

Objetivo:

- I. Funções;
- II. Arrays: métodos `forEach`, `map` e `reduce`;
- III. JSON (JavaScript Object Notation);
- IV. Estruturação e desestruturação;
- V. Async e `await`;
- VI. Classes;
- VII. Export e import.

Observação: antes de começar, crie um projeto para reproduzir os exemplos:

1. Crie a pasta `exemplo` (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 fundamental de um projeto Node (arquivo `package.json`):

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> npm init -y
```

4. No terminal, execute o comando `npm i -D ts-node typescript` para instalar os pacotes `ts-node` e `typescript` como dependências de desenvolvimento;

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> npm i -D ts-node typescript
```

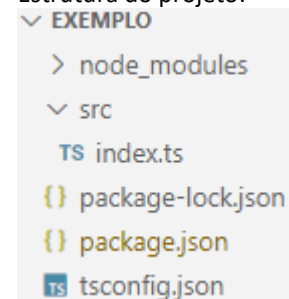
5. No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo `tsconfig.json`):

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> tsc --init
```

6. Crie a pasta `src` na raiz do projeto;
7. Crie o arquivo `index.ts` na pasta `src`;

Estrutura do projeto:



8. Adicione na propriedade `scripts`, do `package.json`, o comando para executar o arquivo `index.ts`. Ao final o arquivo `package.json` terá o seguinte conteúdo:

```
{  
  "name": "exemplo",  
  "version": "1.0.0",  
  "description": "",  
  "scripts": {  
    "start": "ts-node src/index.ts"  
  },  
  "dependencies": {}  
}
```

```
"main": "index.js",
"scripts": {
  "start": "ts-node ./src/index"
},
"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": {
  "start": "npx ts-node ./src/index"
},
```

I. Funções

Nas linguagens JavaScript (JS) e TypeScript (TS) existem três formas de definir uma função. Para mais detalhes acesse <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions>.

1. **Função nomeada:** a forma mais tradicional é usando a palavra-reservada `function` seguida pelo nome da função:

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

Podemos atribuir uma função a uma variável, assim como no exemplo a seguir, porém o nome de função `somar` deixa de existir.

```
const sum = function somar(a: number, b: number): number {
  return a + b;
}

console.log( somar(3,2) ); // errado: somar não está definido
console.log( sum(3,2) ); // correto: a variável sum recebeu a função
```

2. **Função anônima:** é uma função que `não` possui um nome. Nesse caso, usamos uma “expressão de função” para atribuir uma função anônima a uma variável. No exemplo a seguir, o conteúdo da variável `dif` será a função anônima. Desta forma, para chamar a função precisamos usar o nome da variável.

```
const dif = function(a: number, b: number): number {
  return a - b;
}

console.log(dif(2,3));
```

É possível passar o conteúdo de uma variável para outra variável, mas lembre-se que o conteúdo da variável `dif` é apenas a referência (endereço) para a função, então na prática copiou-se apenas o endereço da função para a variável `calc`.

```
const calc = dif;  
console.log(calc(9, 4));
```

Observação: as duas instruções a seguir são diferentes:

```
const x = dif(5,3);  
const y = dif;
```

No 1º caso estamos invocando a função, para ela ser executada, e o resultado será colocado na variável `x`.

No 2º caso estamos apenas lendo o endereço da função na memória sem fazer qualquer execução, ou seja, a variável `y` receberá o endereço da função na memória.

3. **Arrow function (função seta/flecha):** possui uma sintaxe mais curta quando comparada com a função anônima.

Arrow functions são sempre anônimas. A seguir tem-se três declarações distintas de arrow function:

