

# MAIS JAVASCRIPT

---

Function Expression

## Function Declaration

---

São duas as formas mais comuns de declararmos uma função. A que utilizamos até o momento é chamado de Function Declaration.

```
function somar(a,b) {  
    return a + b;  
}  
  
somar(4,2); // 6
```

## Function Expression

---

É criada a partir da declaração de uma variável, na qual assinalamos uma função. Esta função pode ser anônima ou nomeada. A mesma poderá ser ativada através da variável assinalada.

```
const somar = function(a,b) {  
    return a + b;  
}  
  
somar(4,2); // 6
```

## Hoisting

Function Declarations são completamente definidas no momento do hoisting, já function expressions apenas serão definidas no momento da execução. Por isso a ordem da declaração de uma FE possui importância.

```
somar(4,2); // 6
dividir(4,2); // Erro

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

const dividir = function(a,b) {
    return a / b;
}
```

## Arrow Function

---

Podemos criar utilizando arrow functions.

```
const somar = (a, b) => a + b;  
somar(4,2); // 6
```

```
const quadrado = a => a * a;  
quadrado(4); // 16
```

## Estrutura / Preferência

---

Geralmente o uso entre expression / declaration é uma questão de preferência. Function Expression força a criação da mesma antes de sua ativação, o que pode contribuir para um código mais estruturado.

```
// Declarada como método de window
// vaza o escopo de bloco, como se
// fosse criada utilizando var
function somar(a,b) {
    return a + b;
}
const dividir = (a,b) => a / b;

somar(4,2);
dividir(4,2);
```

## IIFE - Immediately Invoked Function Expression

Antes da introdução de modules e da implementação do escopo de bloco, a forma mais comum utilizada para isolarmos o escopo de um código JavaScript era através das chamadas IIFE's.

```
var instrumento = 'Violão';

(function() {
    // código isolado do escopo global
    var instrumento = 'Guitarra';
    console.log(instrumento); // Guitarra
})();

console.log(instrumento); // Violão
```

## IIFE - Arrow Function

Compiladores ainda transformam modules em IIFE's para manter a compatibilidade com browsers antigos.

```
const instrumento = 'Violão';

(() => {
  const instrumento = 'Guitarra';
  console.log(instrumento); // Guitarra
})();

console.log(instrumento); // Violão
```

## Exercícios

---

```
// Remova o erro
priceNumber('R$ 99,99');
const priceNumber = n => +n.replace('R$', '').replace(',', '.');

// Crie uma IIFE e isole o escopo
// de qualquer código JS.

// Como podemos utilizar
// a função abaixo.
const active = callback => callback();
```

# MAIS JAVASCRIPT

---

Factory Function

## Factory Function

São funções que retornam um objeto sem a necessidade de utilizarmos a palavra chave new. Possuem basicamente a mesma função que constructor functions / classes.

```
function createButton(text) {  
    function element() {  
        const buttonElement = document.createElement('button');  
        buttonElement.innerText = text;  
        return buttonElement;  
    }  
    return {  
        element: element,  
        text: text,  
    }  
}  
  
const comprarBtn = createButton('Comprar');
```

# Métodos / Variáveis privadas

Uma das vantagens é a possibilidade de criarmos métodos / variáveis privadas.

```
function criarPessoa(nome, sobrenome) {  
    const nomeCompleto = `${nome} ${sobrenome}`;  
  
    function andar() {  
        return `${nomeCompleto} andou`;  
    }  
    function nadar() {  
        return `${nomeCompleto} nadou`;  
    }  
    return {  
        nome,  
        sobrenome,  
        andar,  
        nadar,  
    }  
}  
  
const designer = criarPessoa('André', 'Rafael');
```

# Ice Factory

Podemos impedir que os métodos e propriedades sejam modificados com `Object.freeze()`. Ideia inicial de Douglas Crockford.

```
'use strict';

function criarPessoa(nome, sobrenome) {
  const nomeCompleto = `${nome} ${sobrenome}`;
  function andar() {
    return `${nomeCompleto} andou`;
  }
  return Object.freeze({
    nome,
    sobrenome,
    andar,
  });
}

const designer = criarPessoa('André', 'Rafael');
```

## Constructor Function / Factory Function

---

Uma das vantagens da Factory Function é a possibilidade de iniciarmos a mesma sem a utilização da palavra chave new. Também é possível fazer isso com uma Constructor Function.

```
function Pessoa(nome) {  
  if (!(this instanceof Pessoa)) // ou (!new.target)  
    return new Pessoa(nome);  
  this.nome = nome;  
}  
  
Pessoa.prototype.andar = function() {  
  return `${this.nome} andou`;  
}  
  
const designer = Pessoa('André');
```

