

Jornada de Aprendizagem

Boas-vindas!

Pronto(a) para dar um upgrade nas suas habilidades de programação? O JavaScript ES6+ traz uma série de novidades que vão transformar a forma como você pensa sobre desenvolvimento web. Prepare-se para explorar features

modernas para criar aplicações mais robustas.

Objetivos de aprendizagem

1

Objetivo 1



Identificar os principais métodos auxiliares para arrays em JavaScript e entender como esses métodos ajudam a manipular dados de forma eficiente.

modernas para criar aplicações mais robustas.

Objetivos de aprendizagem

2

Objetivo 2



Demonstrar como utilizar o DOM (Document Object Model) para acessar e modificar elementos de uma página web, criando interações básicas com o usuário.

modernas para criar aplicações mais robustas.

Objetivos de aprendizagem

3

Objetivo 3



Explorar diferentes formas de percorrer coleções, como o laço for...of e forEach, e reconhecer em quais situações cada um deles é mais apropriado.

modernas para criar aplicações mais robustas.

Objetivos de aprendizagem

4

Objetivo 4



Aplicar conceitos básicos de programação assíncrona, como callbacks e promises, para fazer requisições a uma API e exibir os dados obtidos em uma página web.

modernas para criar aplicações mais robustas.

Objetivos de aprendizagem

5

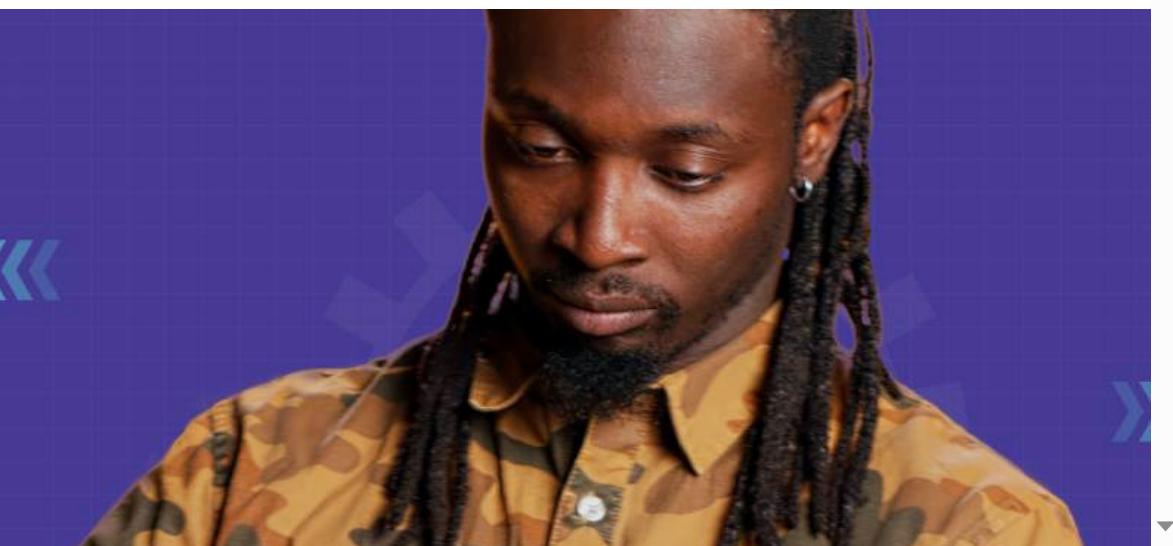
Objetivo 5



Identificar a importância de modularizar o código em JavaScript e reconhecer como a divisão em módulos facilita a organização, manutenção e reutilização de componentes em projetos.

Métodos Auxiliares para Array

Vamos explora métodos auxiliares para Array que foram inicialmente introduzidos no ES5.1 e,



posteriormente, se tornaram parte integral do ES6.



Esses métodos, originalmente implementados em bibliotecas como Lodash e Underscore.js, se tornaram essenciais no desenvolvimento JavaScript devido à sua versatilidade e legibilidade. Dominar esses métodos permite escrever código mais limpo, eficiente e fácil de manter.

Explorando os métodos:

- **forEach**: uma função versátil para percorrer cada elemento de um Array, executando uma ação específica para cada um.
- **map**: transforma cada elemento de um Array em um novo elemento, gerando

um novo Array com as alterações.

- **filter:** cria um novo Array que contém apenas os elementos que atendem a um critério específico.
- **find:** busca um elemento específico em um Array e o retorna caso ele atenda a um critério predefinido.
- **every:** verifica se todos os elementos de um Array atendem a uma condição, retornando true caso todos os elementos a atendam, caso contrário, retorna false.
- **some:** verifica se pelo menos um elemento de um Array atende a uma condição, retornando true se pelo menos um elemento a atende, caso contrário, retorna false.
- **reduce:** combina todos os elementos de um Array em um único valor, aplicando uma função cumulativa a cada um.

forEach: Uma ação para cada elemento

forEach: Uma ação para cada elemento

O método forEach é como um passeio pelo Array. Ele te leva a cada elemento e permite que você faça algo com ele.

Exemplo:

```
const frutas = ['maçã', 'banana', 'uva'];

frutas.forEach(fruta => {
  console.log(`Eu gosto de comer ${fruta}`);
});

// Saída:
// Eu gosto de comer maçã!
// Eu gosto de comer banana!
// Eu gosto de comer uva!
```

map: Transformando Arrays

O método map é como um mágico que transforma cada elemento de um Array em algo novo. Ele cria um novo Array com as transformações.

Exemplo:

```
const numeros = [1, 2, 3, 4, 5];

const numerosDobrados = numeros.map(numero => numero * 2);
console.log(numeros); // [1, 2, 3, 4, 5]
console.log(numerosDobrados); // [2, 4, 6, 8, 10]
```

content_copyUse code with caution.JavaScript

filter: Selecionando o que importa

O método filter é como um guarda que escolhe apenas os elementos que atendem a um requisito. Ele cria um novo Array com os elementos selecionados.

Exemplo:

```
const idades = [10, 18, 15, 22, 16];

const maioresDeldade = idades.filter(idade => idade >= 18);

console.log(maioresDeldade); // [18, 22]
```

find: Encontrando o tesouro

O método `find` é como um detetive que procura um elemento específico dentro de um `Array`. Ele retorna o primeiro elemento que atende a uma condição.

Exemplo:

```
const pessoas = [
  { nome: 'João', idade: 25 },
  { nome: 'Maria', idade: 30 }]
```

```
const pessoas = [  
  { nome: 'João', idade: 25 },  
  { nome: 'Maria', idade: 30 },  
  { nome: 'Pedro', idade: 18 }  
];  
  
const pessoaProcurada = pessoas.find(pessoa => pessoa.nome === 'Maria');  
  
console.log(pessoaProcurada); // { nome: 'Maria', idade: 30 }
```

every: Todos juntos ou nenhum

O método `every` é como um juiz que verifica se todos os elementos de um Array atendem a uma regra. Se todos atenderem, ele declara "todos aprovados!".

Exemplo:

Exemplo:

```
const notas = [8, 9, 10, 7, 8];

const todosAprovados = notas.every(nota => nota >= 7);

console.log(todosAprovados); // true
```

some: Pelo menos um

O método `some` é como um detetive que busca por pelo menos um elemento que atenda a um critério. Se encontrar, ele diz "achei um!".

Exemplo:

```
const produtos = [
  { nome: 'Camiseta', preco: 20 },
```

```
const produtos = [  
  { nome: 'Camiseta', preco: 20 },  
  { nome: 'Calça', preco: 50 },  
  { nome: 'Sapato', preco: 80 }  
];  
  
const temProdutoCaro = produtos.some(produto => produto.preco > 50);  
  
console.log(temProdutoCaro); // true
```

reduce: Combinando elementos

O método reduce é como um mestre artesão que combina todos os elementos de um Array para criar algo novo, usando um processo cumulativo.

Exemplo:

```
const numeros = [1, 2, 3, 4, 5];
const somaTotal = numeros.reduce((acumulador, numero) => acumulador + numero,
0);
console.log(somaTotal); // 15
```

acumulador: é o valor acumulado a cada iteração.

Inicialmente, ele é igual ao valor passado como segundo argumento para reduce (neste caso, 0).



```
const numeros = [1, 2, 3, 4, 5];
const somaTotal = numeros.reduce((acumulador, numero) => acumulador + numero,
0);
console.log(somaTotal); // 15
```



numero: representa o elemento atual do array
durante a iteração.



```
const numeros = [1, 2, 3, 4, 5];
const somaTotal = numeros.reduce((acumulador, numero) => acumulador + numero,
0);
console.log(somaTotal); // 15
```

acumulador + numero: a função passada para reduce está somando o valor acumulado com o valor do elemento atual.



```
const numeros = [1, 2, 3, 4, 5];
const somaTotal = numeros.reduce((acumulador, numero) => acumulador + numero,
0);
console.log(somaTotal); // 15
```

0: este é o valor inicial do acumulador. Se não for fornecido, o reduce usaria o primeiro elemento do array como valor inicial e começaria a acumular a partir do segundo elemento.



Dê o play!

AULA 1 - INTRODUÇÃO



AULA 2 - MÉTODOS AUXILIARES DE ARRAYS



—

Dominar os métodos auxiliares de Array é crucial para qualquer desenvolvedor JavaScript. Eles possibilitam a manipulação de dados de maneira eficiente, concisa e estruturada. Ao aplicá-los em seus projetos, você notará como eles podem simplificar o código e facilitar o processo de desenvolvimento.

