])) {
                // troca os itens
                temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
                trocou = true;
            }
        }
    }
    return array;
}

const numeros = [8, 6, 2, 9, 1, 3];
console.log(bubbleSortNumber(numeros));

const palavras = ["banana", "mamão", "abacaxi", "limão", "atemoia"];
console.log(bubbleSortString(palavras));
```