```
const mult = (a: number, b: number): number => {  
  return a * b;  
};  
const div = (a: number, b: number): number => a / b;  
const pow = (a: number, b: number): number => { return a ** b };
```

Quando a arrow function possui no corpo apenas a instrução `return`, então podemos retirar o `return` e as chaves, assim como fizemos na função `div`. Mas se adicionarmos as chaves, como fizemos na função `pow`, então a função precisará ter a instrução `return`.

II. Arrays

Array é um objeto global do JS usado na construção de arrays - objetos de alto nível semelhante a lista, pois eles podem ser redimensionados na linguagem JS.

A forma tradicional de percorrer os elementos de um array é usando a estrutura de repetição `for`:

```
const vet:number[] = [4, 2, 8, 5];  
for (let i = 0; i < vet.length; i++) {  
  console.log(vet[i]);  
}
```

A seguir tem-se as principais operações que faremos com arrays:

1. **Criar cópia do array:** para criar uma cópia do array podemos usar o spread operator `[...array]`:

```
const w = [4, 2, 8, 5];  
const z = [...w]; //cria uma cópia do array w  
w[1] = 20; //altera um elemento do array w sem alterar o array z  
console.log(w); //resultado [ 4, 20, 8, 5 ]  
console.log(z); //resultado [ 4, 2, 8, 5 ]
```

2. **Remover elementos do array:** o método `splice(indice, quant)` é usado para remover `quant` elementos a partir da posição `indice` do array. No exemplo a seguir serão removidos 3 elementos a partir da 3ª posição do array `w`.

```
const w = [4, 2, 8, 5, 1, 9, 7];
w.splice(2,3); //remove os elementos 8, 5 e 1
console.log(w);
```

3. **Método `forEach`:** o objeto Array possui métodos para iterar sobre os elementos do objeto array.

O método `forEach` itera sobre os elementos do array, isto é, ele chama a função call-back para cada elemento do array (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach).

O método `forEach` não modifica o array original. No exemplo a seguir a função call-back será invocada para cada elemento do array `w`.

```
const w = [4, 2, 8, 5];
//O método forEach recebe como argumento uma função anônima
w.forEach(
    //Essa função anônima pode receber até 3 parâmetros que serão
    //fornecidos pelo forEach.
    //Essa função será invocada para cada elemento do array w
    function (item, indice, array) {
        console.log(indice + ':' + item + ' ' + array);
    }
);
//Saída:
0:4 4,2,8,5
1:2 4,2,8,5
2:8 4,2,8,5
3:5 4,2,8,5

w.forEach(
    //aqui foi passada uma arrow function com 2 parâmetros
    (item, indice) => console.log(indice + ':' + item)
);
//Saída:
0:4
1:2
2:8
3:5

w.forEach(
    //aqui foi passada uma arrow function com 1 parâmetro
    item => console.log(item)
);
//Saída:
4
2
8
5
```

4. **Método `map`:** o método `map` invoca a função call-back, passada como parâmetro para cada elemento do array, e devolve um novo array como resultado (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/map).

O método `map` não modifica o array original. No exemplo a seguir o `map` foi usado para retornar um novo array com cada elemento dobrado.

```
const w = [4, 2, 8, 5];
const r = w.map( function(item){
    return item * 2;
});
console.log(r); //resultado [ 8, 4, 16, 10 ]

const q = w.map( Math.sqrt ); //recebe uma função pronta do JavaScript
console.log(q); //resultado [ 2, 1.41, 2.82, 2.23 ]
```

5. **Método reduce:** o método `reduce` executa a função call-back para cada elemento do array e retorna um único valor. O método `reduce` não modifica o array original. Nos exemplos a seguir será calculado o somatório.

```
const w = [4, 2, 8, 5];
const r = w.reduce( function(soma, item){
    return soma + item;
});
console.log(r); //retorna 19

// usando arrow function
const s = w.reduce( (soma, item) => soma + item );
console.log(s); //retorna 19
```

A função call-back é executada em cada elemento do array (exceto no primeiro, se nenhum valor inicial for fornecido). Por este motivo a letra `a` não foi convertida para maiúsculo no exemplo a seguir.

