BOSCH
Invented for life

Aula 2 - Funções

Docupedia Export

Author: Siqueira Joao (CtP/ETS)
Date: 23-Dec-2022 12:23

Table of Contents

1 Declaração de funções

O **Function Hoisting** faz com que as funções declaradas de maneira clássica sempre são jogadas para o topo do arquivo

```
function_hoisting();









function function_hoisting(){
    console.log("Posso ser declarada depois de ser usada!");
};
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\tempCodeRunnerFile.js"
Posso ser declarada depois de ser usada!

[Done] exited with code=0 in 0.597 seconds
```

First-class Objects faz com que as funções possam ser tratadas como dado

```
const FuncaoComoDado = function() {
    console.log("Sou um dado");
}; // Colocamos uma função anônima dentro da constante

FuncaoComoDado(); // Dessa forma a constante se torna uma função
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Sou um dado

[Done] exited with code=0 in 0.58 seconds
```

Arrow function é uma outra forma de se declarar funções

```
const FuncaoArrow = () => console.log("Outra forma de declaração");

const FuncaoArrow2 = () => {
    console.log("Também pode ser assim");
};

FuncaoArrow();
FuncaoArrow2();
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Outra forma de declaração
Também pode ser assim

[Done] exited with code=0 in 0.651 seconds
```

Função dentro de um objeto

```
1  const objeto = {
2      metodoDeObjeto(){
3          console.log("Sou um metodo dentro do objeto");
4      }
5  };
6
7  objeto.metodoDeObjeto();
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Sou um metodo dentro do objeto

[Done] exited with code=0 in 0.828 seconds
```

2 Callback

Funções de callback servem para que algumas funções ocorram na ordem correta, um exemplo de como isso pode ser importante é quando precisamos puxar dados do Banco de Dados, como o tempo para fazer essa tarefa pode variar, isso pode fazer com que as funções fiquem fora de ordem

```
function funcao1(callback){
  setTimeout(function() {
    console.log('Função 1');
    if (callback) callback();
  }, 2500); // Delay
};

function funcao2(callback){
  setTimeout(function() {
    console.log('Função 2');
    if (callback) callback();
  }, 2000); // Delay
};

function funcao3(callback){
  setTimeout(function() {
    console.log('Função 3');
    if (callback) callback();
  }, 1000); // Delay
};

function fim(){
  console.log("Acabo");
};

funcao1(funcao2(funcao3(fim()))); // Por as funções ocorrem na ordem incorreta por causa da do delay
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Acabo
Função 3
Função 2
Função 1

[Done] exited with code=0 in 3.037 seconds
```

```
function funcao1(callback){
  setTimeout(function() {
    console.log('Função 1');
    if (callback) callback();
  }, 2500); // Delay
};

function funcao2(callback){
  setTimeout(function() {
    console.log('Função 2');
```

```
    if (callback) callback();
  }, 2000); // Delay
};

function funcao3(callback){
  setTimeout(function() {
    console.log('Função 3');
    if (callback) callback();
  }, 1000); // Delay
};

function fim(){
  console.log("Acabo");
};

funcao1(f1Callback);

function f1Callback(){
  funcao2(f2Callback);
};

function f2Callback(){
  funcao3(f3Callback);
};

function f3Callback(){
  fim();
};

// Dessa forma criando funções de Callback conseguimos fazer com que as
funções sejam executadas na ordem correta
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Função 1
Função 2
Função 3
Acabo

[Done] exited with code=0 in 6.091 seconds
```

3 IIFE – Immediately invoked function expression

Dessa forma as funções não são declaradas e são executadas assim que criadas

```
(function(){
    console.log("Essa função vai ser executada imediatamente!")
})();
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Essa função vai ser executada imediatamente!

[Done] exited with code=0 in 0.547 seconds
```

As vezes não queremos que as funções sejam declaradas, pois podem interferir no escopo global

```
const nome = "Um nome";

(function(){
    const nome = "Outro nome";
    console.log(nome);
})();

console.log(nome);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Outro nome
Um nome

[Done] exited with code=0 in 0.573 seconds
```

Mesmo assim podemos passar parâmetros para dentro dela

```
const nome = "Thiago";

(function(nome){
    console.log(nome);
})(nome);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Thiago

[Done] exited with code=0 in 0.547 seconds
```

4 Factory Funtions

Essa é uma função usada para criar objetos

```
function criaPessoa(nome){
    return {
        nome,
        fala (assunto){
            return `${nome} está falando sobre ${assunto}`;
        }
    };
}

const pessoa = criaPessoa("Luiz");
console.log(pessoa);
console.log(pessoa.fala("JavaScript"));
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Luiz', fala: [Function: fala] }
Luiz está falando sobre JavaScript

[Done] exited with code=0 in 0.624 seconds
```

Mas há alguns cuidados que devem ser tomados

```
function criaPessoa(name, height, weight){
    return {
        nome: name,
        altura: height,
        peso: weight,
        imc (assunto){
            let indice = (peso / altura**2).toFixed(2);
            return `${nome} tem o IMC de ${indice}`;
        }
    };
}

const pessoa = criaPessoa("Luiz", 1.7, 65);
console.log(pessoa);
console.log(pessoa.imc());
```

Esse código por exemplo retorna um erro, já que a função não consegue acessar algumas das informações necessárias

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Luiz', altura: 1.7, peso: 65, imc: [Function: imc] }
c:\Users\liq1ct\Desktop\Aula JS\script.js:7
        let indice = (peso / altura**2).toFixed(2);
                      ^

ReferenceError: peso is not defined
    at Object.imc (c:\Users\liq1ct\Desktop\Aula JS\script.js:7:26)
    at Object.<anonymous> (c:\Users\liq1ct\Desktop\Aula JS\script.js:15:20)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47

[Done] exited with code=1 in 0.541 seconds
```

Para resolver isso temos que usar o **this**