CONTINUAR

HORA DA PRÁTICA!

—
Laboratório
de Prática



Olá, Bem-vindo ao Laboratório de Práticas

Nesse espaço você encontrará alguns exercícios para praticar o conteúdo estudado neste curso. Depois de praticar, não esqueça de conferir o gabarito, com a explicação dos exercícios!

Vamos começar?

Atividades

Prepare-se para colocar a mão na massa!

Vamos lá?

[INICIAR >](#)



Exercício 1: Contando Vogais

Crie uma função chamada contarVogais que recebe uma string como argumento e retorna o número de vogais presentes na string. Utilize o método `forEach` para iterar sobre cada caractere da string e o método `some` para verificar se o caractere é uma vogal.



Exemplo:

```
contarVogais("Olá Mundo!"); // Retorna 4
```

Dica: Utilize o método `includes` para verificar se um caractere é uma vogal.

Exercício 2: Filtrando Números Pares

Crie uma função chamada **filtrarPares** que recebe um array de números como argumento e retorna um novo array contendo apenas os números pares. Utilize o método filter para filtrar os números pares.



Exemplo:

```
filtrarPares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]); // Retorna [2, 4, 6, 8, 10]
```

Dica: Utilize o operador % (módulo) para verificar se um número é par.

Exercício 3: Calculando a Média

Crie uma função chamada **calcularMedia** que recebe um array de números como argumento e retorna a média dos números. Utilize o método reduce para calcular a soma dos números e, posteriormente, divida a soma pelo tamanho do array.



Exemplo:

```
calcularMedia([1, 2, 3, 4, 5]); // Retorna 3
```

Dica: Utilize o método length para obter o tamanho do array.

Gabarito - Exercício 1: Contando Vogais

Crie uma função chamada **contarVogais** que recebe uma string como argumento e retorna o número de vogais presentes na string. Utilize o método `forEach` para iterar sobre cada caractere da string e o método `some` para verificar se o caractere é uma vogal.



Exemplo:

```
contarVogais("Olá Mundo!"); // Retorna 4
```

Dica: Utilize o método `includes` para verificar se um caractere é uma vogal.

```
function contarVogais(texto) {  
  let contagem = 0;
```

Dica: Utilize o método includes para verificar se um caractere é uma vogal.

```
function contarVogais(texto) {  
  let contagem = 0;  
  
  const vogais = ['a', 'e', 'i', 'o', 'u'];  
  
  texto.toLowerCase().forEach(caractere => {  
    if (vogais.includes(caractere)) {  
      contagem++;  
    }  
  });  
  
  return contagem;  
}  
  
console.log(contarVogais("Olá Mundo!")); // Retorna 4
```



Gabarito - Exercício 2: Filtrando Números Pares

Crie uma função chamada **filtrarPares** que recebe um array de números como argumento e retorna um novo array contendo apenas os números pares. Utilize o método filter para filtrar os números pares.



Exemplo:

```
filtrarPares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]); // Retorna [2, 4, 6, 8, 10]
```

Dica: Utilize o operador %(módulo) para verificar se um número é par.

```
function filtrarPares(numeros) {  
  return numeros.filter(numero => numero % 2 === 0);  
}
```

Gabarito - Exercício 2: Filtrando Números Pares

Crie uma função chamada **filtrarPares** que recebe um array de números como argumento e retorna um novo array contendo apenas os números pares. Utilize o método filter para filtrar os números pares.

Exemplo:

```
filtrarPares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]); // Retorna [2, 4, 6, 8, 10]
```



Dica: Utilize o operador %(módulo) para verificar se um número é par.

```
function filtrarPares(numeros) {  
  return numeros.filter(numero => numero % 2 === 0);  
}  
  
console.log(filtrarPares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])); // Retorna [2, 4, 6, 8, 10]
```

Gabarito - Exercício 3: Calculando a Média

Crie uma função chamada **calcularMedia** que recebe um array de números como argumento e retorna a média dos números. Utilize o método reduce para calcular a soma dos números e, posteriormente, divida a soma pelo tamanho do array.

Exemplo:

```
calcularMedia([1, 2, 3, 4, 5]); // Retorna 3
```



Dica: Utilize o método length para obter o tamanho do array.

```
function calcularMedia(numeros) {  
  const soma = numeros.reduce((acumulador, numero) => acumulador + numero, 0);  
  return soma / numeros.length;  
}  
  
console.log(calcularMedia([1, 2, 3, 4, 5])); // Retorna 3
```

Crie uma função chamada **calcularMedia** que recebe um array de números como argumento e retorna a média dos números. Utilize o método reduce para calcular a soma dos números e, posteriormente, divida a soma pelo tamanho do array.

Exemplo:

```
calcularMedia([1, 2, 3, 4, 5]); // Retorna 3
```



Dica: Utilize o método length para obter o tamanho do array.

```
function calcularMedia(numeros) {  
  const soma = numeros.reduce((acumulador, numero) => acumulador + numero, 0);  
  return soma / numeros.length;  
}  
  
console.log(calcularMedia([1, 2, 3, 4, 5])); // Retorna 3
```

Até o próximo Laboratório de Prática!



INICIAR NOVAMENTE

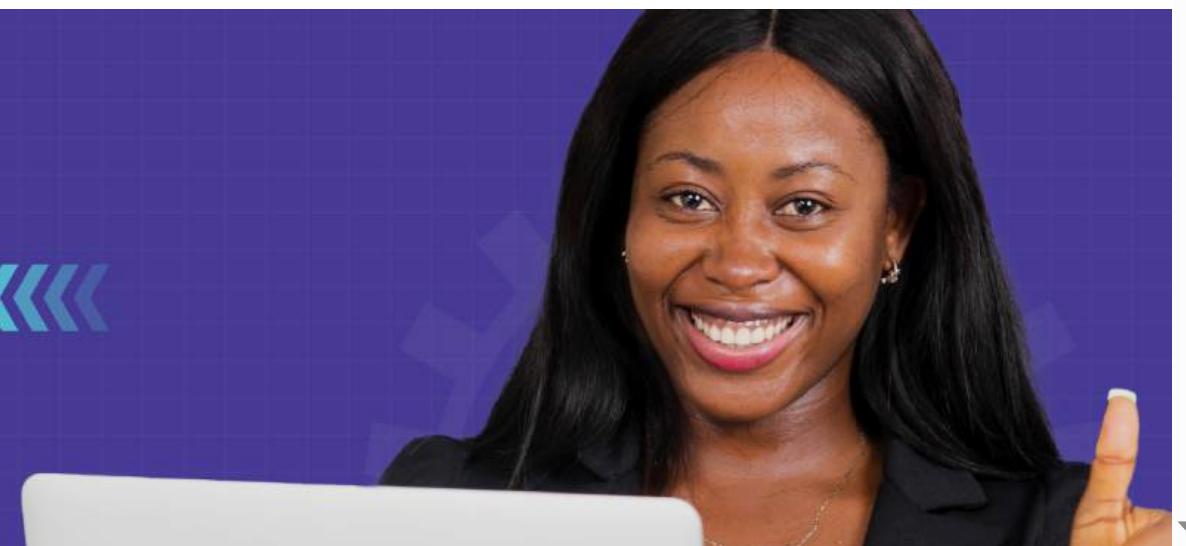


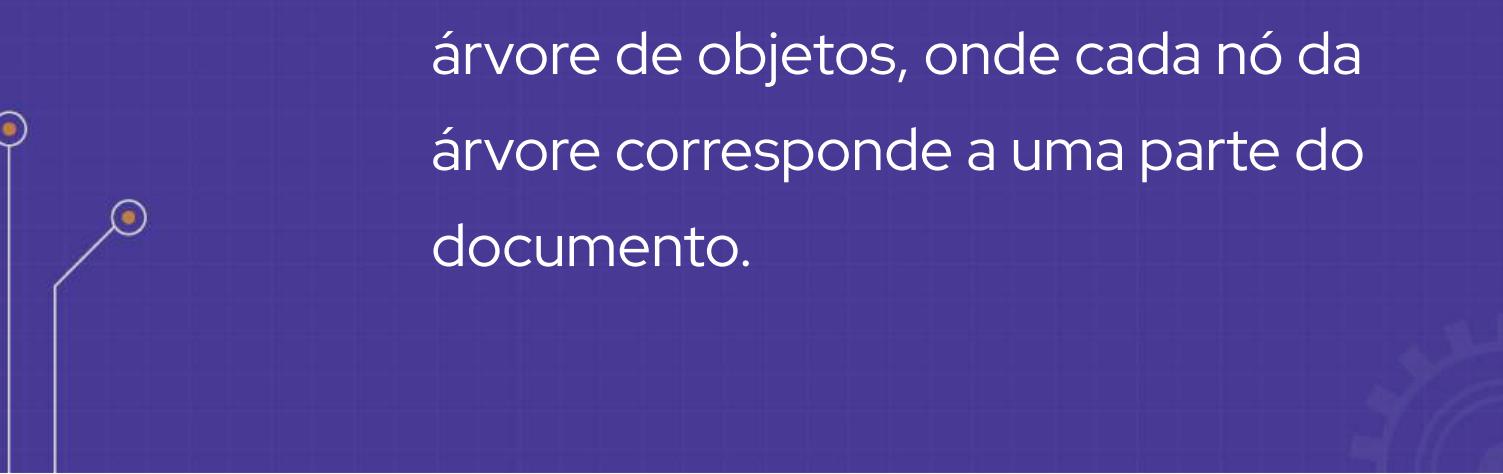
1 2 3

Você finalizou o Laboratório de Prática! Lembre-se! Você
sempre pode voltar para treinar mais um pouco e rever os
gabaritos. Siga adiante!