```
const letras = ['a', 'b', 'c', 'd', 'e']
const s = letras.reduce( (soma, item) => soma + item.toUpperCase() );
console.log(s); //retorna aBCDE
```

Ao fornecermos um **valor inicial**, daí todos os elementos do array serão considerados.

```
const letras = ['a', 'b', 'c', 'd', 'e']
const s = letras.reduce( (soma, item) => soma + item.toUpperCase(), "" );
console.log(s); //retorna ABCDE
```

Os métodos `forEach`, `map` e `reduce` são comumente usados para operar sobre arrays, mas observe que eles são usados em situações distintas:

- `forEach` não possui retorno, pois opera sobre cada elemento do array atual;
- `map` retorna um novo array, onde cada elemento do array atual sofre a operação. Ele não altera o array atual;
- `reduce` retorna um único valor. Ele não altera o array atual.

Aqui foram apresentados alguns métodos do objeto Array, para mais detalhes acesse https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array.

III. JSON (JavaScript Object Notation)

É um formato de dados baseado em texto seguindo a sintaxe de objeto JS. É comumente usado para transmitir dados em aplicações Web, do cliente para o servidor e vice-versa. Ele é uma alternativa a linguagem de marcação XML para o intercâmbio de dados na Web.

O JS fornece um objeto `JSON` global que possui métodos disponíveis para conversão de texto para objeto nativo e vice-versa:

- `parse`: recebe como parâmetro uma string JSON e retorna o objeto JS correspondente. Esse processo é chamado de `deserialization`;
- `stringify`: recebe como parâmetro um objeto nativo e retorna uma string JSON. Esse processo é chamado de `serialization`.

A seguir tem-se um exemplo:

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

  print() {
    console.log(this.nome + " " + this.idade);
  }
}

let c = new Cliente("Ana", 21);
//serialization do objeto nativo
let d = JSON.stringify(c); //converte de objeto para string no formato JSON
console.log(typeof c); //objeto
console.log(typeof d); //string
//veja que os métodos não são convertidos em string
console.log(d); // o resultado é {"nome":"Ana","idade":21}
//deserialization
console.log(JSON.parse(d)); //o resultado é { nome: 'Ana', idade: 21 }
```

Observações:

- JSON é puramente um formato de dados — contém apenas propriedades, sem métodos, veja que no exemplo anterior o método `print` não foi serializado;
- JSON requer aspas duplas para serem usadas em torno de strings e nomes de propriedades. Aspas simples não são válidas:

```
const a = '{"nome":"Mara","idade":21}';
console.log(JSON.parse(a)); //correto
const b = '{nome:"Mara","idade":21}';
//console.log(JSON.parse(b)); //errado: a propriedade nome está sem aspas
const c = '{"nome':'Mara','idade':21}";
//errado: as propriedades e o valor string envolvidos por aspas simples
//console.log(JSON.parse(c));
```

IV. Estruturação e desestruturação

A desestruturação (destructuring) é uma característica do JS e TS que permite extrair valores de objetos ou arrays e atribuí-los a variáveis de forma mais concisa.

No exemplo a seguir a atribuição por desestruturação (destructuring assignment) é usada para copiar os valores das propriedades `bairro` e `cidade` do objeto JSON para as variáveis `bairro` e `cidade`. A desestruturação ocorre entre chaves `{}` no lado esquerdo da atribuição.

```
const endereco = {
  logradouro: 'Rua um',
  nro: 123,
  bairro: 'Vila Jardim',
  cep: 12345678,
  cidade: 'Jacareí',
```

```
    uf: 'SP'
  };

  // atribuição por desestruturação: as variáveis recebem os valores das propriedades
  const {bairro,cidade} = endereco;
  console.log(bairro); // o resultado é Vila Jardim
  console.log(cidade); // o resultado é Jacareí
```

Observação: a atribuição por desestruturação só funciona se as variáveis tiverem exatamente os nomes das propriedades do objeto JSON.

