Desafio Aula anterior resolução

Desafio 01 - Criar um objeto calculadora, baseado numa classe Calculadora, que deve possuir:

- um atributo `resultado`
- um construtor
- 4 métodos de operações: `somar()`, `subtrair()`, `multiplicar()` e `dividir()`
- um método para exibição: `mostrarResultado()`

O objeto deve ser inicializado com um valor definido e deve executar todos os módulo, nesta ordem: `somar()`, `subtrair()`, `multiplicar()`, `dividir()`, `mostrarResultado()`.

Exemplo:

```
// inicializado com 3
// 3 somado(5) = 8
// 8 subtraído(2) = 6
// 6 multiplicado por(3) = 18
// 18 dividido por(2) = 9
// exibe 9
```

Desafio Aula anterior resolução

1ª passo // inicializado com 3

```
class Calculadora {

}
```

Esse código deve ser criado dentro da classe Calculadora, o constructor vai criar o objeto na memoria e vai receber um parâmetro(presultado), nesse caso será o 3 no momento que o objeto for criado conforme pede o enunciado do desafio.

```
    resultado: number;   // atributo
    constructor(presultado: number) {  // métodos construtor
    this.resultado = presultado
  }
```

Desafio Aula anterior resolução

2ª passo // 3 somado(5) = 8

Esse código deve ser criado dentro da classe Calculadora

```
somar(valor: number) {
   this.resultado += valor
}
```

3ª passo // 8 subtraído(2) = 6

Esse código deve ser criado dentro da classe Calculadora

```
subtrair(valor: number) {
   this.resultado -= valor
}
```

4ª passo // 6 multiplicado por(3) = 18

Esse código deve ser criado dentro da classe Calculadora

```
multiplicar(valor: number) {
   this.resultado *= valor
}
```

5ª passo // 18 dividido por(2) = 9

Esse código deve ser criado dentro da classe Calculadora

```
dividir(valor: number) {
   this.resultado /= valor
}
```

6ª passo // exibe 9

Esse código deve ser criado dentro da classe Calculadora

```
mostrarResultado(){
   return this.resultado
}
```

```
let calc = new Calculadora(3)  //inicializada com 3

calc.somar(5)          // 3 somado 5 = 8
calc.subtrair(2)       // 8 subtraido 2 = 6
calc.multiplicar(3)    // 6 multiplicado por 3 = 18
calc.dividir(2)        // 18 dividido por 2 = 9
console.log( calc.mostrarResultado() ) // exibe 9
```

Typescript - Interface

Uma interface define a estrutura das classes que a implementam, diferente da herança tradicional uma classe do tipo interface obriga que todos os métodos e atributos devem ser implementados de alguma forma, é como se fosse um contrato entre a interface e a classe que a está associada a ela.

```typescript
interface IAnimal {
  // Atributos
  nome: string;
  idade: number;
  estaVivo: boolean;

  //Métodos
  nascer(): void;
  crescer(): void;
  morrer(): void;

}
```

```typescript
class Animal implements IAnimal {

  nome: string = "";
  idade: number = 0;
  estaVivo: boolean = true;

  nascer() {
    alert("O animal nasceu!!")
  }
  crescer() {
    alert("O animal cresceu!!")
  }
  morrer() {
    alert("O animal morreu!!")
  }
}
```

Typescript - Acessors (get/set)

TypeScript suporta getters/setters que são usados para acesso a atributos protegidos da classe.
Criando um atributo como **private,** você deve ter um método **get** e um método **set** para alterar os valores desse atributo. Como no exemplo a baixo:

```
class Pessoa {
    private _name: string = "";
    get name(): string {
        return this._name;
    }
    set name(p : string) {
        this._name = p;
    }
}
var p = new Pessoa(); //Instanciando classe
p.name = “Flavio Mota"; //set
console.log(p.name); //get
```

http://clubedosgeeks.com.br/programacao/javascript/typescript-orientado-objetos

Typescript - Herança

Para se herdar métodos e atributos de uma classe para outra, como você pode imaginar, basta usar **extends**. Se um método da classe pai for sobrescrito na classe filha, você pode chamar o método da classe pai usando **super**, como vemos a seguir:

```
//Classe pai
class Animal {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