```
1  function criaPessoa(name, height, weight){
2      return {
3          nome: name,
4          altura: height,
5          peso: weight,
6          imc (assunto){
7              let indice = (this.peso / this.altura**2).toFixed(2);
8              return `${this.nome} tem o IMC de ${indice}`;
9          }
10     };
11 }
12
13 const pessoa = criaPessoa("Luiz", 1.7, 65);
14 console.log(pessoa);
15 console.log(pessoa.imc());
```

Dessa forma o **this** faz com que o método consiga acessar o informação necessária, já que ele assume o próprio objeto, por exemplo this.peso – pessoa.peso são a mesma coisa

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Luiz', altura: 1.7, peso: 65, imc: [Function: imc] }
Luiz tem o IMC de 22.49

[Done] exited with code=0 in 0.526 seconds
```

5 Set e Get

Uma dificuldade que pode acontecer é quando temos que criar um atributo com base nos outros, pois se mudarmos o atributo criado, os que foram usados para criar não mudam junto

```
function criaPessoa(name, last_name){
    return {
        nome: name,
        sobrenome: last_name,
        nomeCompleto: this.nome + " " + this.sobrenome,
    };
}

const pessoa = criaPessoa("Luiz", "Santos");

console.log(pessoa.nome);
console.log(pessoa.sobrenome);
console.log(pessoa.nomeCompleto);

pessoa.nomeCompleto = "Outro nome";
console.log("\n");

console.log(pessoa.nome);
console.log(pessoa.sobrenome);
console.log(pessoa.nomeCompleto);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Luiz
Santos
Luiz Santos


Luiz
Santos
Outro nome

[Done] exited with code=0 in 0.598 seconds
```

Para resolver isso temos que usar o **Get** e **Set**

Com o **get** já conseguimos resolver uma parte do problema, pois ele faz a parte de exibir, ou seja, só é chamado quando queremos puxar esse elemento

```
function criaPessoa(name, last_name){
    return {
        nome: name,
        sobrenome: last_name,

        get nomeCompleto(){
            return `${this.nome} ${this.sobrenome}`;
        },
```

```
    };
}

const pessoa = criaPessoa("Luiz", "Santos");

console.log(pessoa.nome);
console.log(pessoa.sobrenome);
console.log(pessoa.nomeCompleto);

pessoa.nomeCompleto = "Outro nome";
console.log("\n");

console.log(pessoa.nome);
console.log(pessoa.sobrenome);
console.log(pessoa.nomeCompleto);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Luiz
Santos
Luiz Santos


Luiz
Santos
Luiz Santos

[Done] exited with code=0 in 0.565 seconds
```

Diferente do **set** que é chamado quando queremos modificar um elemento

```
function criaPessoa(name, last_name){
    return {
        nome: name,
        sobrenome: last_name,

        get nomeCompleto(){
            return `${this.nome} ${this.sobrenome}`;
        },

        set nomeCompleto(valor){ // O valor é o elemento que foi enviado ao tentar modificar o atributo
            valor = valor.split(' ');
            this.nome = valor.shift();
            this.sobrenome = valor.join(' ');
        },
    };
}

const pessoa = criaPessoa("Luiz", "Santos");

console.log(pessoa.nome);
console.log(pessoa.sobrenome);
console.log(pessoa.nomeCompleto);

```

```
24  pessoa.nomeCompleto = "Outro nome";
25  console.log("\n");
26
27  console.log(pessoa.nome);
28  console.log(pessoa.sobrenome);
29  console.log(pessoa.nomeCompleto);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Luiz
Santos
Luiz Santos


Outro
nome
Outro nome

[Done] exited with code=0 in 0.692 seconds
```

6 Construtor Functions

Funções construtoras são outra forma de criar objetos

```
function Pessoa(nome, sobrenome){
    this.nome = nome;
    this.sobrenome = sobrenome;

    const metodoPrivado = () => console.log("Esse metodo não aparece para ser chamado");
    this.metodo = () => {
        console.log("Esse metodo pode ser chamado e chamar os metodos internos");
        metodoPrivado();
    };
}

const pessoa = new Pessoa('Marcio', 'Antonio');

pessoa.metodo();
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Esse metodo pode ser chamado e chamar os metodos internos
Esse metodo não aparece para ser chamado

[Done] exited with code=0 in 0.524 seconds
```

7 Funções Recursivas

Funções recursivas são funções que se chamam de volta, parecido com laços de repetição

```
function recursiva(i){
    if (i > 10) return;
    console.log(i);
    i++;
    recursiva(i);
}

recursiva(0);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
0
1
2
3
4
5
6
7
8
9
10

[Done] exited with code=0 in 0.539 seconds
```

8 Funções geradoras

As funções geradoras, que são caracterizadas pelo * depois de function, são funções capazes de retornar valores diferentes cada vez que são chamadas

Um exemplo de gerador finito é

```
function* geradorFinito(){
    yield "Valor 1";
    yield "Valor 2";
    yield "Valor 3";
}

const gerador = geradorFinito();
console.log(gerador.next());
console.log(gerador.next());
console.log(gerador.next());
console.log(gerador.next());
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
object
{ value: 'Valor 1', done: false }
{ value: 'Valor 2', done: false }
{ value: 'Valor 3', done: false }
{ value: undefined, done: true }

[Done] exited with code=0 in 0.513 seconds
```

Caso usado em um for ele ira parar assim que não houveram mais valores para serem enviados

```
function* geradorFinito(){
    yield "Valor 1";
    yield "Valor 2";
    yield "Valor 3";
}

const gerador = geradorFinito();
for (let valor of gerador){
    console.log(valor);
}
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Valor 1
Valor 2
Valor 3

[Done] exited with code=0 in 0.534 seconds
```

Mas podemos criar geradores infinitos também

```
function* geradorInfinito(){
    let i = 0;
    while (true){
        yield i;
        i++;
    };
}

const gerador = geradorInfinito();

console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
console.log(gerador.next().value);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
0
1
2
3
4
5
6
7

[Done] exited with code=0 in 0.594 seconds
```