De modo oposto, existe a estruturação de objetos JSON. Na estruturação, variáveis são usadas para compor objetos JSON, os nomes das variáveis serão as propriedades do objeto e os valores das variáveis serão os valores das propriedades. No exemplo a seguir, as variáveis nome, idade e peso serão as propriedades do objeto JSON. A estruturação ocorre entre chaves {} no lado direito da atribuição.

```
const nome = "Ana";
const idade = 22;
const peso = 61.5;
// na estruturação as variáveis são colocadas nas chaves do lado direito da atribuição
const pessoa = {nome,idade,peso};
console.log(pessoa); // o resultado é { nome: 'Ana', idade: 22, peso: 61.5 }
```

A desestruturação de arrays é feita usando colchetes [] no lado esquerdo da atribuição. A desestruturação permite extrair elementos de um array e atribuí-los às variáveis individuais. No exemplo a seguir os 3 primeiros elementos do array serão copiados, respectivamente, para as variáveis nome, carro e fruta. Observe que o 4º elemento do array não foi copiado.

```
const textos = ["Ana", "Uno", "Laranja", "Couve"];
// desestruturação para atribuir os elementos do array às variáveis
const [nome, carro, fruta] = textos;
console.log(nome); // o resultado é Ana
console.log(carro); // o resultado é Uno
console.log(fruta); // o resultado é Laranja
```

A estruturação de arrays é usada para criar arrays combinando elementos de outras variáveis. Ela é realizada usando colchetes [] no lado direito da atribuição. No exemplo a seguir os valores das variáveis foram usados para compor os valores do array a ser criado:

```
const base = 2;
const altura = 3;
const profundidade = 4;
// estruturação para criar um array combinando os valores das variáveis base, altura e profundidade
const medidas = [base,altura,profundidade];
console.log(medidas); // o resultado é [ 2, 3, 4 ]
```

A desestruturação de um objeto JSON pode ser feita na atribuição do parâmetro de uma função. No exemplo a seguir, a desestruturação ocorre colocando as chaves {} na atribuição do parâmetro da função **exibir**:

```
// retira do objeto JSON apenas a propriedade carro
function exibir({carro}:any){
```

```
        console.log(carro);
    }

    const obj = {
        pessoa: {
            nome: "Ana",
            idade: 22
        },
        carro: {
            marca: "Fiat",
            modelo: "Uno"
        }
    };

    // chama a função passando o JSON que está na variável obj
    exibir(obj);
```

Podemos desestruturar um objeto JSON aninhado. No exemplo a seguir as chaves verdes desestrutura o objeto para obter a propriedade veículo e as chaves amarelas desestrutura o objeto veículo para obter a propriedade tipo:

```
const pessoa = {
    nome: "Ana",
    veiculo: {
        tipo: {
            marca: "GM",
            modelo: "Corsa"
        },
        ano: 2010
    }
};

const {veiculo:{tipo}} = pessoa;
console.log(tipo);
```

V. Async e await

Async function (função assíncrona) retorna por padrão um objeto Promise (compromisso). No exemplo a seguir a chamada `multiplicar(4)` retornará uma promise.

Para pegar o resultado/falha de uma promise temos de usar os métodos `then` ou `catch` atachadas ao objeto. Somente um dos métodos `then` (no caso de sucesso) ou `catch` (no caso de falha) podem ser executadas por chamada da promise. No exemplo a seguir será chamado o método `then` para a chamada do objeto que está na variável `primeiro` e será chamado o método `catch` para a chamada do objeto que está na variável `segundo`.

```
async function multiplicar(nro?: number) {
    if (nro !== undefined) {
        return nro * 2;
    }
    throw new Error("Não definido");
}
```



```
const primeiro = multiplicar(4);
//o método then será invocado pelo resolve da promise
primeiro
  .then((result) => console.log("Then:", result))
  .catch((result) => console.log("Catch:", result.message))
  .finally(() => console.log("Finally do primeiro"));
console.log("Após a promise:", primeiro); //resultado é Promise { 8 }

const segundo = multiplicar();
//o método catch será invocado pelo reject da promise
segundo
  .then((result) => console.log("Then: " + result))
  .catch((result) => console.log("Catch: ", result.message))
  .finally(() => console.log("Finally do segundo"));
```