DOM (Document Object Model)

O **DOM** é uma interface de programação que representa documentos HTML ou XML como uma





árvore de objetos, onde cada nó da árvore corresponde a uma parte do documento.

Essa estrutura permite que linguagens de programação, como JavaScript, interajam e manipulem a estrutura, estilo e conteúdo de páginas web.

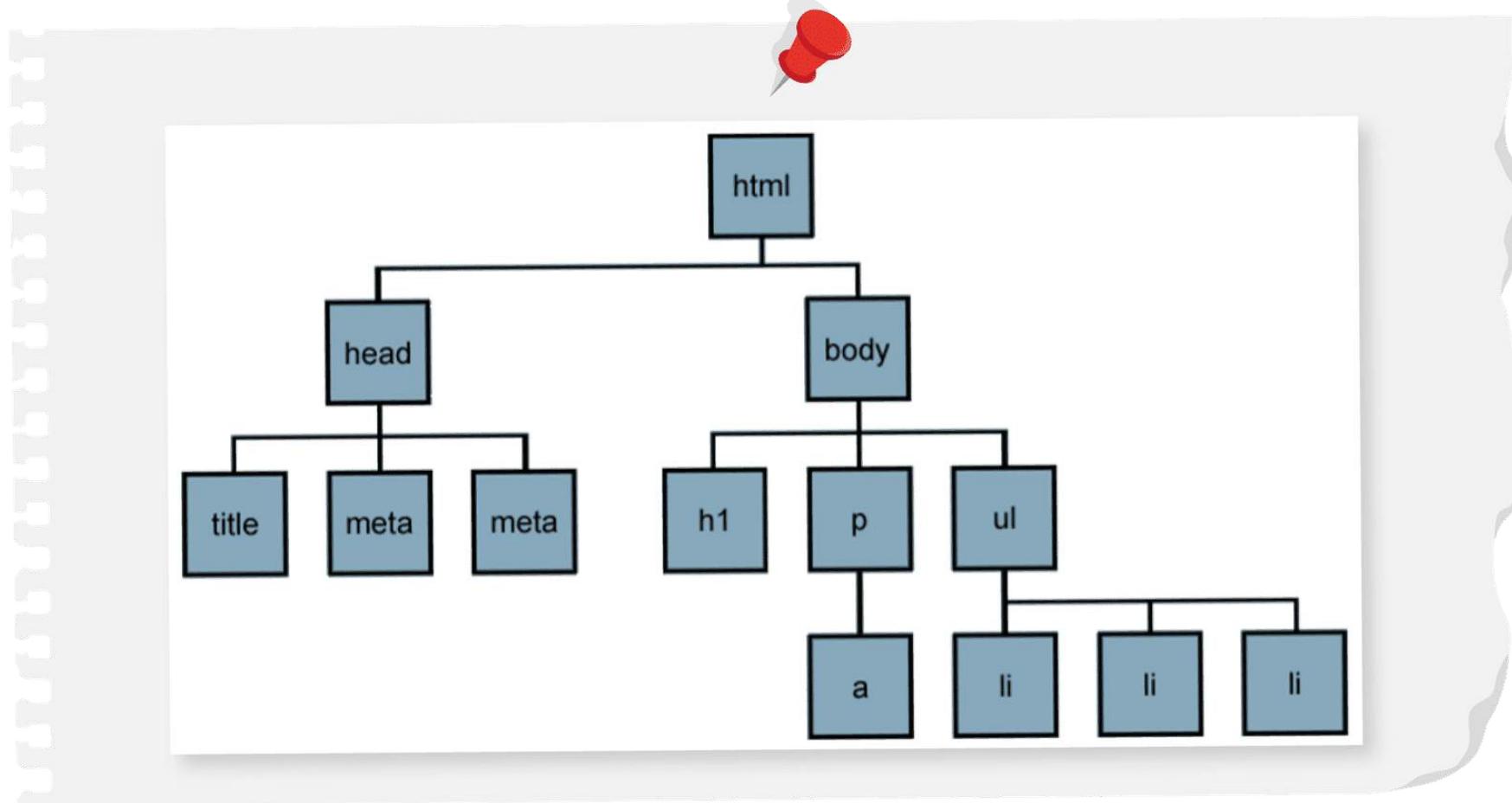
Estrutura do DOM

-

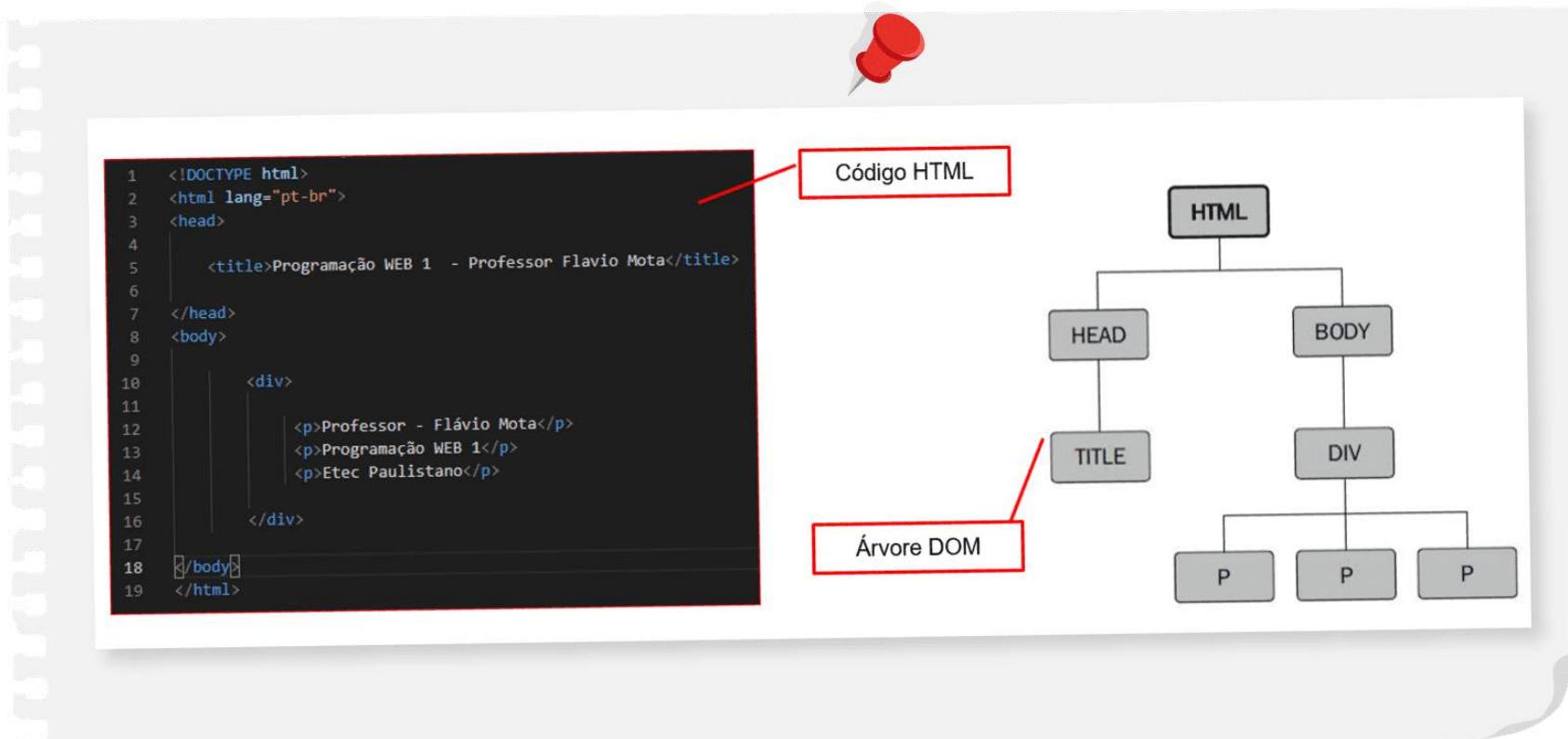
Documento: o ponto de entrada para o conteúdo da página. Em JavaScript, o

- **Documento:** o ponto de entrada para o conteúdo da página. Em JavaScript, o objeto document representa a página inteira.
- **Elementos (Nodes):** cada elemento HTML no DOM é representado como um nó. Por exemplo, uma tag <div> no HTML é um nó no DOM.
- **Atributos:** os atributos de elementos HTML, como class, id, ou src, são acessíveis e manipuláveis através do DOM.
- **Texto:** o conteúdo textual de elementos também é representado como nós de texto.

Árvore DOM



A estrutura do DOM é hierárquica, com um único nó raiz (geralmente o <html>) que contém todos os outros nós como filhos, netos etc. Isso cria uma árvore de elementos que pode ser navegada e modificada usando JavaScript.



Selecionando e Manipulando Elementos DOM com JavaScript

Selecionar e manipular elementos DOM é uma habilidade fundamental para criar páginas web dinâmicas e interativas com JavaScript. Ao dominar essas técnicas, você poderá criar experiências de usuário mais ricas e personalizadas.

Ao selecionar e manipular elementos DOM, você pode:

- **Alterar o conteúdo:** modificar o texto dentro de elementos, como parágrafos ou títulos.
- **Modificar a estrutura:** adicionar, remover ou reordenar elementos HTML.
- **Modificar o estilo:** alterar as propriedades CSS de elementos, como cor, tamanho e posicionamento.
- **Criar interatividade:** responder a ações do usuário, como cliques, e atualizar o conteúdo da página dinamicamente.