## Exemplo Real

---

```
function $$(selectedElements) {
  const elements = document.querySelectorAll(selectedElements);

  function on(onEvent, callback) {
    elements.forEach(element => {
      element.addEventListener(onEvent, callback);
    });
    return this; // retornar this irá funcionar da mesma forma
  }

  function hide() {
    elements.forEach(element => {
      element.style.display = 'none';
    });
    return this;
  }

  function show() {
    elements.forEach(element => {
      element.style.display = 'initial';
    });
  }
}
```

{

```
function addClass(className) {  
    elements.forEach(element => {  
        element.classList.add(className);  
    });  
    return this;  
}
```

```
function removeClass(className) {  
    elements.forEach(element => {  
        element.classList.add(className);  
    });  
    return this;  
}
```

```
return Object.freeze({  
    elements,  
    on,  
    hide,  
    show,  
    addClass,  
    removeClass,
```

```
const buttons = $$('button');
buttons.hide().show().addClass('ativo').removeClass('ativo');
```

# MAIS JAVASCRIPT

---

Closures e Debugging

## Escopo

Quando criamos uma função, a mesma possui acesso à todas as variáveis criadas em seu escopo e também ao escopo pai. A mesma coisa acontece para funções dentro de funções.

```
let item1 = 1;
function func1() {
    let item2 = 2;
    function func2() {
        let item3 = 3;
    }
}

// func1, possui acesso à
// item1 e item2

// func2, possui acesso à
// item1, item2 e item3
```

## Clojures

A `funcao2` possui 4 escopos. O primeiro escopo é o Local, com acesso ao `item3`. O segundo escopo dá acesso ao `item2`, esse escopo é chamado de Clojure (`funcao1`) (escopo de função dentro de função). O terceiro escopo é o Script com acesso ao `item1` e o quarto escopo é o Global/Window.

```
let item1 = 1;
function funcao1() {
  let item2 = 2;
  function funcao2() {
    let item3 = 3;
    console.log(item1);
    console.log(item2);
    console.log(item3);
  }
  funcao2();
}
```

## Debugging

É possível "debugarmos" um código JavaScript utilizando ferramentas do browser ou através do próprio Visual Studio Code. Se o código possuir qualquer Web API, o processo deve ser feito no Browser. Plugins podem interferir no debug dentro do browser.

```
debugger; // adicione a palavra debugger
let item1 = 1;
function funcao1() {
  let item2 = 2;
  function funcao2() {
    let item3 = 3;
    console.log(item1);
    console.log(item2);
    console.log(item3);
  }
  funcao2();
}
```

## Caso Clássico

---

Um dos casos mais clássicos para a demonstração de Clojures é através da criação de uma função de incremento. É como se a função incrementar carregasse uma mochila chamada contagem, onde uma referência para as suas variáveis estão contidas na mesma.

```
function contagem() {  
  let total = 0;  
  return function incrementar() {  
    total++;  
    console.log(total);  
  }  
}  
  
const ativarIncrementar = contagem();  
ativarIncrementar(); // 1  
ativarIncrementar(); // 2  
ativarIncrementar(); // 3
```

## Clojures na Real

Todas as funções internas da Factory Function possuem uma closure de `$$`. As mesmas contém uma referência à variável `elements` declarada dentro do escopo da função.

```
function $$(selectedElements) {  
  const elements = document.querySelectorAll(selectedElements);  
  
  function hide() { ... }  
  function show() { ... }  
  function on() { ... }  
  function addClass() { ... }  
  function removeClass() { ... }  
  
  return { hide, show, on, addClass, removeClass }  
}
```

# MAIS JAVASCRIPT

---

Destructuring

## Destructuring

---

Permite a desestruturação de Arrays e Objetos. Atribuindo suas propriedades à novas variáveis.

```
const carro = {  
    marca: 'Fiat',  
    ano: 2018,  
    portas: 4,  
}  
  
const {marca, ano} = carro;  
  
console.log(marca); // Fiat  
console.log(ano); // 2018
```

# Destructuring Objects

A desestruturação irá facilitar a manipulação de dados.  
Principalmente quando temos uma grande profundidade de  
objetos.

```
const cliente = {  
    nome: 'Andre',  
    compras: {  
        digitais: {  
            livros: ['Livro 1', 'Livro 2'],  
            videos: ['Video JS', 'Video HTML']  
        },  
        fisicas: {  
            cadernos: ['Caderno 1']  
        }  
    }  
  
    console.log(cliente.compras.digitais.livros);  
    console.log(cliente.compras.digitais.videos);  
  
    const {livros, videos} = cliente.compras.digitais;
```

```
console.log(videos);
```

# Nesting

É possível aninhar uma desestruturação dentro de outra.