Resultado do código: observe a ordem de execução das instruções.

```
Após a promise: Promise { 8 } ← console.log("Após a promise:", primeiro)
Then: 8 ← console.log("Then:", result)
Catch: Não definido ← console.log("Catch: ", result.message)
Finally do primeiro ← console.log("Finally do primeiro")
Finally do segundo ← console.log("Finally do segundo")
```

O método `finally` é executado independentemente do fluxo passar pelos métodos `then` ou `catch`.

No JS, promises são usadas para lidar com operações **assíncronas** e oferecem uma maneira mais fácil de gerenciar fluxos de controle e manipular resultados assíncronos. As promises representam um valor que pode estar disponível agora, no futuro ou nunca.

Essencialmente, uma promise é um objeto retornado para o qual adicionamos call-backs, em vez de passar call-backs para a função. No exemplo anterior, **then**, **catch** e **finally** são métodos call-back adicionadas a chamada das promises.

O operador **await** (aguardar) pode ser usado somente no corpo de funções assíncronas para pausar a execução até que uma **Promise** seja resolvida. O operador **await** nos permite escrever código assíncrono de maneira síncrona, facilitando o tratamento de Promises.

No exemplo a seguir observe que a 2ª chamada da função **exec** deveria terminar antes da 1ª, porém a instrução **await** fará com que a 2ª chamada só ocorrerá após a 1ª ser concluída.

```
const a = await exec(10, 300); //espera 300 milissegundos
const b = await exec(20, 100); //só começará após terminar o anterior
```

No exemplo a seguir o resultado **30** só estará disponível após as duas chamadas da função **exec** serem concluídas.

A função **calc** retorna uma Promise, por esse motivo precisamos fazer **x.then()**.

```
function exec(val:number, tempo:number):Promise<number> {
  return new Promise(resolve => {
    setTimeout(
      () => {
        console.log('terminou: ' + val);
        resolve(val) // chamará o método then
      },
      tempo
    )
  });
}

//função async retorna uma Promise
async function calc() {
  const a = await exec(10, 300); //espera 300 milissegundos
  //só irá começar após terminar a anterior
  const b = await exec(20, 100);
  return a + b;
}

const x = calc();
//o then será a função resolve da Promise calc
x.then(result => console.log("Then:", result) );

//resultado é Promise { <pending> },
//pois a promise está no estado pendente
console.log("Última:",x);
```

Resultado do código ao lado:

```
Última: Promise { <pending> }
terminou: 10
terminou: 20
Then: 30
```

Para mais detalhes https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function e <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/await>.

VI. Classes

As classes são usadas para criarmos **objetos instanciados**. JSON são **objetos literais**, isto é, escrevemos o conteúdo do objeto conforme o criamos. Os objetos literais são usados para representar apenas 1 objeto, já as classes funcionam como templates para a criação de vários objetos.

Uma classe é formada basicamente por propriedades (variáveis), construtor e métodos (funções).

Geralmente, usamos o construtor para inicializar as propriedades. No exemplo a seguir o construtor recebe uma string e um number como parâmetro e esses valores são usados para inicializar, respectivamente, as propriedades nome e idade.

O termo **this** refere-se à instância atual da classe. Ele é uma palavra-reservada que permite acessar as propriedades e métodos da instância da classe dentro dos métodos e construtor da classe. No exemplo a seguir, a instrução **this.nome** acessará a propriedade nome do objeto que está sendo criado.