Podemos fazer também que geradores que funcionam entre si

```
function* gerador1(){
    yield 0;
    yield 1;
    yield 2;
}
function* gerador2(){
    yield* gerador1();
    yield 3;
    yield 4;
    yield 5;
}

const gerador = gerador2();
for (let valor of gerador){
    console.log(valor);
}
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
0
1
2
3
4
5

[Done] exited with code=0 in 0.533 seconds
```

O **return** serve para terminar com a função geradora

```
1  function* geradorComReturn(){
2      yield () => console.log("Yield 1");
3      return () => console.log("Return");
4      yield () => console.log("Yield 2");
5  }
6  gerador = geradorComReturn();
7
8  yield1 = gerador.next().value;
9  yield2 = gerador.next().value;
10 yield3 = gerador.next().value;
11
12 yield1()
13 yield2()
14 yield3()
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Yield 1
Return
c:\Users\liq1ct\Desktop\Aula JS\script.js:14
yield3()
^

TypeError: yield3 is not a function
    at Object.<anonymous> (c:\Users\liq1ct\Desktop\Aula JS\script.js:14:1)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47

[Done] exited with code=1 in 0.551 seconds
```

Ou seja, caso seja usado um return o resto do código da função geradora passa a não ser executada

9 DOM para selecionar parte do HTML

querySelector( ) – retorna o primeiro elemento que tenha aquele parâmetro

querySelectorAll( ) – retorna todos os elementos que tenham aquele parâmetro

getElementById( ) – retorna o primeiro elemento que tenha aquele Id

getElementsByName( ) – retorna todos os elementos que tenham aquele name

getElementsByTagName( ) – retorna todos os elementos aquela tag

addEventListener( ) – adiciona um manipulador de eventos no documento

classList – retorna a lista de classes de uma elemento

10 Exercício de criar uma calculadora

Crie uma calculadora simples, utilizando tudo que já sabe sobre funções e DOM

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="style.css" type="text/css" rel="stylesheet">
    <title>Calculadora</title>
</head>
<body>
    <div class="container">
        <h1>Calculadora</h1>

        <table class="calculadora">
            <tr>
                <td colspan="4"><input type="text" class="display"></td>
            </tr>

            <tr>
                <td><button class="btn btn-clear">C</button></td>
                <td><button class="btn btn-num">(</button></td>
                <td><button class="btn btn-num">)</button></td>
                <td><button class="btn btn-num">/</button></td>
            </tr>

            <tr>
                <td><button class="btn btn-num">7</button></td>
                <td><button class="btn btn-num">8</button></td>
                <td><button class="btn btn-num">9</button></td>
                <td><button class="btn btn-num">*</button></td>
            </tr>

            <tr>
                <td><button class="btn btn-num">4</button></td>
                <td><button class="btn btn-num">5</button></td>
                <td><button class="btn btn-num">6</button></td>
                <td><button class="btn btn-num">+</button></td>
            </tr>

            <tr>
                <td><button class="btn btn-num">1</button></td>
                <td><button class="btn btn-num">2</button></td>
                <td><button class="btn btn-num">3</button></td>
                <td><button class="btn btn-num">-</button></td>
            </tr>

            <tr>
                <td><button class="btn btn-num">.</button></td>
```

```
                <td><button class="btn btn-num">0</button></td>
                <td><button class="btn btn-del">«</button></td>
                <td><button class="btn btn-eq">=</button></td>
            </tr>
        </table>
    </div>
    <script src="script.js"></script>
</body>
</html>
```

CSS

```
body{
    background-color: rgb(0, 126, 175);
}

.container{
    text-align: center;
    width: 40%;
    height: 500px;
    margin: auto;
    background-color: aliceblue;
    border-radius: 10px;
}

h1{
    padding-top: 55px;
}
.calculadora{
    text-align: center;
    width: 400px;
    margin: auto;
    background-color: aliceblue;
    border-radius: 10px;
}
.display{
    font-size: 2em;
    width: 100%;
    text-align: right;
}

.btn{
    width: 100%;
    height: 50px;
    background-color: #dfdfdf;
}
.btn:hover{
    background-color: #9e9e9e;
}
```

11 Correção

JavaScript

```javascript
function Calculadora(){
    this.display = document.querySelector('.display');

    this.capturaCliques = () => {
        document.addEventListener('click', event => {
            const elemento = event.target; // Identifica onde o evento aconteceu
            if (elemento.classList.contains('btn-num')) this.addNum(elemento);
            if (elemento.classList.contains('btn-clear')) this.clear();
            if (elemento.classList.contains('btn-del')) this.del();
            if (elemento.classList.contains('btn-eq')) this.equal();
        });
    };

    this.capturaEnter = () => {
        document.addEventListener('keypress', btn => {
            if (btn.keyCode === 13) this.equal();
        });
    };

    this.addNum = elemento => {
        this.display.value += elemento.innerText;
        this.display.focus(); // Retorna o foco para o display
    };

    this.clear = () => this.display.value = "";

    this.del = () => this.display.value = this.display.value.slice(0, -1);

    this.equal = () => {
        try {
            const conta = eval(this.display.value); // Realiza uma conta que esteja no formato de String
            this.display.value = conta;
        } catch(e){
            alert("Conta inválida");
            return;
        }
    };

    this.inicia = () => {
        this.capturaCliques();
        this.capturaEnter();
    };
}

const calculadora = new Calculadora();
calculadora.inicia();
```

12 Object.defineProperty( ) e Object.defineProperties( )

Ao criarmos um objeto as vezes é preciso fazer com que alguns dados não sejam mostrados ou não possam ser modificados

```
1  function Produto (nome, preco, estoque) {
2      this.nome = nome;
3      this.preco = preco;
4      this.estoque = estoque;
5  }
6
7  const p1 = new Produto("Camiseta", 20, 3);
8
9  console.log(p1.estoque);
10 p1.estoque = 500;
11 console.log(p1.estoque);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
3
500

[Done] exited with code=0 in 0.569 seconds
```

Object.defineProperty( ) é usado para definir parâmetros de um atributo de um objeto