Como selecionar elementos DOM?



getElementById: seleciona um único elemento pelo seu ID.



Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo

textContent: altera o texto dentro de um elemento.

innerHTML: altera o HTML dentro de um elemento.

createElement: cria um novo elemento.

Como selecionar elementos DOM?



getElementsByName: seleciona todos os elementos com uma determinada tag.



Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo

textContent: altera o texto dentro de um elemento.

innerHTML: altera o HTML dentro de um elemento.

createElement: cria um novo elemento.

Como selecionar elementos DOM?



getElementsByClassName: seleciona todos os elementos com uma determinada classe.



Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo

textContent: altera o texto dentro de um elemento.

innerHTML: altera o HTML dentro de um elemento.

createElement: cria um novo elemento.

Como selecionar elementos DOM?



querySelector: seleciona o primeiro elemento que corresponde a um seletor CSS.



Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo

textContent: altera o texto dentro de um elemento.

innerHTML: altera o HTML dentro de um elemento.

createElement: cria um novo elemento.

Como selecionar elementos DOM?



querySelectorAll: seleciona todos os elementos que correspondem a um seletor CSS.



Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo

`textContent`: altera o texto dentro de um elemento.

`innerHTML`: altera o HTML dentro de um elemento.

`createElement`: cria um novo elemento.

Uma vez que você selecionou um elemento, pode manipulá-lo de diversas formas:

Alterar o conteúdo	<code>textContent</code> : altera o texto dentro de um elemento. <code>innerHTML</code> : altera o HTML dentro de um elemento.
Modificar a estrutura	<code>createElement</code> : cria um novo elemento. <code>appendChild</code> : adiciona um elemento filho a outro. <code>removeChild</code> : remove um elemento filho.
Modificar o estilo	<code>style</code> : acessa as propriedades CSS inline de um elemento.

Veja a seguir alguns exemplos de seleção e manipulação de elementos com DOM com Javascript.

INICIAR >



```
// 1. document.getElementById(id):  
  
<div id="myDiv">Este é um div com id "myDiv".</div>  
  
<script>  
  
    // Seleciona o elemento com id "myDiv"  
  
    let elemento = document.getElementById("myDiv");  
  
    console.log(elemento.textContent); // Saída: "Este é um div com id 'myDiv'."  
  
</script>  
  
// document.getElementsByClassName(className):  
  
<p class="myClass">Primeiro parágrafo com a classe "myClass".</p>  
  
<p class="myClass">Segundo parágrafo com a classe "myClass".</p>  
  
<script>
```



```
// Seleciona todos os elementos com a classe "myClass"

let elementos = document.getElementsByClassName("myClass");

console.log(elementos.length); // Saída: 2

console.log(elementos[0].textContent); // Saída: "Primeiro parágrafo com a classe
'myClass'."

</script>
```

3. document.getElementsByTagName(tagName):

```
<h1>Título 1</h1>
```

```
<h1>Título 2</h1>
```

```
<h2>Título 3</h2>
```

```
<script>
```

```
// Seleciona todos os elementos <h1>
```

```
<script>

    // Seleciona todos os elementos <h1>

    let titulos = document.getElementsByTagName("h1");

    console.log(titulos.length); // Saída: 2

    console.log(titulos[1].textContent); // Saída: "Título 2"

</script>

4. document.querySelector(selector):

<p class="intro">Este é um parágrafo de introdução.</p>

<p>Este é um parágrafo normal.</p>

<script>

    // Seleciona o primeiro elemento com a classe "intro"

    let paragrafo = document.querySelector(".intro");

    console.log(paragrafo.textContent); // Saída: "Este é um parágrafo de introdução."

</script>
```



```
</script>
```

5. document.querySelectorAll(selector):

```
<ul>
```

```
    <li>Item 1</li>
```

```
    <li>Item 2</li>
```

```
    <li>Item 3</li>
```

```
</ul>
```

```
<script>
```

```
    // Seleciona todos os elementos <li>
```

```
    let itens = document.querySelectorAll("ul li");
```

```
    console.log(itens.length); // Saída: 3
```

```
    console.log(itens[2].textContent); // Saída: "Item 3"
```

```
</script>
```



Manipulando Elementos:

element.innerHTML:

```
<div id="content">Texto original</div>
```

```
<script>
```

```
let div = document.getElementById("content");
```

```
// Modifica o conteúdo HTML
```

```
div.innerHTML = "<strong>Texto atualizado com HTML</strong>";
```

```
</script>
```

element.textContent:

```
<p id="text">Texto original</p>
```

```
<script>
```



```
<script>

    let paragrafo = document.getElementById("text");

    // Modifica o conteúdo de texto

    paragrafo.textContent = "Texto atualizado";

</script>

element.setAttribute(attribute, value):



<script>

    let img = document.getElementById("imagem");

    // Define um novo valor para o atributo 'src'

    img.setAttribute("src", "imagem2.jpg");

</script>

element.getAttribute(attribute):
```



`element.getAttribute(attribute):`

```
<a id="link" href="https://www.example.com">Clique aqui</a>
```

`<script>`

```
let link = document.getElementById("link");
```

```
// Obtém o valor do atributo 'href'
```

```
let href = link.getAttribute("href");
```

```
console.log(href); // Saída: "https://www.example.com"
```

`</script>`

`element.style.propertyName:`

```
<div id="box" style="width: 100px; height: 100px; background-color: red;"></div>
```

`<script>`

```
let box = document.getElementById("box");
```

```
// Modifica a cor de fundo do elemento
```

```
box.style.backgroundColor = "blue";
```



```
// Modifica a cor de fundo do elemento  
box.style.backgroundColor = "blue";  
 </script>
```

1 2 3

Eventos em JavaScript

O que são eventos?

Eventos são ações ou ocorrências que acontecem na página, como cliques de mouse, pressionamento de teclas, carregamento de páginas etc. O JavaScript pode responder a esses eventos utilizando "event listeners" (ouvintes de eventos).

Adicionando e Removendo Event Listeners

Os event listeners são uma ferramenta poderosa para criar páginas web interativas. Ao entender como adicioná-los e removê-los, você poderá construir aplicações web mais dinâmicas e responsivas.

- **element.addEventListener(event, function):** adiciona um ouvinte de eventos a um elemento. Exemplo: button.addEventListener('click', function() { alert('Clicado!'); });
- **element.removeEventListener(event, function):** remove um ouvinte de eventos de um elemento.

Principais Eventos

1

Mouse Events:

- **click:** disparado quando o elemento é clicado.

- **dblclick**: disparado quando o elemento é duplamente clicado.
- **mouseover**: disparado quando o mouse passa sobre um elemento.
- **mouseout**: disparado quando o mouse sai de um elemento.

2

Keyboard Events:

- **keydown**: disparado quando uma tecla é pressionada.
- **keyup**: disparado quando uma tecla é solta.
- **keypress**: disparado quando uma tecla é pressionada e solta.

3

Form Events:

- **submit**: disparado quando um formulário é enviado.
- **change**: disparado quando o valor de um campo de formulário é alterado.

- **focus**: disparado quando um elemento de formulário ganha foco.
- **blur**: disparado quando um elemento de formulário perde o foco.

Manipulando o DOM com Métodos de JavaScript

Criando e Adicionando Elementos:

`document.createElement(tagName)`: cria um novo elemento HTML especificado.



- **focus**: disparado quando um elemento de formulário ganha foco.
- **blur**: disparado quando um elemento de formulário perde o foco.

Manipulando o DOM com Métodos de JavaScript

Criando e Adicionando Elementos:



parentElement.appendChild(childElement):
adiciona um novo nó filho ao final de um nó pai.



- **focus**: disparado quando um elemento de formulário ganha foco.
- **blur**: disparado quando um elemento de formulário perde o foco.

Manipulando o DOM com Métodos de JavaScript

Criando e Adicionando Elementos:

parentElement.insertBefore(newElement, referenceElement): insere um novo elemento antes de um elemento de referência.



Removendo e Substituindo Elementos:



parentElement.removeChild(childElement):

remove um elemento filho de um nó pai.



1

Mouse Events

Conheça a seguir.



Removendo e Substituindo Elementos:



parentElement.replaceChild(newElement, oldElement): substitui um elemento filho por outro.



1

Mouse Events

Conheça a seguir.