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

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

  imprimir(){
    console.log(`${this.nome} possui ${this.idade} anos`);
  }
}

const x = new Pessoa("Ana",18);
const y = new Pessoa("Pedro",20);
x.imprimir();
y.imprimir();
```

```
PS C:\Desktop\exemplo> npm start
> exemplo@1.0.0 start
> ts-node ./src/index
Ana possui 18 anos
Pedro possui 20 anos
```

Para criar uma instância (objeto) da classe, usamos a palavra-chave **new** seguida pelo nome da classe. Como exemplo, **new Pessoa("Ana",18)** invocará o construtor da classe, e este por sua vez criará uma instância da classe e retornará essa instância para colocarmos na variável **x**.

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 (objetos) individuais da classe. Isso significa que eles podem ser acessados diretamente na classe, sem a necessidade de criar uma instância da classe.

No exemplo a seguir a propriedade **pi** e o método **somar** são acessados usando o nome da classe **Calcular**. A vantagem dos membros estáticos é não ter de construir um objeto para usar os membros.

```
class Calcular {
  static pi:number = 3.14159;

  static somar(a:number, b:number):number {
    return a + b;
  }
}

console.log("PI:", Calcular.pi); //o resultado é 3.14159
console.log("Soma:", Calcular.somar(2,3)); //o resultado é 5
```

VII. 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:

```

EXEMPLO
├── node_modules
├── src
│   ├── operacoes
│   │   ├── TS matematica.ts
│   │   ├── TS saudacao.ts
│   │   ├── TS texto.ts
│   │   └── TS index.ts
│   ├── package-lock.json
│   └── package.json
└── tsconfig.json
    
```

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

Instruções para criar o projeto e fazer os exercícios:

1. Crie um projeto assim como você fez com o projeto de exemplo no início desta aula;
2. 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;
3. 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": "aula1",
  "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": "^5.1.3"
  }
}
```



```
}  
}
```

Exercício 1 – Na linguagem TS as funções podem ser escritas na notação de Arrow Function. Reescrever a função formatar usando a notação de Arrow Function inline, ou seja, não usar **chaves** e o termo **return**.

```
function formatar(entrada: string): string {  
    return `

${entrada}</div>`;  
}  
  
console.log(formatar("Ana"));  
console.log(formatar("12"));


```

Exemplo de saída:

```
PS D:\aula1> npm run um  
> aula1@1.0.0 um  
> ts-node ./src/exercicio1  
<div>Ana</div>  
<div>12</div>
```

Exercício 2 – Reescrever o código a seguir substituindo a estrutura de repetição for pelo método forEach do objeto Array. Forneça uma arrow function como parâmetro para o método forEach.

Requisitos:

- Exportar por default a função formatar no arquivo exercicio1.ts;
- Comentar as instruções de console no arquivo exercicio1.ts.

```
import formatar from "./exercicio1";  
  
const nomes = ["Ana", "Pedro", "Lucas", "Maria"];  
for(let i = 0; i < nomes.length; i++){  
    console.log(formatar(nomes[i]));  
}
```

Exemplo de saída:

```
PS D:\aula1> npm run dois  
> aula1@1.0.0 dois  
> ts-node ./src/exercicio2  
<div>Ana</div>  
<div>Pedro</div>  
<div>Lucas</div>  
<div>Maria</div>
```

Exercício 3 – Copiar os elementos dos arrays carros e motos para a variável veiculos utilizando spread operator. Na sequência, utilize o método forEach e a função formatar para exibir os elementos do array veiculos, assim como é mostrado ao lado.

```
import formatar from "./exercicio1";  
  
const carros = ["Gol", "Corsa", "Uno", "Fiesta"];  
const motos = ["CG", "XRE", "Biz"];  
const veiculos =
```

Exemplo de saída:

```
PS D:\aula1> npm run tres  
> aula1@1.0.0 tres  
> ts-node ./src/exercicio3  
<div>Gol</div>  
<div>Corsa</div>  
<div>Uno</div>  
<div>Fiesta</div>  
<div>CG</div>  
<div>XRE</div>  
<div>Biz</div>
```

Exercício 4 – Reescrever o corpo da função listar usando o método reduce do objeto Array.