```
1  function Produto (nome, preco, estoque) {
2      this.nome = nome;
3      this.preco = preco;
4      Object.defineProperty(this, 'estoque', {
5          enumerable: false, // Define se a chave pode ser mostrada junto
   das outras chaves
6          value: estoque, // Define o valor da chave
7      });
8  }
9
10 const p1 = new Produto("Camiseta", 20, 3);
11
12 console.log(Object.keys(p1));
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
[ 'nome', 'preco' ]

[Done] exited with code=0 in 0.583 seconds
```

No caso não foi mostrado a chave estoque por causa do enumerable estar como false

```
1  function Produto (nome, preco, estoque) {
```

```
    this.nome = nome;
    this.preco = preco;
    Object.defineProperty(this, 'estoque', {
        enumerable:true, // Define se a chave pode ser mostrada
        value: estoque, // Define o valor da chave
        writable: false, // Define se o valor da chave pode ser alterado
    });
}

const p1 = new Produto("Camiseta", 20, 3);

console.log(p1.estoque);
p1.estoque = 500;
console.log(p1.estoque);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
3
3

[Done] exited with code=0 in 0.571 seconds
```

No caso o valor do estoque não foi mudado por causa do writable estar como false

```
function Produto (nome, preco, estoque) {
    this.nome = nome;
    this.preco = preco;
    Object.defineProperty(this, 'estoque', {
        enumerable:true, // Define se a chave pode ser mostrada
        value: estoque, // Define o valor da chave
        writable: false, // Define se o valor da chave pode ser alterado
        configurable: false // Define se a chave pode ser reconfigurada ou
deletada
    });
}

const p1 = new Produto("Camiseta", 20, 3);

console.log(p1.estoque);
delete p1.estoque;
console.log(p1.estoque);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
3
3

[Done] exited with code=0 in 0.578 seconds
```

Nesse caso o estoque não foi deletado por causa do configurable estar como false

```
function Produto (nome, preco, estoque) {
    this.nome = nome;
```

```
    this.preco = preco;

    Object.defineProperty(this, 'estoque', {
        enumerable:true, // Define se a chave pode ser mostrada
        value: estoque, // Define o valor da chave
        writable: false, // Define se o valor da chave pode ser alterado
        configurable: false // Define se a chave pode ser reconfigurada ou deletada
    });

    Object.defineProperty(this, 'estoque', {
        enumerable:false, // Define se a chave pode ser mostrada
        value: estoque, // Define o valor da chave
        writable: true, // Define se o valor da chave pode ser alterado
        configurable: true // Define se a chave pode ser reconfigurada ou deletada
    });
}

const p1 = new Produto("Camiseta", 20, 3);

console.log(p1);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
c:\Users\liq1ct\Desktop\Aula JS\script.js:12
    Object.defineProperty(this, 'estoque', {
           ^

TypeError: Cannot redefine property: estoque
    at Function.defineProperty (<anonymous>)
    at new Produto (c:\Users\liq1ct\Desktop\Aula JS\script.js:12:12)
    at Object.<anonymous> (c:\Users\liq1ct\Desktop\Aula JS\script.js:20:12)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47

[Done] exited with code=1 in 0.594 seconds
```

Nesse caso ocorre um erro por conta de tentar reconfigurar uma chave que não pode ser reconfigurada por causa do configurable estar false

Tudo isso também se aplica ao Object.defineProperties, que apenas muda um pouco a sintaxe

```
function Produto (nome, preco, estoque) {
    Object.defineProperties(this, {
        nome: {
            enumerable: true,
            value: nome,
            writable: false,
            configurable: false
        },
        preco: {
            enumerable: true,
            value: preco,
            writable: true,
            configurable: false
```

```
        },
        estoque: {
            enumerable: true,
            value: estoque,
            writable: true,
            configurable: false
        }
    });
}

const p1 = new Produto("Camiseta", 20, 3);

console.log(p1);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Produto { nome: 'Camiseta', preco: 20, estoque: 3 }

[Done] exited with code=0 in 0.546 seconds
```

13 Getters e Setters

Usando get e set não temos que usar value e writable, já que vamos fazer uma função que ira cuidar disso

```
function Produto (nome, preco, estoque) {
    this.nome = nome;
    this.preco = preco;
    Object.defineProperty(this, 'estoque', {
        enumerable:true,
        configurable: false,
        get: () => {
            return estoque
        },
        set: (valor) => {
            if (typeof(valor) !== 'number') {
                throw new TypeError("Valor inválido");
            }
            estoque = valor;
        }
    });
}

const p1 = new Produto("Camiseta", 20, 3);

console.log(p1.estoque);
p1.estoque = 500;
console.log(p1.estoque);
p1.estoque = "Não é número";
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
3
500
c:\Users\liq1ct\Desktop\Aula JS\script.js:12
                throw new TypeError("Valor inválido");
                ^

TypeError: Valor inválido
    at Produto.set [as estoque] (c:\Users\liq1ct\Desktop\Aula JS\script.js:12:23)
    at Object.<anonymous> (c:\Users\liq1ct\Desktop\Aula JS\script.js:24:12)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47

[Done] exited with code=1 in 0.571 seconds
```

14 Métodos úteis para objetos

Object.assign( ) é usado para criar um objeto juntando dois, caso queria apenas duplicar um objeto pode passar o primeiro como objeto vazio

```
const produto = { nome: "Produto", preco: 1.8 };
const caneca = Object.assign({}, produto);

caneca.nome = "Caneca";
produto.preco = 10;

console.log(produto);
console.log(caneca);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Produto', preco: 10 }
{ nome: 'Caneca', preco: 1.8 }

[Done] exited with code=0 in 0.599 seconds
```

Object.getOwnPropertyDescriptor( ), mostra os parâmetros de um atributo do objeto

```
const produto = { nome: "Produto", preco: 1.8 };

console.log(produto);
console.log(Object.getOwnPropertyDescriptor(produto, 'nome'));
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Produto', preco: 1.8 }
{
  value: 'Produto',
  writable: true,
  enumerable: true,
  configurable: true
}