```
//Classe filha
class Snake extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 5) {
    super.move(distanceInMeters);
  }
}

let x: Snake = new Snake("Jurema");
x.move();
```

Desafio - 01

Modele uma classe que represente um caixa que implemente dois métodos: sacar e depositar, esse modelo deve ser construído com um saldo inicial de R$ 1000 e atualizado de acordo com o método chamado.

Exemplo:

Saldo = 1000
Depositar(500)
Saldo = 1500
Sacar(250)
Saldo = 1250

RECODE pro

MÓDULOS – IMPORT E EXPORT

TS

Uma das estratégias consagradas em desenvolvimento de software para lidar com complexidade de código é a da modularização. Em termos simples, modularizar significa dividir o código em partes que representam uma abstração funcional e reaproveitável da aplicação.

Com o ES6, tudo o que temos no JavaScript agora é interpretado como um módulo. Podemos modularizar desde uma variável, uma função, até mesmo uma classe inteira. Cada módulo é armazenado em um arquivo JavaScript.
Para que os módulos sejam compartilhados entre os diversos arquivos do código, precisamos utilizar as palavras reservadas **import** e **export** para importá-los e exportá-los,respectivamente.

RECODE pro

MÓDULOS – IMPORT E EXPORT

TS

Nesse exemplo, vamos usar dois arquivos typescript: code1.ts, code2.ts, onde o arquivo code1.ts vai expor classes para que o code2.ts possa importar essas classes e usa-las como se fosse estruturas locais.

Code1.ts

```ts
export interface controle{
    ligar():void;
    desligar():void;
}

export class animal{
    nome:string="";
    especie:string="";
}

export class pessoa{
    nome:string="";
    idade:number=0;
    falar():void{
        alert("Olaaaa!!")
    };
}
```

Code2.ts

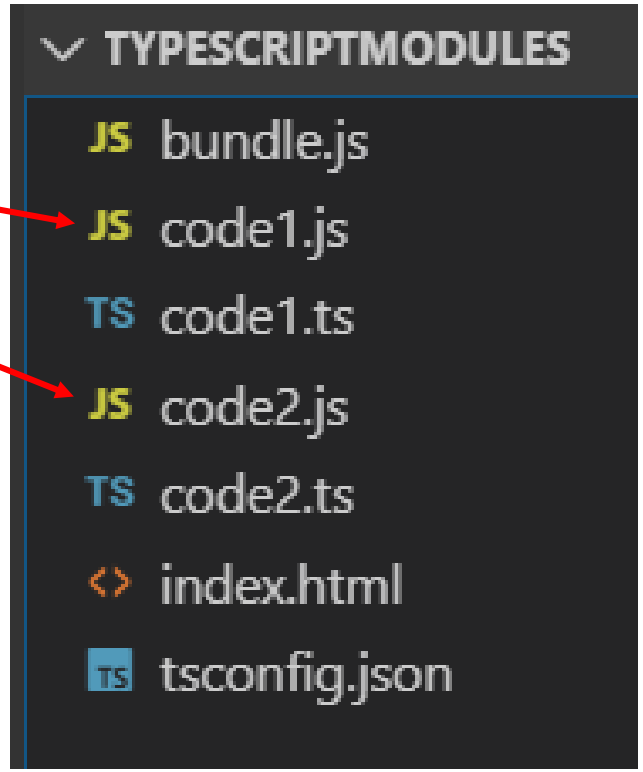
```ts
import { controle } from "./code1";
import { pessoa } from "./code1";

class controleTV implements controle{
    ligar(){
        alert("TV ligou")
    }
    desligar(){
        alert("TV Desligou!")
    }
}

let pessoa1:pessoa = new pessoa();
pessoa1.falar();

let tv1: controleTV = new controleTV();
tv1.ligar();
```

MÓDULOS – IMPORT E EXPORT

Após a criação dos arquivos typescript para que os navegadores possam interpretar os códigos os mesmos devem ser transpilados(convertidos) em Javascript, afinal de contas os navegadores só reconhecem Javascript. Usamos o comando **tsc** e o nome dos arquivos que serão transpilados conforme exemplo abaixo.