Observação: as sequências de escape `\n` e `\t` são usadas em strings, respectivamente, para quebrar linha e tabulação. Eles foram colocados aqui

Exemplo de saída:

somente para melhorar a apresentação visual no terminal, mas eles não são usados na linguagem HTML.

As sequências de escape são usadas, nas linguagens de programação, para facilitar a formatação de texto, criar novas linhas, adicionar tabulações e inserir outros caracteres especiais em strings.

```
function item(entrada:string):string {
    return ` \t<li>${entrada}</li>\n`;
}

function listar(elementos:string[]):string {
    let soma = "";
    for( let i = 0; i < elementos.length; i++ ){
        soma += item(elementos[i]);
    }
    return `<ul>\n${soma}</ul>\n`;
}

const frutas = ["Manga", "Laranja", "Maça", "Uva"];
const resultado = listar(frutas);
console.log(resultado);
```

Exercício 5 – A função `listar`, do arquivo `exercicio4.ts`, recebe um array de strings. Reescrever o código a seguir para converter o conteúdo da variável `legumes` para array de strings para poder enviar para a função `listar`.

Requisitos:

- Poderá ser modificado apenas o código sinalizado em amarelo. Dica: use o método `split` do objeto `string`;
- Exportar de forma nomeada a função `listar` no arquivo `exercicio4.ts`;
- Importar a função `listar` no arquivo `exercicio5.ts`;
- Comentar as instruções de console no arquivo `exercicio4.ts`.

```
const legumes = "Beterraba,Cenoura,Tomate,Repolho";
const resultado = listar(legumes);
console.log(resultado);
```

Exercício 6 – Uma função pode ser criada e chamada anonimamente. O código a seguir cria uma função e chama ela. Esse recurso só é útil quando a função será chamada somente 1 vez. Nesse caso, a função precisa ser envolvida pelos parênteses verdes e os parênteses amarelos são usados para chamar a função delimitada pelos parênteses verdes.

```
PS D:\aula1> npm run quatro
> aula1@1.0.0 quatro
> ts-node ./src/exercicio4
<ul>
    <li>Manga</li>
    <li>Laranja</li>
    <li>Maça</li>
    <li>Uva</li>
</ul>
```

Exemplo de saída:

```
PS D:\aula1> npm run cinco
> aula1@1.0.0 cinco
> ts-node ./src/exercicio5
<ul>
    <li>Beterraba</li>
    <li>Cenoura</li>
    <li>Tomate</li>
    <li>Repolho</li>
</ul>
```

Exemplo de saída:

```
PS D:\aula1> npm run seis
> aula1@1.0.0 seis
> ts-node ./src/exercicio6
5
```

```
[ function () {  
    console.log('Bom dia');  
} ]();
```

Usando o recurso de criar e chamar uma função anonimamente. Codificar uma função que recebe os números 2 e 3, e imprime no terminal a soma deles.

Exercício 7 – O webservice da ViaCEP (<http://viacep.com.br>) é usado para consultar CEPs. A URL <https://viacep.com.br/ws/12243750/json> retornará um JSON com o endereço do Parque Vicentina Aranha.

O pacote axios (<https://www.npmjs.com/package/axios>) é usado para processar requisições HTTP. Será necessário digitar o comando a seguir no terminal para instalar o pacote axios no seu projeto:

```
TERMINAL  PROBLEMS  OUTPUT  
  
PS D:\aula1> npm i axios
```

Após instalar o pacote axios você pode testar o código a seguir. O método `get` retorna uma promise, por este motivo usamos o método `then` (da promise) para pegar o resultado da requisição HTTP. O objeto JSON retornado pelo axios será colocado na variável `res`.

```
import axios from "axios";  
  
const cep = "12243750";  
const url = `https://viacep.com.br/ws/${cep}/json`;  
  
axios.get(url)  
  .then( res => console.log(res) )  
  .catch( e => console.log(e.message) );
```

Observe que o objeto que está na variável `res` possui as propriedades: `status`, `statusText`, `headers`, `config`, `request` e `data`. Desestruturar a variável `res` para extrair do objeto JSON apenas a propriedade `data`.