[Done] exited with code=0 in 0.608 seconds
```

Object.entries( ) retorna um array que tem tanto as chaves quanto os valores

```
const produto = { nome: "Produto", preco: 1.8 };

console.log(produto);
for (let [chave, valor] of Object.entries(produto)) {
    console.log(chave, valor);
}
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{ nome: 'Produto', preco: 1.8 }
nome Produto
preco 1.8

[Done] exited with code=0 in 0.603 seconds
```

15 Prototypes

Todos os objetos tem uma referencia interna para um protótipo (__proto__) que vem da propriedade prototype da função construtora que foi usada para cria-lo. Quanto tentamos acessar um membro de um objeto, primeiro o motor do JS vai tentar encontrar este membro no próprio objeto e depois a cadeia de protótipos é usada até o topo (null) até encontrar (ou não) tal membro.

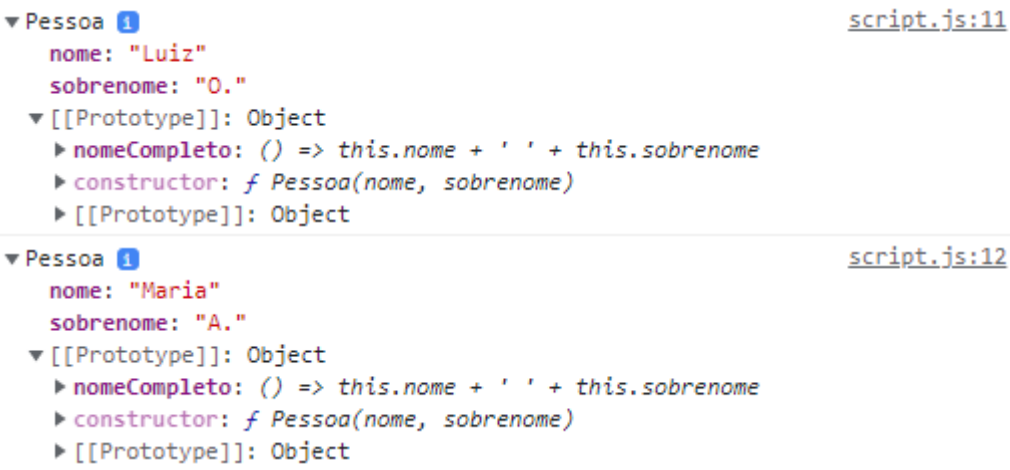
O prototype é usado para métodos que são usados por todos os objetos da mesma forma

```
1  function Pessoa(nome, sobrenome) {
2      this.nome = nome;
3      this.sobrenome = sobrenome;
4      this.nomeCompleto = () => this.nome + ' ' + this.sobrenome;
5  }
6  const pessoa1 = new Pessoa('Luiz', 'O.');
7  const pessoa2 = new Pessoa('Maria', 'A.');
8
9  console.dir(pessoa1);
10 console.dir(pessoa2);
```

```
▼Pessoa i                                                    script.js:9
    nome: "Luiz"
  ▶ nomeCompleto: () => this.nome + ' ' + this.sobrenome
    sobrenome: "O."
  ▶ [[Prototype]]: Object
▼Pessoa i                                                    script.js:10
    nome: "Maria"
  ▶ nomeCompleto: () => this.nome + ' ' + this.sobrenome
    sobrenome: "A."
  ▶ [[Prototype]]: Object
```

Como mostrado a função nomeCompleto é criado dentro dos dois objetos, mesmo sendo igual para os dois

```
1  function Pessoa(nome, sobrenome) {
2      this.nome = nome;
3      this.sobrenome = sobrenome;
4  }
5
6  Pessoa.prototype.nomeCompleto = () => this.nome + ' ' + this.sobrenome;
7
8  const pessoa1 = new Pessoa('Luiz', 'O.');
9  const pessoa2 = new Pessoa('Maria', 'A.');
10
11 console.dir(pessoa1);
12 console.dir(pessoa2);
```



Agora nesse os dois continuam tendo o método de nomeCompleto, porem esse método foi criado só uma vez e esta referenciado aos dois objetos

```
function Produto (nome, preco){
    this.nome = nome;
    this.preco = preco;
}

Produto.prototype.desconto = function(percentual) {this.preco = this.preco * (1 - (percentual/100))};
Produto.prototype.aumento = function(percentual) {this.preco = this.preco * (1 + (percentual/100))};

const produto1 = new Produto("Camiseta", 50);

const produto2 = {
    nome: "Caneca",
    preco: 15
};
Object.setPrototypeOf(produto2, Produto); // Define um prototype

const produto3 = Object.create(Produto.prototype, {
    nome: {
        enumerable: true,
        value: "Terceiro item",
        writable: true,
        configurable: false,
    },
    preco: {
        enumerable: true,
        value: 42,
        writable: true,
        configurable: false,
    },
}); // Cria o objeto já tendo um prototype

console.dir(produto1);
console.dir(produto2);
console.dir(produto3);
```

Para Factory Functions há duas formas de se criar Prototypes

```
function criaProduto (nome, preco, estoque) {
    const produtoPrototype = {
        desconto(percentual) {
            this.preco = this.preco * (1 - (percentual/100));
        },
        aumento(percentual) {
            this.preco = this.preco * (1 + (percentual/100));
        }
    };

    return Object.create(produtoPrototype, {
        nome: { value: nome },
        preco: { value: preco },
        estoque: { value: estoque }
    });
}

const produto = criaProduto("Genérico", 10, 10);
```

```
> produto
< ▼{nome: 'Genérico', preco: 10, estoque: 10}
    estoque: 10
    nome: "Genérico"
    preco: 10
  ▼[[Prototype]]: Object
    ▶aumento: ƒ aumento(percentual)
    ▶desconto: ƒ desconto(percentual)
    ▶[[Prototype]]: Object
>
```

Ou