```
tsc code1 code2
```

Esse comando gera os arquivos Javascript do projeto.



RECODE pro

MÓDULOS – IMPORT E EXPORT

TS

Arquivos Javascript gerados

Code1.js

```
"use strict";
exports.__esModule = true;
var animal = /** @class */ (function () {
    function animal() {
        this.nome = "";
        this.especie = "";
    }
    return animal;
}());
exports.animal = animal;
var pessoa = /** @class */ (function () {
    function pessoa() {
        this.nome = "";
        this.idade = 0;
    }
    pessoa.prototype.falar = function () {
        alert("Olaaaa!!");
    };
    ;
    return pessoa;
}());
exports.pessoa = pessoa;
```

Code2.js

```
"use strict";
exports.__esModule = true;
var code1_1 = require("./code1");
var controleTV = /** @class */ (function () {
    function controleTV() {
    }
    controleTV.prototype.ligar = function () {
        alert("TV ligou");
    };
    controleTV.prototype.desligar = function () {
        alert("TV Desligou!");
    };
    return controleTV;
}());
var pessoa1 = new code1_1.pessoa();
pessoa1.falar();
var tv1 = new controleTV();
tv1.ligar();
```

Atenção especial para o método **require()**

Ao usarmos o require, Node.js se encarrega de importar o módulo para dentro do arquivo. Feito isso, já é possível utilizar os métodos disponíveis no módulo

MÓDULOS – IMPORT E EXPORT

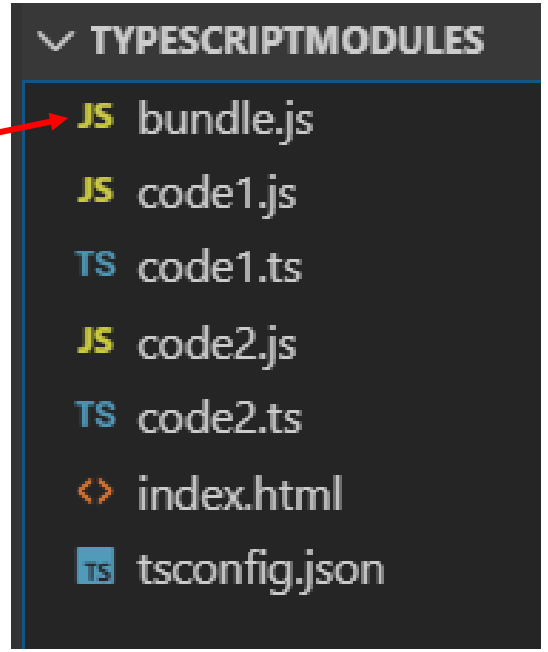
Com o ES6 varias novas funcionalidades foram criadas pra o Javascript e algumas delas os navegadores ainda não conseguem interpretar, por isso os códigos typescript devem ser transpilados para o Javascript tradicional. Quando usamos muitas dependências de vários módulos precisamos usar ferramentas que nos auxiliam com nossos módulos, uma dessas ferramentas é o **Browserify** responsável por fazer o empacotamento dos módulos em um só arquivo no formato js.

Instalação da ferramenta

```
npm install -g browserify
```

Gerando o arquivo principal dos módulos da aplicação

```
browserify code2.js > bundle.js
```



Chamada no arquivo HTML

```
<script src='bundle.js'></script>
```

RECODE pro

Desafio - 02

TS

Faça o mesmo desafio anterior usando os conceitos de módulos, para cada função crie um módulo e centralize em um único arquivo usando o **Browserify.**

Desafio 01 - Criar um objeto calculadora, baseado numa classe Calculadora, que deve possuir:

- um atributo `resultado`
- um construtor
- 4 métodos de operações: `somar()`, `subtrair()`, `multiplicar()` e `dividir()`
- um método para exibição: `mostrarResultado()`

O objeto deve ser inicializado com um valor definido e deve executar todos os módulo, nesta ordem: `somar()`, `subtrair()`, `multiplicar()`, `dividir()`, `mostrarResultado()`.

Exemplo:

```
// inicializado com 3
// 3 somado(5) = 8
// 8 subtraído(2) = 6
// 6 multiplicado por(3) = 18
// 18 dividido por(2) = 9
// exibe 9
```