Exemplo de saída:

```
PS D:\aula1> npm run sete  
> aula1@1.0.0 sete  
> ts-node ./src/exercicio7  
{  
  cep: '12243-750',  
  logradouro: 'Rua Engenheiro Prudente Meireles de Moraes',  
  complemento: '',  
  bairro: 'Vila Adyana',  
  localidade: 'São José dos Campos',  
  uf: 'SP',  
  ibge: '3549904',  
  gia: '6452',  
  ddd: '12',  
  siafi: '7099'  
}
```


Exercício 8 – O método `regioes` da classe `Ibge` faz uma requisição no webservice do IBGE para obter as regiões do país. Fazer a chamada do método `regioes` para listar as regiões do país. Lembre-se que um método assíncrono retorna uma `Promise`. Para mais detalhes sobre o webservice de localidades do IBGE acesse <https://servicodados.ibge.gov.br/api/docs/localidades>.

```
import axios from "axios";

interface Regiao {
  id: number;
  sigla: string;
  nome: string;
}

class Ibge {
  static async regioes(): Promise<Regiao[]>{
    const url = "https://servicodados.ibge.gov.br/api/v1/localidades/regioes";
    try {
      // o operador await faz esperar a requisição HTTP
      // usou-se o operador de desestruturação para obter apenas a propriedade data do JSON
      const {data} = await axios.get(url);
      return data;
    }
    catch (erro:any) {
      console.log(erro.message);
      return []; // o array será vazio em caso de falha
    }
  }
}
```

Exemplo de saída:

```
PS D:\aula1> npm run oito
> aula1@1.0.0 oito
> ts-node ./src/exercicio8

[
  { id: 1, sigla: 'N', nome: 'Norte' },
  { id: 2, sigla: 'NE', nome: 'Nordeste' },
  { id: 3, sigla: 'SE', nome: 'Sudeste' },
  { id: 4, sigla: 'S', nome: 'Sul' },
  { id: 5, sigla: 'CO', nome: 'Centro-Oeste' }
]
```

Exercício 9 – A URL <http://servicodados.ibge.gov.br/api/v1/localidades/regioes/3/estados> retorna os estados da região sudeste, o `id` da região precisa ser passado na URL. Adicionar na classe `Ibge`, do arquivo `exercicio8.ts`, um método que recebe o `id` da região e retorna os estados dessa região.

Exemplo de saída:

```
PS D:\aula1> npm run nove
> aula1@1.0.0 nove
> ts-node ./src/exercicio9
[
  {
    id: 41,
    sigla: 'PR',
    nome: 'Paraná',
    regioao: { id: 4, sigla: 'S', nome: 'Sul' }
  },
  {
    id: 42,
    sigla: 'SC',
    nome: 'Santa Catarina',
    regioao: { id: 4, sigla: 'S', nome: 'Sul' }
  },
  {
    id: 43,
    sigla: 'RS',
    nome: 'Rio Grande do Sul',
    regioao: { id: 4, sigla: 'S', nome: 'Sul' }
  }
]
```

Exercício 10 – Codificar uma requisição na API <http://servicodados.ibge.gov.br/api/v1/localidades/estados> e exibir os nomes dos estados do país no terminal.

Parte do resultado:

```
PS D:\aula1> npm run dez
> aula1@1.0.0 dez
> ts-node ./src/exercicio10
Rondônia - RO
Acre - AC
Amazonas - AM
Roraima - RR
Pará - PA
Amapá - AP
Tocantins - TO
Maranhão - MA
```