Click

```
<!DOCTYPE html>

<html>
  <head>
    <title>Exemplo de Click</title>
  </head>
  <body>
    <button id="myButton">Clique Aqui</button> <p id="result"></p>
    <script>
      // 1. Seleciona o botão e o elemento <p> pelo seu ID.
```



```
const button = document.getElementById("myButton");

const result = document.getElementById("result");

// 2. Adiciona um ouvinte de evento "click" ao botão.

button.addEventListener("click", function() {

    // 3. Quando o botão é clicado, o conteúdo do elemento <p> é atualizado.

    result.textContent = "Você clicou no botão!";

});

</script>

</body>

</html>
```

Dblclick

```
<!DOCTYPE html>

<html>
    <head>
        <title>Exemplo de Dblclick</title>
    </head>
    <body>
        <div id="myDiv">Clique duas vezes aqui</div>
        <p id="result"></p>
        <script>
```



```
<script>

    // 1. Seleciona o div e o elemento <p> pelo seu ID.

    const myDiv = document.getElementById("myDiv");

    const result = document.getElementById("result");

    // 2. Adiciona um ouvinte de evento "dblclick" ao div.

    myDiv.addEventListener("dblclick", function() {

        // 3. Quando o div é clicado duas vezes, o conteúdo do elemento <p> é atualizado.

        result.textContent = "Você clicou duas vezes!";

    });

</script>

</body>

</html>
```

Mouseover e Mouseout

```
<!DOCTYPE html>

<html>
    <head>
        <title>Exemplo de Mouseover e Mouseout</title>
    </head>
    <body>
        <div id="myDiv" style="width: 200px; height: 100px; background-color: lightgray;">Passe o mouse aqui</div>
        <p id="result"></p>
        <script>
```



```
<script>

    // 1. Seleciona o div e o elemento <p> pelo seu ID.

    const myDiv = document.getElementById("myDiv");

    const result = document.getElementById("result");

    // 2. Adiciona ouvintes de evento "mouseover" e "mouseout" ao div.

    myDiv.addEventListener("mouseover", function() {

        // 3. Quando o mouse entra no div, o conteúdo do elemento <p> é atualizado.

        result.textContent = "Mouse sobre o div";

    });

    myDiv.addEventListener("mouseout", function() {

        // 4. Quando o mouse sai do div, o conteúdo do elemento <p> é atualizado.

        result.textContent = "Mouse fora do div";

    });

</script>
```



```
myDiv.addEventListener("mouseover", function() {  
    // 3. Quando o mouse entra no div, o conteúdo do elemento <p> é atualizado.  
  
    result.textContent = "Mouse sobre o div";  
});  
  
myDiv.addEventListener("mouseout", function() {  
    // 4. Quando o mouse sai do div, o conteúdo do elemento <p> é atualizado.  
  
    result.textContent = "Mouse fora do div";  
});  
  
</script>  
  
</body>  
  
</html>
```

Keyboard Events

Conheça a seguir.

1 2 3 4 5 6 7 8 9 10 11 12



Keydown

```
<!DOCTYPE html>

<html>
  <head>
    <title>Exemplo de Keydown</title>
  </head>
  <body>
    <input type="text" id="myInput" placeholder="Pressione uma tecla">
    <p id="result"></p>
    <script>
```



```
<script>

    // 1. Seleciona o input e o elemento <p> pelo seu ID.

    const myInput = document.getElementById("myInput");

    const result = document.getElementById("result");

    // 2. Adiciona um ouvinte de evento "keydown" ao input.

    myInput.addEventListener("keydown", function(event) {

        // 3. Quando uma tecla é pressionada, o conteúdo do elemento <p> é atualizado.

        // A propriedade `event.key` retorna a tecla que foi pressionada.

        result.textContent = `Você pressionou a tecla: ${event.key}`;

    });

</script>

</body>

</html>
```



Keyup

```
<!DOCTYPE html>

<html>

<head>

<title>Exemplo de Keyup</title>

</head>

<body>

<input type="text" id="myInput" placeholder="Pressione e solte uma tecla">

<p id="result"></p>

<script>
```



```
<script>

    // 1. Seleciona o input e o elemento <p> pelo seu ID.

    const myInput = document.getElementById("myInput");

    const result = document.getElementById("result");

    // 2. Adiciona um ouvinte de evento "keyup" ao input.

    myInput.addEventListener("keyup", function(event) {

        // 3. Quando uma tecla é solta, o conteúdo do elemento <p> é atualizado.

        // A propriedade `event.key` retorna a tecla que foi solta.

        result.textContent = `Você soltou a tecla: ${event.key}`;

    });

</script>

</body>

</html>
```



Keypress

```
<!DOCTYPE html>

<html>

<head>

<title>Exemplo de Keypress</title>

</head>

<body>

<input type="text" id="myInput" placeholder="Pressione uma tecla">

<p id="result"></p>

<script>
```



```
<script>

    // 1. Seleciona o input e o elemento <p> pelo seu ID.

    const myInput = document.getElementById("myInput");

    const result = document.getElementById("result");

    // 2. Adiciona um ouvinte de evento "keypress" ao input.

    myInput.addEventListener("keypress", function(event) {

        // 3. Quando uma tecla é pressionada e solta, o conteúdo do elemento <p> é
        // atualizado.

        // A propriedade `event.key` retorna a tecla que foi pressionada e solta.

        result.textContent = `Você pressionou e soltou a tecla: ${event.key}`;

    });

</script>

</body>

</html>
```



Form Events

Conheça a seguir.

1 2 3 4 5 6 7 8 9 10 11 12



Submit

```
<!DOCTYPE html>

<html>
  <head>
    <title>Exemplo de Submit</title>
  </head>
  <body>
    <form id="myForm">
      <label for="name">Nome:</label>
      <input type="text" id="name" name="name"><br><br>
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```



```
<input type="submit" value="Enviar">

</form>

<p id="result"></p>

<script>

// 1. Seleciona o formulário e o elemento <p> pelo seu ID.

const form = document.getElementById("myForm");

const result = document.getElementById("result");

// 2. Adiciona um ouvinte de evento "submit" ao formulário.

form.addEventListener("submit", function(event) {

    // 3. Quando o formulário é enviado, o evento padrão é prevenido (impedindo o
envio para o servidor).

    event.preventDefault();

    // 4. O conteúdo do elemento <p> é atualizado com os dados do formulário.

    result.textContent = `Formulário enviado! Nome:`;
})
```



```
// 2. Adiciona um ouvinte de evento "submit" ao formulário.  
  
form.addEventListener("submit", function(event) {  
  
    // 3. Quando o formulário é enviado, o evento padrão é prevenido (impedindo o  
    envio para o servidor).  
  
    event.preventDefault();  
  
    // 4. O conteúdo do elemento <p> é atualizado com os dados do formulário.  
  
    result.textContent = `Formulário enviado! Nome:  
    ${document.getElementById("name").value}`;  
  
});  
  
</script>  
  
</body>  
  
</html>
```



Change

```
<!DOCTYPE html>

<html>
  <head>
    <title>Exemplo de Change</title>
  </head>
  <body>
    <label for="select">Selecione uma opção:</label>
    <select id="select">
      <option value="opcao1">Opção 1</option>
      <option value="opcao2">Opção 2</option>
      <option value="opcao3">Opção 3</option>
    </select>
  </body>
</html>
```



```
<option value="opcao2">Opção 2</option>

</select>

<p id="result"></p>

<script>

    // 1. Seleciona o elemento <select> e o elemento <p> pelo seu ID.

    const select = document.getElementById("select");

    const result = document.getElementById("result");

    // 2. Adiciona um ouvinte de evento "change" ao elemento <select>.

    select.addEventListener("change", function() {

        // 3. Quando o valor do elemento <select> é alterado, o conteúdo do elemento <p>
        // é atualizado.

        result.textContent = `Você selecionou: ${select.value}`;

    });

</script>
```



```
const select = document.getElementById("select");

const result = document.getElementById("result");

// 2. Adiciona um ouvinte de evento "change" ao elemento <select>.

select.addEventListener("change", function() {

    // 3. Quando o valor do elemento <select> é alterado, o conteúdo do elemento <p>
    // é atualizado.

    result.textContent = `Você selecionou: ${select.value}`;

});

</script>

</body>

</html>
```

Focus e Blur

```
<!DOCTYPE html>

<html>

<head>

<title>Exemplo de Focus e Blur</title>

</head>

<body>

<input type="text" id="myInput" placeholder="Digite algo">

<p id="result"></p>

<script>
```





```
<script>

    // 1. Seleciona o input e o elemento <p> pelo seu ID.

    const myInput = document.getElementById("myInput");

    const result = document.getElementById("result");

    // 2. Adiciona ouvintes de evento "focus" e "blur" ao input.

    myInput.addEventListener("focus", function() {

        // 3. Quando o input recebe foco, o conteúdo do elemento <p> é atualizado.

        result.textContent = "Campo de texto em foco";

    });

    myInput.addEventListener("blur", function() {

        // 4. Quando o input perde foco, o conteúdo do elemento <p> é atualizado.

        result.textContent = "Campo de texto sem foco";

    });

</script>
```



```
// 2. Adiciona ouvintes de evento "focus" e "blur" ao input.  
  
myInput.addEventListener("focus", function() {  
  
    // 3. Quando o input recebe foco, o conteúdo do elemento <p> é atualizado.  
  
    result.textContent = "Campo de texto em foco";  
  
});  
  
myInput.addEventListener("blur", function() {  
  
    // 4. Quando o input perde foco, o conteúdo do elemento <p> é atualizado.  
  
    result.textContent = "Campo de texto sem foco";  
  
});  
  
</script>  
  
</body>  
  
</html>
```

Manipulando o DOM com Métodos de JavaScript

Criando e Adicionando Elementos

`document.createElement(tagName):`

```
<div id="container"></div>

<script>
    // Cria um novo elemento <p>
    let novoParagrafo = document.createElement("p");

    // Define o conteúdo de texto do novo parágrafo
    novoParagrafo.textContent = "Este é um novo parágrafo criado dinamicamente.';

    // Exibe o elemento no console para verificar
    console.log(novoParagrafo);

</script>
```

Esse exemplo cria um novo elemento de parágrafo (<p>) usando document.createElement("p"). O conteúdo do parágrafo é definido usando textContent, mas ele ainda não foi adicionado ao DOM.

parentElement.appendChild(childElement):

```
<div id="container"></div>

<script>
    // Cria um novo elemento <p>
    let novoParagrafo = document.createElement("p");
    novoParagrafo.textContent = "Este parágrafo foi adicionado ao final do container.";