```
1  const desconto = {
2      desconto(percentual) {
3          this.preco = this.preco * (1 - (percentual/100));
4      }
5  };
6
7  const aumento = {
8      aumento(percentual) {
9          this.preco = this.preco * (1 + (percentual/100));
10     }
11 };
12
13 const produtoPrototype = Object.assign({}, desconto, aumento);
14
15 function criaProduto (nome, preco, estoque) {
16     return Object.create(produtoPrototype, {
17         nome: { value: nome },
18         preco: { value: preco },
19         estoque: { value: estoque }
20     });
21 }
22
23 const produto = criaProduto("Genérico", 10, 10);
```

```
> produto
< ▼{nome: 'Genérico', preco: 10, estoque: 10}
    estoque: 10
    nome: "Genérico"
    preco: 10
  ▼[[Prototype]]: Object
    ▶aumento: ƒ aumento(percentual)
    ▶desconto: ƒ desconto(percentual)
    ▶[[Prototype]]: Object
>
```

16 Herança

```
1  function Produto (nome, preco, estoque){
2      this.nome = nome;
3      this.preco = preco;
4      this.estoque = estoque;
5  }
6
7  Produto.prototype.desconto = function(percentual) {this.preco = this.preco
   * (1 - (percentual/100))};
8  Produto.prototype.aumento = function(percentual) {this.preco = this.preco
   * (1 + (percentual/100))};
9
10 function Camiseta (nome, preco, estoque, cor){
11     Produto.call(this, nome, preco, estoque); // Chama a função produto
12     this.cor = cor;
13 }
14 Camiseta.prototype = Object.create(Produto.prototype); // Cria um
   prototype para camiseta igual o de Produto
15 Camiseta.prototype.constructor = Camiseta; // Ao criar um prototype igual
   de Produto, perdemos o constructor de Camiseta, então temos que cria-lo de
   novo
16
17 const produto = new Produto("Genérico", 10, 10);
18 const camiseta = new Camiseta("Camiseta", 50, 15, "Branca");
19
20 console.dir(produto);
21 console.dir(camiseta);
```

```
▼Produto i                                                    script.js:20
   estoque: 10
   nome: "Genérico"
   preco: 10
  ▼[[Prototype]]: Object
    ▶aumento: ƒ (percentual)
    ▶desconto: ƒ (percentual)
    ▶constructor: ƒ Produto(nome, preco, estoque)
    ▶[[Prototype]]: Object
▼Camiseta i                                                   script.js:21
   cor: "Branca"
   estoque: 15
   nome: "Camiseta"
   preco: 50
  ▼[[Prototype]]: Produto
    ▶constructor: ƒ Camiseta(nome, preco, estoque, cor)
    ▼[[Prototype]]: Object
      ▶aumento: ƒ (percentual)
      ▶desconto: ƒ (percentual)
      ▶constructor: ƒ Produto(nome, preco, estoque)
      ▶[[Prototype]]: Object
>
```

17 Exercício de validar um CPF

Um CPF é validado com uma conta que gera os dois últimos números utilizando os 9 primeiros, e também não sendo uma sequencia, por exemplo:

705.484.450-52 é um CPF valido, pois

7 * 10 = 70

0 * 9 = 0

5 * 8 = 40

4 * 7 = 28

8 * 6 = 48

4 * 5 = 20

4 * 4 = 16

5 * 3 = 15

0 * 2 = 0

Soma de tudo = 237

11 - (237 % 11) = 5 (Primeiro dígito)

Se o número for maior que 9, deve ser considerado 0.

7 * 11 = 77

0 * 10 = 0

5 * 9 = 45

4 * 8 = 32

8 * 7 = 56

4 * 6 = 24

4 * 5 = 20

5 * 4 = 20

0 * 3 = 0

5 * 2 = 10

Soma de tudo = 284

11 - (284 % 11) = 2 (Segundo dígito)

Se o número for maior que 9, deve ser considerado 0.

Mas 111.111.111-11 não é valido porque mesmo a conta estando correta ele é uma sequencia

17.1 Dicas:

.replace(/\D+/g, '') faz com que a string fique com apenas números nela

Array.from( ) faz uma string ser transformada em array

.repeat( ) faz com que algo seja repetido uma determinada quantia de vezes

18 Correção

```
function ValidaCPF(cpfEnviado){

    Object.defineProperty(this, 'cpfLimpo', {
        get: function() {
            return cpfEnviado.replace(/\D+/g, ''); // é utilizado o get apenas para limpar o número ao receber
        }
    });
}

ValidaCPF.prototype.valida = function() {
    if (typeof this.cpfLimpo === 'undefined' || this.cpfLimpo.length !== 11 || this.isSequencia()) return false; // caso o cpf não seja enviado, tenha um tamanho diferente de 11 ou seja uma sequencia retorna false

    let cpfParcial = this.cpfLimpo.slice(0, -2);
    let digito = this.geraDigito(cpfParcial);

    cpfParcial = cpfParcial + String(digito);
    digito = this.geraDigito(cpfParcial);

    cpfParcial = cpfParcial + String(digito);

    return cpfParcial === this.cpfLimpo ? true : false;
};

ValidaCPF.prototype.geraDigito = function(cpfParcial) {
    const cpfArray = Array.from(cpfParcial);

    let count = cpfArray.length + 1;
    const total = cpfArray.reduce((soma, valor) => {
        soma += Number(valor) * count;
        count--;
        return soma;
    }, 0);
    const digito = 11 - (total % 11);

    return digito > 9 ? 0 : digito;
}

ValidaCPF.prototype.isSequencia = function() {
    return this.cpfLimpo[0].repeat(11) === this.cpfLimpo;
}

const cpf = new ValidaCPF( Número do CPF );
console.log(cpf.valida());
```

19 Polimorfismo

Capacidade de funções que se originam da mesma função mãe agirem de formas diferentes

```
function Conta(agencia, conta, saldo){
    this.agencia = agencia;
    this.conta = conta;
    this.saldo = saldo;
}