COPiar

```
const cliente = {  
    nome: 'Andre',  
    compras: {  
        digitais: {  
            livros: ['Livro 1', 'Livro 2'],  
            videos: ['Video JS', 'Video HTML']  
        },  
        fisicas: {  
            cadernos: ['Caderno 1']  
        }  
    }  
  
    const {fisicas, digitais, digitais: {livros, videos}} =  
        cliente.compras;  
  
    console.log(fisicas);  
    console.log(livros);
```

```
console.log(digitais);
```

## Nome das Variáveis

---

É necessário indicar o nome da propriedade que você deseja desestruturar de um objeto. É possível mudar o nome da variável final com:

```
const cliente = {  
    nome: 'André',  
    compras: 10,  
}  
  
const {nome, compras} = cliente;  
// ou  
const {nome: nomeCliente, compras: comprasCliente} = cliente;
```

## Valor Inicial

---

Caso a propriedade não exista o valor padrão dela será `undefined`. É possível modificar este valor no momento da desestruturação.

```
const cliente = {  
    nome: 'Andre',  
    compras: 10,  
}  
  
const {nome, compras, email = 'email@gmail.com', cpf} = cliente;  
console.log(email) // email@gmail.com  
console.log(cpf) // undefined
```

## Destructuring Arrays

---

Para desestruturar array's você deve colocar as variáveis entre [ ] colchetes.

```
const frutas = ['Banana', 'Uva', 'Morango'];

const primeiraFruta = frutas[0];
const segundaFruta = frutas[1];
const terceiraFruta = frutas[2];

// Com destructuring
const [primeira, segunda, terceira] = frutas;
```

## Declaração de Variáveis

---

A desestruturação pode servir para declararmos uma sequência de variáveis.

```
const primeiro = 'Item 1';
const segundo = 'Item 2';
const terceiro = 'Item 3';
// ou
const [primeiro, segundo, terceiro] = ['Item 1', 'Item 2',
'Item 3'];
```

## Argumento Desestruturado

Se uma função espera receber como argumento um objeto, podemos desestruturar ele no momento da declaração.

```
function handleKeyboard(event) {  
    console.log(event.key);  
}  
// Com Destructuring  
function handleKeyboard({key}) {  
    console.log(key);  
}  
  
document.addEventListener('keyup', handleKeyboard);
```

# Exercícios

---

```
// Extraia o backgroundColor, color e margin do btn
const btn = document.querySelector('button');
const btnStyles = getComputedStyle(btn);

// Troque os valores das variáveis abaixo
let cursoAtivo = 'JavaScript';
let cursoInativo = 'HTML';

// Corrija o erro abaixo
const cachorro = {
    nome: 'Bob',
    raca: 'Labrador',
    cor: 'Amarelo'
}

const {bobCor: cor} = cachorro;
```

# MAIS JAVASCRIPT

---

Rest e Spread

## Parâmetros

---

Nem todos os parâmetros que definimos são utilizados quando uma função é executada, devido a falta de argumentos. Por isso é importante nos prepararmos para caso estes argumentos não sejam declarados.

```
function perimetroForma(lado, totalLados) {  
    return lado * totalLados;  
}  
  
perimetroForma(10, 4); // 40  
perimetroForma(10); // NaN
```

## Parâmetro Padrão (Default) ES5

Antes do ES6 a forma de definirmos um valor padrão para um parâmetro, era através de uma expressão.

```
function perimetroForma(lado, totalLados) {  
    totalLados = totalLados || 4; // se não for definido, será  
    igual à 4  
    return lado * totalLados;  
}  
  
perimetroForma(10, 3); // 30  
perimetroForma(10); // 40
```

## Parâmetro Padrão (Default) ES6+

---

Com o ES6 é possível definirmos o valor de um parâmetro direto na declaração do mesmo, caso o argumento não seja passado no momento da execução.

```
function perimetroForma(lado, totalLados = 4) {  
    return lado * totalLados;  
}  
  
perimetroForma(10, 5); // 50  
perimetroForma(10); // 40
```

## Arguments

---

A palavra chave `arguments` é um objeto Array-like criado dentro da função. Esse objeto contém os valores dos argumentos.

```
function perimetroForma(lado, totalLados = 4) {  
    console.log(arguments)  
    return lado * totalLados;  
}  
  
perimetroForma(10);  
perimetroForma(10, 4, 20);
```

## Parâmetro Rest

---

É possível declararmos uma parâmetro utilizando `...` na frente do mesmo. Assim todos os argumentos que passarmos na ativação da função, ficarão dentro do parâmetro.

```
function anunciarGanhadores(...ganhadores) {  
    ganhadores.forEach(ganhador => {  
        console.log(ganhador + ' ganhou.')  
    });  
}  
  
anunciarGanhadores('Pedro', 'Marta', 'Maria');
```

## Único Rest

---

Só é possível ter um único parâmetro rest e ele deve ser o último.

```
function anunciarGanhadores(premio, ...ganhadores) {  
    ganhadores.forEach(ganhador => {  
        console.log(ganhador + ' ganhou um ' + premio)  
    });  
}  
  
anunciarGanhadores('Carro', 'Pedro', 'Marta', 'Maria');
```