    // Seleciona o elemento pai (container)
    let container = document.getElementById("container");

    // Adiciona o novo parágrafo ao final do container
    container.appendChild(novoParagrafo);
```

parentElement.appendChild(childElement):

```
<div id="container"></div>

<script>
    // Cria um novo elemento <p>
    let novoParagrafo = document.createElement("p");
    novoParagrafo.textContent = "Este parágrafo foi adicionado ao final do container.";

    // Seleciona o elemento pai (container)
    let container = document.getElementById("container");

    // Adiciona o novo parágrafo ao final do container
    container.appendChild(novoParagrafo);

</script>
```

Aqui, o novo parágrafo criado é adicionado ao final do elemento com o id "container" usando appendChild. O parágrafo agora faz parte do DOM e é exibido na página.

parentElement.insertBefore(newElement, referenceElement):

```
<ul id="lista">
  <li>Item 1</li>
  <li>Item 3</li>
</ul>

<script>
  // Cria um novo item de lista <li>
  let novoItem = document.createElement("li");
  novoItem.textContent = "Item 2";

  // Seleciona o elemento pai (lista)
  let lista = document.getElementById("lista");
  lista.appendChild(novoItem);
</script>
```

```
// Seleciona o elemento de referência (Item 3)
let item3 = lista.getElementsByTagName("li")[1];

// Insere o novo item antes do Item 3
lista.insertBefore(novoItem, item3);

</script>
```

Nesse exemplo, um novo item de lista () é criado e inserido antes de um item de referência existente (Item 3) na lista. O novo item é adicionado em uma posição específica dentro do DOM.

Removendo e Substituindo Elementos

parentElement.removeChild(childElement):

```
<ul id="lista">
```

```
<ul id="lista">  
    <li>Item 1</li>  
    <li id="item2">Item 2</li>  
    <li>Item 3</li>  
</ul>  
  
<script>  
    // Seleciona o elemento pai (lista)  
    let lista = document.getElementById("lista");  
  
    // Seleciona o elemento filho a ser removido (Item 2)  
    let item2 = document.getElementById("item2");  
  
    // Remove o Item 2 da lista  
    lista.removeChild(item2);  
  
</script>
```

Aqui, o item da lista com o id "item2" é removido da lista pai usando removeChild. Após a

Aqui, o item de lista com o id "item2" é removido da lista pai usando `removeChild`. Após a remoção, o item não faz mais parte do DOM e não será exibido na página.

parentElement.replaceChild(newElement, oldElement):

```
<div id="container">  
  <p id="paragrafoOriginal">Este é o parágrafo original.</p>  
</div>  
  
<script>  
  // Cria um novo parágrafo <p>  
  let novoParagrafo = document.createElement("p");  
  novoParagrafo.textContent = "Este é o novo parágrafo que substitui o original."  
  
  // Seleciona o elemento pai (container)  
  let container = document.getElementById("container");  
  
  // Seleciona o parágrafo original a ser substituído  
  let paragrafoOriginal = container.querySelector("#paragrafoOriginal");  
  container.replaceChild(novoParagrafo, paragrafoOriginal);  
</script>
```

```
// Seleciona o parágrafo original a ser substituído  
let paragrafoOriginal = document.getElementById("paragrafoOriginal");  
  
// Substitui o parágrafo original pelo novo parágrafo  
container.replaceChild(novoParagrafo, paragrafoOriginal);  
  
</script>
```



Dê o play!

AULA 3 - DOM



AULA 4 - EVENTOS



PROPOSTA DE EXERCÍCIO



HORA DA PRÁTICA!

—
Laboratório
de Prática



Olá, Bem-vindo ao Laboratório de Práticas

Nesse espaço você encontrará alguns exercícios para praticar o conteúdo estudado neste curso. Depois de praticar, não esqueça de conferir o gabarito, com a explicação dos exercícios!

Vamos começar?

Atividades

Prepare-se para colocar a mão na massa!

Vamos lá?

[INICIAR >](#)



Exercício 1: Trocando o Conteúdo de um Botão

Objetivo: criar um botão que, ao ser clicado, troca seu texto entre "Mostrar Detalhes" e "Ocultar Detalhes".

1 2 3 ✓



COMPLETE PARA PROSSEGUIR

Exercício 2: Contando Cliques em um Botão

Objetivo: criar um botão que exiba a quantidade de vezes que foi clicado em um parágrafo.

1 **2** 3 ✓



COMPLETE PARA PROSSEGUIR

Exercício 3: Alterando a Cor de Fundo de um Elemento

Objetivo: criar um elemento (por exemplo, uma div) que muda a cor de fundo quando o mouse passa sobre ele.

1 2 3 ✓



COMPLETE PARA PROSSEGUIR

Gabarito

Agora chegou a hora de conferir!

INICIAR >



Gabarito - Exercício 1: Trocando o Conteúdo de um Botão

Objetivo: criar um botão que, ao ser clicado, troca seu texto entre "Mostrar Detalhes" e "Ocultar Detalhes".

HTML:

```
<!DOCTYPE html>

<html>

<head>

<title>Exercício 1</title>

</head>

<body>
```

```
<body>

    <button id="myButton">Mostrar Detalhes</button>

    <p id="details" style="display: none;">Detalhes ocultos.</p>

    <script src="script.js"></script>

</body>

</html>
```



JavaScript (script.js):

```
const button = document.getElementById("myButton");

const details = document.getElementById("details");

button.addEventListener("click", function() {

    if (button.textContent === "Mostrar Detalhes") {
```

```
if (button.textContent === "Mostrar Detalhes") {  
  
    button.textContent = "Ocultar Detalhes";  
  
    details.style.display = "block"; // Mostra o parágrafo com detalhes  
  
} else {  
  
    button.textContent = "Mostrar Detalhes";  
  
    details.style.display = "none"; // Oculta o parágrafo com detalhes  
  
}  
  
});
```



Explicação:

- 1. Seleção de elementos:** o código seleciona o botão e o parágrafo com detalhes usando seus IDs.
- 2. Adicionando evento:** um ouvinte de eventos "click" é adicionado ao botão.

2. **Adicionando evento:** um ouvinte de eventos "click" é adicionado ao botão.

3. **Verificando o texto do botão:** quando o botão é clicado, o código verifica qual o texto atual do botão ("Mostrar Detalhes" ou "Ocultar Detalhes").

4. **Trocando texto e exibindo/ocultando detalhes:** se o texto for "Mostrar Detalhes", ele é trocado para "Ocultar Detalhes", e o parágrafo com detalhes é exibido. Caso contrário, o texto do botão volta para "Mostrar Detalhes" e o parágrafo é ocultado.

1 2 3 ✓

Gabarito - Exercício 2: Contando Cliques em um Botão

Objetivo: criar um botão que exiba a quantidade de vezes que foi clicado em um parágrafo.

HTML:

```
<!DOCTYPE html>

<html>

<head>

<title>Exercício 2</title>

</head>

<body>
```

```
<body>
```

```
    <button id="clickButton">Clique aqui</button>
```

```
    <p id="clickCount">Contagem de cliques: 0</p>
```

```
    <script src="script.js"></script>
```

```
  </body>
```

```
</html>
```



JavaScript (script.js):

```
const clickButton = document.getElementById("clickButton");
```

```
const clickCount = document.getElementById("clickCount");
```

```
let count = 0; // Variável para armazenar a contagem de cliques
```

```
let count = 0; // Variável para armazenar a contagem de cliques
```

```
clickButton.addEventListener("click", function() {  
    count++; // Incrementa a contagem a cada clique  
    clickCount.textContent = `Contagem de cliques: ${count}`; // Atualiza o texto do parágrafo  
});
```



Explicação:

1. **Seleção de elementos:** o código seleciona o botão e o parágrafo usando seus IDs.

2. **Variável para contagem:** uma variável count é criada para armazenar o número de cliques, iniciando em 0.

3. **Adicionando evento:** um ouvinte de eventos "click" é adicionado ao botão.

```
});
```

Explicação:

- 1. Seleção de elementos:** o código seleciona o botão e o parágrafo usando seus IDs.
- 2. Variável para contagem:** uma variável count é criada para armazenar o número de cliques, iniciando em 0.
- 3. Adicionando evento:** um ouvinte de eventos "click" é adicionado ao botão.
- 4. Atualizando contagem:** quando o botão é clicado, a variável count é incrementada e o texto do parágrafo é atualizado com a nova contagem.

Gabarito - Exercício 3: Alterando a Cor de Fundo de um Elemento

Objetivo: criar um elemento (por exemplo, uma div) que muda a cor de fundo quando o mouse passa sobre ele.

HTML:

```
<!DOCTYPE html>

<html>

<head>

<title>Exercício 3</title>

</head>

<body>
```

```
<body>

<div id="myDiv" style="width: 200px; height: 100px; background-color:
lightblue;">Passe o mouse aqui</div>

<script src="script.js"></script>

</body>

</html>
```



JavaScript (script.js):