Conta.prototype.sacar = function(valor) {
    if (valor > this.saldo) {
        console.log("Saldo insuficiente");
        return;
    }
    console.log(`Saque de R$ ${valor.toFixed(2)}`);
    this.saldo -= valor;
    this.verSaldo();
};

Conta.prototype.depositar = function(valor) {
    console.log(`Deposito de R$ ${valor.toFixed(2)}`);
    this.saldo += valor;
    this.verSaldo();
};

Conta.prototype.verSaldo = function() {
    console.log(`Ag./Conta: ${this.agencia}/${this.conta}\nSaldo: R$ ${thi
s.saldo.toFixed(2)}\n`);
};


function Conta_Corrente(agencia, conta, saldo, limite) {
    Conta.call(this, agencia, conta, saldo);
    this.limite = limite;
}

Conta_Corrente.prototype = Object.create(Conta.prototype);

Conta_Corrente.prototype.constructor = Conta_Corrente;

Conta_Corrente.prototype.sacar = function(valor) {
    if (valor > (this.saldo + this.limite)) {
        console.log("Saldo insuficiente");
        return;
    }
    console.log(`Saque de R$ ${valor.toFixed(2)}`);
    this.saldo -= valor;
    this.verSaldo();
};


function Conta_Poupanca(agencia, conta, saldo) {
    Conta.call(this, agencia, conta, saldo);
}

```

```
52  Conta_Poupanca.prototype = Object.create(Conta.prototype);
53
54  Conta_Poupanca.prototype.constructor = Conta_Poupanca;
55
56  const conta_corrente = new Conta_Corrente('11111-11', '1234', 100, 150);
57  const conta_poupanca = new Conta_Poupanca('11111-11', '5678', 100);
58
59  conta_corrente.sacar(150);
60  conta_poupanca.sacar(150);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Ag./Conta: 11111-11/1234
Saldo: R$ 100.00

Ag./Conta: 11111-11/5678
Saldo: R$ 100.00

Saque de R$ 150.00
Ag./Conta: 11111-11/1234
Saldo: R$ -50.00

Saldo insuficiente

[Done] exited with code=0 in 0.586 seconds
```

20 Objeto Map( )

Quando queremos criar um criar um objeto enumerados com o próprio id podemos ter problemas dependendo do código que criar e do que precisamos, por exemplo:

```
const pessoas = [
    { id: 3, nome: "Luiz"},
    { id: 2, nome: "Maria"},
    { id: 1, nome: "Helena"},
];

const novasPessoas = {};
for (const pessoa of pessoas) {
    const id = pessoa.id;
    novasPessoas[id] = { ...pessoa };
}

console.log(novasPessoas);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
{
  '1': { id: 1, nome: 'Helena' },
  '2': { id: 2, nome: 'Maria' },
  '3': { id: 3, nome: 'Luiz' }
}

[Done] exited with code=0 in 0.589 seconds
```

Nesse exemplo temos dois problemas, sendo o primeiro o que o id ficou em string e o segundo que perdemos a ordem, para resolver isso podemos usar o objeto map nesse caso

```
const pessoas = [
    { id: 3, nome: "Luiz"},
    { id: 2, nome: "Maria"},
    { id: 1, nome: "Helena"},
];

const novasPessoas = {};
for (const pessoa of pessoas) {
    const id = pessoa.id;
    novasPessoas[id] = { ...pessoa };
}

console.log(novasPessoas);
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Map(3) {
  3 => { id: 3, nome: 'Luiz' },
  2 => { id: 2, nome: 'Maria' },
  1 => { id: 1, nome: 'Helena' }
}

[Done] exited with code=0 in 0.646 seconds
```

Dessa forma conseguimos criar um objeto onde não trocamos o tipo do id e também não perdemos a ordem das informações

21 Classes

A declaração de classes é bem mais fácil, já que os métodos são passados para o prototype automaticamente

```
class Pessoa {
    constructor(nome, sobrenome) {
        this.nome = nome;
        this.sobrenome = sobrenome;
    }

    falar() {
        console.log(`${this.nome} está falando`);
    }
}

const pessoa = new Pessoa('Luiz', 'Miranda');
```

```
> pessoa
< ▼Pessoa {nome: 'Luiz', sobrenome: 'Miranda'}
    nome: "Luiz"
    sobrenome: "Miranda"
  ▼[[Prototype]]: Object
    ▶constructor: class Pessoa
    ▶falar: ƒ falar()
    ▶[[Prototype]]: Object
>
```

A herança nas classes é bem mais direta também

```
class DispositivoEletronico {
    constructor (nome) {
        this.nome = nome;
        this.ligado = false;
    }

    ligar() {
        if (this.ligado) {
            console.log(this.nome + " já está ligado");
            return;
        }
        console.log("Ligando " + this.nome);
        this.ligado = true;
    }

    desligar() {
        if (!this.ligado) {
            console.log(this.nome + " já está desligado");
            return;
        }
        console.log("Desligando " + this.nome);
        this.ligado = false;
    }
```

```
}

class Smartphone extends DispositivoEletronico {
    constructor(nome, cor, modelo) {
        super(nome);
        this.cor = cor;
        this.modelo = modelo;
    }
}
```

Métodos de instância e estáticos

```
class classeExemplo {
    constructor(nome) {
        this.nome = nome;
    }

    // Método de instância
    metodoInstancia() {
        console.log("Esse metodo é referente ao objeto");
        console.log(this);
    }

    // Método estático
    static metodoEstatico() {
        console.log("Esse metodo é referente a classe");
        console.log(this);
    }
}

const classe_de_exemplo = new classeExemplo("Exemplo");
classe_de_exemplo.metodoInstancia();
classeExemplo.metodoEstatico();
```

```
[Running] node "c:\Users\liq1ct\Desktop\Aula JS\script.js"
Esse metodo é referente ao objeto
classeExemplo { nome: 'Exemplo' }
Esse metodo é referente a classe
[class classeExemplo]

[Done] exited with code=0 in 0.655 seconds
```

22 Desafio de fazer uma verificação completa de formulário

HTML