## Rest vs Arguments

---

Apesar de parecidos o parâmetro rest e a palavra arguments possuem grandes diferenças. Sendo rest uma array real e arguments um objeto Array-like.

```
function anunciarGanhadores(premio, ...ganhadores) {  
    console.log(ganhadores);  
    console.log(arguments);  
}  
  
anunciarGanhadores('Carro', 'Pedro', 'Marta', 'Maria');
```

## Operador Spread

Assim como o rest, o operador Spread também utiliza os `...` para ser ativado. O spread irá distribuir um item iterável, um por um.

```
const frutas = ['Banana', 'Uva', 'Morango'];
const legumes = ['Cenoura', 'Batata'];

const comidas = [...frutas, 'Pizza', ...legumes];
```

## Spread Argument

O Spread pode ser muito útil para funções que recebem uma lista de argumentos ao invés de uma array.

```
const numeroMaximo = Math.max(4,5,20,10,30,2,33,5); // 33

const listaNumeros = [1,13,21,12,55,2,3,43];
const numeroMaximoSpread = Math.max(...listaNumeros);
```

## Transformar em Array

---

É possível transformar itens iteráveis em uma array real com o spread.

```
const btns = document.querySelectorAll('button');
const btnsArray = [...btns];

const frase = 'Isso é JavaScript';
const fraseArray = [...frase];
```

# Exercícios

---

```
// Reescreva a função utilizando
// valores iniciais de parâmetros com ES6
function createButton(background, color) {
    background = background || 'blue';
    if(color === undefined) {
        color = 'red';
    }
    const buttonElement = document.createElement('button');
    buttonElement.style.background = background;
    return buttonElement;
}

const redButton = createButton();

// Utilize o método push para inserir as frutas ao final de
// comidas.
const frutas = ['Banana', 'Uva', 'Morango'];
const comidas = ['Pizza', 'Batata'];
```

# MAIS JAVASCRIPT

---

Loops e Iterable

## Iterable

---

São objetos que possuem o método [Symbol.iterator], geralmente no protótipo, é dentro dele que a função que lida com a iteração será definida. Ex: Array, String, NodeList, boa parte das Array-Like e outros.

```
const frutas = ['Banana', 'Morango', 'Uva'];
const frase = 'Isso é JavaScript';

fetch('https://pokeapi.co/api/v2/pokemon/')
  .then(({headers}) => console.log(headers));
```

## for...of

É possível fazemos um loop por cada iteração do objeto iterável utilizando o for...of. Além deste loop podemos também utilizar o Spread Operator nos mesmos.

```
const frutas = ['Banana', 'Morango', 'Uva'];
const frase = 'Isso é JavaScript';

for(const fruta of frutas) {
    console.log(fruta);
}

for(const char of frase) {
    console.log(char);
}
```

## Spread e for...of

Com o for loop podemos manipular cada um dos elementos do objeto iterável.

```
const buttons = document.querySelectorAll('button');

for(const btn of buttons) {
  btn.style.background = 'blue';
}

console.log(...buttons);
```

## Apenas Iteráveis

---

O `for...of` não irá funcionar em um objeto comum que não seja iterável.

```
const carro = {  
    marca: 'Honda',  
    ano: 2018,  
}  
  
// Erro, não é Iterável  
for(const propriedade of carro) {  
    console.log(propriedade);  
}
```

## for...in

Este loop irá retornar a chave (key) de todas as propriedades enumeráveis (que não sejam símbolos) de um objeto.

```
const carro = {  
    marca: 'Honda',  
    ano: 2018,  
}  
  
for(const propriedade in carro) {  
    console.log(propriedade);  
}
```

## Arrays e for...in

---

Uma Array é um objeto, porém a chave de cada valor é igual ao seu index.

```
const frutas = ['Banana', 'Morango', 'Uva'];

for(const index in frutas) {
    console.log(index);
}

for(const index in frutas) {
    console.log(frutas[index]);
}
```

## Chave e Valor

---

Utilizando o `for...in` podemos retornar todas as chaves e valores de propriedades enumeráveis.

```
const btn = document.querySelector('button');
const btnStyles = getComputedStyle(btn);

for(const style in btnStyles) {
  console.log(` ${style}: ${btnStyles[style]}`);
}
```

## Do / While

Outro tipo de loop é o Do / While. Não é muito utilizado.

```
let i = 0;  
do {  
    console.log(i++)  
} while (i <= 5);
```

## Exercícios

---

```
// Crie 4 li's na página  
// Utilizando o for...of  
// adicione uma classe a cada li  
  
// Utilize o for...in para listar  
// todos as propriedades e valores  
// do objeto window
```