```html
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Modelo</title>
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <section class="container">
    <h1>Formulário de cadastro</h1>
    <ul>
      <li>Nenhum campo pode estar vazio</li>
      <li>Usuário só poderá conter letras e/ou números</li>
      <li>Usuário deverá ter entre 3 e 12 caracteres</li>
      <li>Senha precisa ter entre 6 e 12 caracteres</li>
      <li>Senha e repetir senha devem ser iguais</li>
    </ul>

    <form action="/" method="POST" class='formulario'>
      <label>Nome</label>
      <input type="text" class="nome validar">
      <label>Sobrenome</label>
      <input type="text" class="sobrenome validar">
      <label>CPF</label>
      <input type="text" class="cpf validar">
      <label>Usuário</label>
      <input type="text" class="usuario validar">
      <label>Senha</label>
      <input type="password" class="senha validar">
      <label>Repetir Senha</label>
      <input type="password" class="repetir-senha validar">
      <button type="submit">Enviar</button>
    </form>
  </section>

  <script src="validaCPF.js"></script>
  <script src="main.js"></script>
</body>

</html>
```

CSS

```css
@import url('https://fonts.googleapis.com/css?family=Open+Sans:400,700&display=swap');
:root {
  --primary-color: rgb(17, 86, 102);
  --primary-color-darker: rgb(9, 48, 56);
}

* {
  box-sizing: border-box;
  outline: 0;
}

body {
  margin: 0;
  padding: 0;
  background: var(--primary-color);
  font-family: 'Open sans', sans-serif;
  font-size: 1.3em;
  line-height: 1.5em;
}

.container {
  max-width: 640px;
  margin: 50px auto;
  background: #fff;
  padding: 20px;
  border-radius: 10px;
}

form input, form label, form button {
  display: block;
  width: 100%;
  margin-bottom: 10px;
}

form input {
  font-size: 24px;
  height: 50px;
  padding: 0 20px;
}

form input:focus {
  outline: 1px solid var(--primary-color);
}

form button {
  border: none;
  background: var(--primary-color);
  color: #fff;
  font-size: 18px;
  font-weight: 700;
  height: 50px;
  cursor: pointer;
  margin-top: 30px;
}
```

```
55
56  form button:hover {
57    background: var(--primary-color-darker);
58  }
59  .error-text {
60    font-size: 12px;
61    color: red;
62  }
```

Dicas:

- Usar o validaCPF do ultimo exercício
- .match(/^[a-zA-Z0-9]+$/g) verifica se há apenas letras e números na string
- .previousElementSibling retorna o Elemento anterior do HTML
- .preventDefault tira as características de um elemento
- .submit consegue enviar um formulario pelo JS
- .insertAdjacentElement( ) Coloca um elemento ao lado de outro no HMTL

23 Correção

JavaScript

```
class ValidaFormulario {
    constructor() {
      this.formulario = document.querySelector('.formulario');
      this.eventos();
    }

    eventos() {
      this.formulario.addEventListener('submit', e => { // Captura o evento de submit
        this.handleSubmit(e);
      });
    }

    handleSubmit(e) {
      e.preventDefault(); // Não envia o formulario
      const camposValidos = this.camposSaoValidos();
      const senhasValidas = this.senhasSaoValidas();

      if(camposValidos && senhasValidas) {
        alert('Formulário enviado.');
        this.formulario.submit(); // Envia o formulario
      }
    }

    senhasSaoValidas() {
      let valid = true;

      const senha = this.formulario.querySelector('.senha');
      const repetirSenha = this.formulario.querySelector('.repetir-senha');

      if(senha.value !== repetirSenha.value) {
        valid = false;
        this.criaErro(senha, 'Campos senha e repetir senha precisar ser iguais.');
        this.criaErro(repetirSenha, 'Campos senha e repetir senha precisar ser iguais.');
      }

      if(senha.value.length < 6 || senha.value.length > 12) {
        valid = false;
        this.criaErro(senha, 'Senha precisa estar entre 6 e 12 caracteres.');
      }

      return valid;
    }

    camposSaoValidos() {
      let valid = true;

```

```
47        for(let errorText of this.formulario.querySelectorAll('.error-text')
) {
48          errorText.remove(); // Remove essa class
49        }
50
51        for(let campo of this.formulario.querySelectorAll('.validar')) {
52          const label = campo.previousElementSibling.innerText; // Pega o
elemento anterior
53
54          if(!campo.value) {
55            this.criaErro(campo, `Campo "${label}" não pode estar em
branco.`);
56            valid = false;
57          }
58
59          if(campo.classList.contains('cpf')) {
60            if(!this.validaCPF(campo)) valid = false;
61          }
62
63          if(campo.classList.contains('usuario')) {
64            if(!this.validaUsuario(campo)) valid = false;
65          }
66
67        }
68
69        return valid;
70      }
71
72      validaUsuario(campo) {
73        const usuario = campo.value;
74        let valid = true;
75
76        if(usuario.length < 3 || usuario.length > 12) {
77          this.criaErro(campo, 'Usuário precisa ter entre 3 e 12
caracteres.');
78          valid = false;
79        }
80
81        if(!usuario.match(/[a-zA-Z0-9]+/g)) { // Verifica se a string tem
apenas letras e numeros
82          this.criaErro(campo, 'Nome de usuário precisar conter apenas
letras e/ou números.');
83          valid = false;
84        }
85
86        return valid;
87      }
88
89      validaCPF(campo) {
90        const cpf = new ValidaCPF(campo.value);
91
92        if(!cpf.valida()) {
93          this.criaErro(campo, 'CPF inválido.');
94          return false;
95        }
96
97        return true;
98      }
99
100     criaErro(campo, msg) {
```

```
101         const div = document.createElement('div');
102         div.innerHTML = msg;
103         div.classList.add('error-text');
104         campo.insertAdjacentElement('afterend', div); // Coloca o Elemento
    apos o campo acabar
105       }
106     }
107
108     const valida = new ValidaFormulario();
```