```
const myDiv = document.getElementById("myDiv");

myDiv.addEventListener("mouseover", function() {

    myDiv.style.backgroundColor = "yellow"; // Muda a cor de fundo para amarelo

});
```

```
});
```

```
myDiv.addEventListener("mouseout", function() {  
    myDiv.style.backgroundColor = "lightblue"; // Volta para a cor original  
});
```



Explicação:

- 1. Seleção de elemento:** o código seleciona a div usando seu ID.
- 2. Adicionando eventos:** ouvintes de eventos "mouseover" e "mouseout" são adicionados à div.
- 3. Alterando cor:** quando o mouse entra na div ("mouseover"), a cor de fundo é alterada para amarelo. Quando o mouse sai ("mouseout"), a cor de fundo volta para a cor original.

Você finalizou o Laboratório de Prática! Lembre-se! Você
sempre pode voltar para treinar mais um pouco e rever os
gabaritos. Siga adiante!

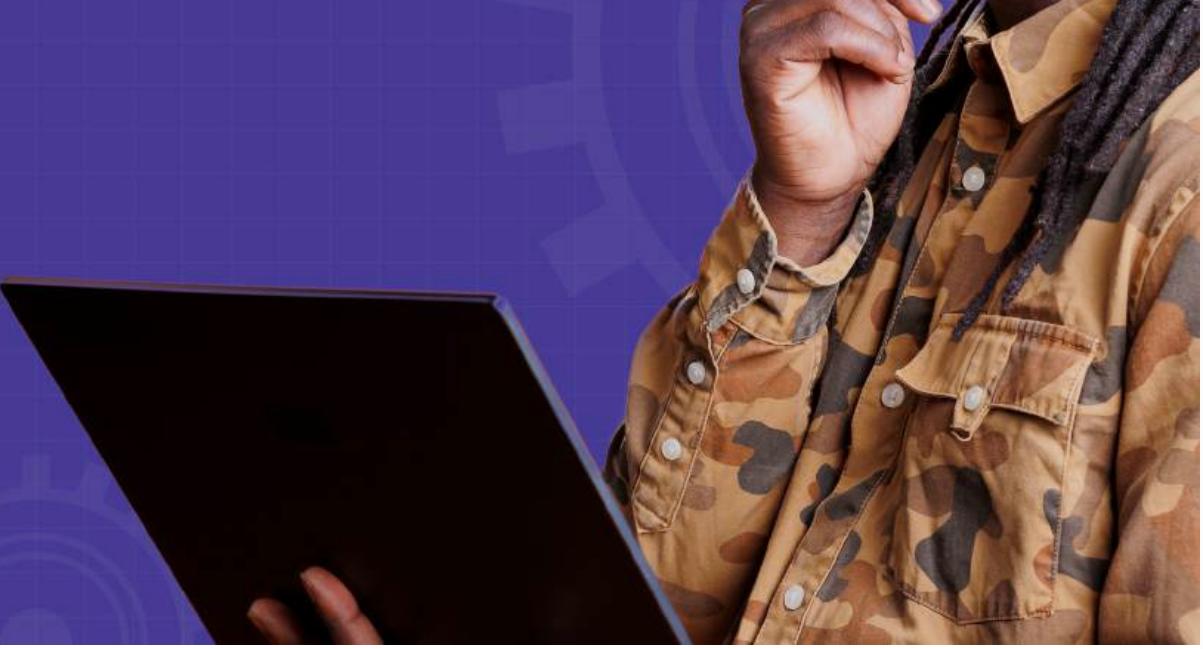
CONTINUAR

Explorando o Laço for...of

Imagine um guia turístico que leva você a uma jornada fascinante por uma cidade, mostrando suas ruas, casas e



seus habitantes. No mundo da programação, o laço for...of é como esse guia, conduzindo você por meio de coleções de dados, revelando cada um de seus elementos.



O Que é o Laço for...of?

O laço for...of é uma ferramenta poderosa para percorrer iteráveis, como Arrays, Sets e Maps. Ele nos permite acessar cada elemento da coleção de forma organizada, realizando ações com cada um deles.

Sintaxe:

```
for (variavel of iteravel) {  
    // Aqui você coloca o código que será executado para cada item da coleção  
}
```

Exemplo Prático 1: Brinquedos Favoritos

Imagine que você tem uma lista de seus brinquedos favoritos:

```
var brinquedos = ['carrinho', 'boneca', 'bola', 'blocos'];  
  
for (var brinquedo of brinquedos) {  
    console.log("Olha, encontrei um " + brinquedo + "!");  
}
```

Ao executar esse código, você verá no console:

Olha, encontrei um carrinho!

Olha, encontrei um boneca!

Olha, encontrei um bola!

Olha, encontrei um blocos!

Entendendo o funcionamento:

- **for (var brinquedo of brinquedos):** aqui, você define a variável brinquedo para armazenar o valor de cada item da

coleção brinquedos. A cada iteração do laço, brinquedo assume o valor do próximo elemento da coleção.

- **console.log("Olha, encontrei um " + brinquedo + "!");**: o código dentro do laço é executado para cada elemento da coleção. Nesse caso, ele imprime uma mensagem com o nome do brinquedo.



Comparando com o Laço for...in: O laço for...in também é usado para percorrer coleções, mas sua função é diferente. Ele é ideal para iterar sobre as chaves de um objeto, enquanto o for...of percorre os valores dentro de um iterável.

Exemplo Prático 2: Dados do Amigo

Imagine que você tem um objeto com informações sobre um amigo:

```
var amigo = {  
    nome: 'João',  
    idade: 25,  
    cidade: 'São Paulo'  
};  
for (var chave in amigo) {  
    console.log("A " + chave + " do meu amigo é: " + amigo[chave]);  
}  
content_copyUse code with caution.JavaScript
```

A saída no console será:

A nome do meu amigo é: João

A idade do meu amigo é: 25

A cidade do meu amigo é: São Paulo

content_copyUse code with caution.

break e continue - Controlando o Passeio:

- **break:** você pode interromper o laço for...of no meio do caminho.
- **continue:** você pode pular para o próximo item da coleção, ignorando o item

- **continue:** você pode pular para o próximo item da coleção, ignorando o item atual.

Exemplo Prático 3: Brinquedos que Começam com "B"

Você só quer ver os brinquedos que começam com a letra "b":

```
var brinquedos = ['carrinho', 'boneca', 'bola', 'blocos'];
for (var brinquedo of brinquedos) {
  if (brinquedo.startsWith('b')) {
    console.log("Que legal! Um " + brinquedo + "!");
  } else {
    continue; // Pula para o próximo brinquedo
  }
}
```

A saída no console será:

Que legal! Um bola!

Que legal! Um blocos!

content_copyUse code with caution.

Conclusão

O laço `for...of` é uma ferramenta poderosa para trabalhar com coleções no JavaScript, permitindo que você explore seus elementos de forma organizada e eficiente.

Experimente-o em seus projetos e descubra novas formas de manipular seus dados!

Explorando Mapas e WeakMaps

Mapas

Imagine um mapa da sua cidade. Cada rua é identificada por um nome (a chave) e tem um endereço (o valor). No JavaScript, o Map funciona da mesma forma.

Exemplo:

```
var familia = new Map();

familia.set('João', 'Rua A, 123');
familia.set('Maria', 'Rua B, 456');
familia.set('Pedro', 'Rua C, 789');

console.log(familia.get('João')) // Saída: Rua A, 123
```

Exemplo 2: Cadastro de Alunos

Imagine que você precisa criar um sistema para cadastrar alunos em um curso. Cada aluno tem um nome, uma idade e um curso. Podemos usar um Map para armazenar essas informações de forma organizada.

Criando o Map:

```
const alunos = new Map();  
content_copyUse code with caution.JavaScript
```

Adicionando Alunos:

```
alunos.set('João', { idade: 20, curso: 'Engenharia' });  
alunos.set('Maria', { idade: 22, curso: 'Medicina' });  
alunos.set('Pedro', { idade: 19, curso: 'Ciência da Computação' });  
content_copyUse code with caution.JavaScript
```

Buscando Informações:

```
console.log(alunos.get('João')) // Saída: { idade: 20, curso: 'Engenharia' }  
content_copyUse code with caution.JavaScript
```

Iterando sobre os Alunos:

```
for (const [nome, dados] of alunos) {  
    console.log(`Aluno: ${nome}`);  
    console.log(`Idade: ${dados.idade}`);  
    console.log(`Curso: ${dados.curso}`);  
    console.log('---');  
}
```

Saída:

Aluno: João

Idade: 20

Saída:

Aluno: João

Idade: 20

Curso: Engenharia

Aluno: Maria

Idade: 22

Curso: Medicina

Aluno: Pedro

Idade: 19

Curso: Ciência da Computação

WeakMaps: Guardando Informações Privadas



São como mapas, mas as chaves só podem ser objetos.



Vantagens: ideal para guardar informações confidenciais sobre objetos.

Exemplo:

```
var gato = { nome: 'Miau' };
var cachorro = { nome: 'Auau' };
```



A existência da chave depende da referência ao objeto.



Vantagens: ideal para guardar informações confidenciais sobre objetos.

Exemplo:

```
var gato = { nome: 'Miau' };
var cachorro = { nome: 'Auau' };
```

Exemplo:

```
var gato = { nome: 'Miau' };
var cachorro = { nome: 'Auau' };

var animaisFavoritos = new WeakMap();
animaisFavoritos.set(gato, 'peixe');
animaisFavoritos.set(cachorro, 'ossos');

console.log(animaisFavoritos.get(gato)); // Saída: peixe
```

Exemplo 2: Dados sensíveis de Usuários

Imagine que você está desenvolvendo um site com um sistema de login. Cada usuário possui dados confidenciais como senha e endereço. É crucial que esses dados sejam protegidos e acessíveis apenas para o sistema interno. Um WeakMap é a ferramenta ideal para esse tipo de situação.

Criando o WeakMap:

```
const dadosSensiveis = new WeakMap();  
content_copyUse code with caution.Ja
```

Adicionando Dados:

```
const usuario1 = { nome: 'Ana' };  
const usuario2 = { nome: 'Bruno' };  
  
dadosSensiveis.set(usuario1, { senha: 'senha123', endereco: 'Rua A, 123' });  
dadosSensiveis.set(usuario2, { senha: 'senha456', endereco: 'Rua B, 456' });
```

```
content_copyUse code with caution.Ja
```

Acessando Dados:

Acessando Dados:

```
console.log(dadosSensiveis.get(usuario1)); // Saída: { senha: 'senha123', endereço: 'Rua  
A, 123' }  
content_copyUse code with caution.Ja
```

Iterando sobre os Dados (não possível):

O WeakMap não permite iteração direta sobre seus elementos. Isso garante a segurança das informações confidenciais. Se você tentar iterar, receberá um erro.

Conclusão:

- O Map é ideal para guardar e organizar informações, permitindo acesso rápido e fácil por meio de suas chaves.
- O WeakMap é a escolha ideal para armazenar dados confidenciais que devem

ser protegidos e acessíveis apenas internamente.



Atenção!

Os exemplos são simplificados para fins didáticos. Na prática, o uso de Map e WeakMap pode ser mais complexo, dependendo da aplicação.



Dê o play!

AULA 5 - FOR OF, MAP E WEAKMAP



HORA DA PRÁTICA!

—
Laboratório
de Prática



Olá, Bem-vindo ao Laboratório de Práticas

Nesse espaço você encontrará alguns exercícios para praticar o conteúdo estudado neste curso. Depois de praticar, não esqueça de conferir o gabarito, com a explicação dos exercícios!

Vamos começar?

Atividades

Prepare-se para colocar a mão na massa!

Vamos lá?

[INICIAR >](#)



Exercício 1: Lista de Compras (for...of)

Crie um programa que recebe uma lista de compras do usuário e imprime cada item da lista com a frase "Preciso comprar [item]".



Exemplo de Entrada:

Leite, Ovos, Pão, Frango

Exemplo de Saída:

Preciso comprar Leite.

Preciso comprar Ovos.

Preciso comprar Pão.

Preciso comprar Frango.

content_copyUse code with caution.

Exercício 2: Idade Média (for...of)

Crie um programa que recebe uma lista de idades e calcula a idade média das pessoas.



Exemplo de Entrada:

20, 25, 30, 35

content_copyUse code with caution.

Exemplo de Saída:

A idade média é: 27.5

content_copyUse code with caution.

Exercício 3: Cartão de Membros (Map)

Crie um programa que armazena os dados de membros de um clube em um Map. Cada membro deve ter um nome e uma data de aniversário. Imprima o nome e a data de aniversário de cada membro.

Exemplo de Entrada (dados a serem adicionados no Map):

João, 10/10/1990

Maria, 25/12/1995

Pedro, 01/01/2000

content_copyUse code with caution.

Exemplo de Saída:

Nome: João, Data de Aniversário: 10/10/1990

Nome: Maria, Data de Aniversário: 25/12/1995

Nome: Pedro, Data de Aniversário: 01/01/2000

content_copyUse code with caution.

Exercício 4: Senha Segura (WeakMap)

Crie um programa que guarda senhas de usuários em um WeakMap. Cada usuário tem um nome e uma senha. Imprima a senha do usuário "João" (não é possível iterar sobre o WeakMap, apenas acessar o valor associado a uma chave).



Exemplo de Entrada (dados a serem adicionados no WeakMap):

João, 123456

Maria, senha123

content_copyUse code with caution.

Exemplo de Saída:

A senha do usuário João é: 123456

content_copyUse code with caution.

Exercício 5: Lista de Tarefas (Map e for...of)

Crie um programa que simula uma lista de tarefas. Cada tarefa possui um título e um status (pendente, em andamento, concluída). Utilize um Map para armazenar as tarefas. Imprima a lista de tarefas com seus respectivos status.



Exemplo de Entrada (dados a serem adicionados no Map):

Lavar a louça, pendente

Fazer compras, em andamento

Estudar para a prova, concluída

content_copyUse code with caution.

Exemplo de Saída:

Tarefas:

Lavar a louça: pendente

Imprima a lista de tarefas com seus respectivos status.

Exemplo de Entrada (dados a serem adicionados no Map):

Lavar a louça, pendente

Fazer compras, em andamento

Estudar para a prova, concluída

content_copyUse code with caution.



Exemplo de Saída:

Tarefas:

Lavar a louça: pendente

Fazer compras: em andamento

Estudar para a prova: concluída

content_copyUse code with caution.

Gabarito

Agora chegou a hora de conferir!

INICIAR >



Gabarito - Exercício 1

```
const listaCompras = prompt("Digite sua lista de compras, separando os itens por vírgula:").split(',');
```

```
for (const item of listaCompras) {
```

```
    console.log(`Preciso comprar ${item}.`);
```

```
}
```

content_copyUse code with caution.JavaScript

Gabarito - Exercício 2

```
const idades = prompt("Digite as idades, separadas por  
vírgula:").split(',').map(Number);  
  
let somaldades = 0;  
  
for (const idade of idades) {  
  
    somaldades += idade;  
  
}  
  
const idadeMedia = somaldades / idades.length;  
  
console.log(`A idade média é: ${idadeMedia}`);  
  
content_copyUse code with caution.JavaScript
```



Gabarito - Exercício 3

```
const membros = new Map();

membros.set('João', '10/10/1990');

membros.set('Maria', '25/12/1995');

membros.set('Pedro', '01/01/2000');

for (const [nome, dataAniversario] of membros) {

    console.log(`Nome: ${nome}, Data de Aniversário: ${dataAniversario}`);

}

content_copyUse code with caution.Javascript
```



Gabarito- Exercício 4

```
const senhas = new WeakMap();

const usuario1 = { nome: 'João' };

const usuario2 = { nome: 'Maria' };

senhas.set(usuario1, '123456');

senhas.set(usuario2, 'senha123');

console.log(`A senha do usuário João é: ${senhas.get(usuario1)})`);

content_copyUse code with caution.JavasCript
```



Gabarito - Exercício 5

```
const tarefas = new Map();

tarefas.set('Lavar a louça', 'pendente');

tarefas.set('Fazer compras', 'em andamento');

tarefas.set('Estudar para a prova', 'concluída');

console.log('Tarefas:');

for (const [titulo, status] of tarefas) {

  console.log(`${titulo}: ${status}`);

}

content_copyUse code with caution.JavaScript
```



Dicas

- Utilize `prompt()` para receber dados do usuário.
- Utilize `split(',')` para dividir uma string em uma lista de itens.
- Utilize `map(Number)` para converter uma lista de strings em números.
- Utilize `console.log()` para imprimir informações no console.

Você finalizou o Laboratório de Prática! Lembre-se! Você
sempre pode voltar para treinar mais um pouco e rever os
gabaritos. Siga adiante!

CONTINUAR

Programação Assíncrona em JavaScript: Callbacks, Promises e Async/Await

A programação assíncrona é
fundamental em JavaScript para lidar





com operações que levam tempo para serem concluídas, como requisições de rede, acesso a arquivos e temporizadores.



Vamos explorar os principais conceitos e ferramentas para lidar com assincronicidade em JavaScript.

Callbacks

São funções passadas como argumento para outras funções, que serão executadas quando a operação assíncrona for concluída.



Vantagens: simplicidade e ampla compatibilidade com navegadores antigos.



Saiba Mais!

Callback Hell acontece quando muitas funções assíncronas são aninhadas, tornando o código confuso. Para evitar isso, use Promises ou async/await para uma estrutura mais limpa e organizada. Saiba mais clicando no botão.

[ACESSE AQUI](#)



Desvantagens: o famoso "Callback Hell" - código aninhado e difícil de ler e manter em operações assíncronas encadeadas.



Saiba Mais!

Callback Hell acontece quando muitas funções assíncronas são aninhadas, tornando o código confuso. Para evitar isso, use Promises ou async/await para uma estrutura mais limpa e organizada. Saiba mais clicando no botão.

ACESSE AQUI

Promises

Trata-se de objetos que representam o resultado futuro de uma operação assíncrona, podendo estar em três estados: pendente, resolvida (com um valor) ou rejeitada (com um erro).

Vantagens: lidam com o problema do Callback Hell, fornecendo uma maneira mais estruturada de encadear operações assíncronas com `.then()`, `.catch()` e `.finally()`.



É uma sintaxe mais moderna e legível para se trabalhar, permitindo escrever código assíncrono que se parece com código síncrono.

Promises

Trata-se de objetos que representam o resultado futuro de uma operação assíncrona, podendo estar em três estados: pendente, resolvida (com um valor) ou rejeitada (com um erro).

Desvantagens: podem ser um pouco mais complexas de entender inicialmente do que callbacks.



É uma sintaxe mais moderna e legível para se trabalhar, permitindo escrever código assíncrono que se parece com código síncrono.



Vantagens: código mais limpo, legível e fácil de manter, com melhor tratamento de erros usando try...catch.



Callbacks em JavaScript: Analogias e Exemplos Práticos

Callbacks são **funções que passamos como argumento para outras funções**, para que sejam executadas posteriormente, quando um evento ou ação específica acontecer. Essa é a base da programação assíncrona em JavaScript, permitindo que o código lide com tarefas que levam tempo, como buscar dados da internet ou esperar um temporizador.



Desvantagens: requer suporte a recursos mais modernos do JavaScript (ES2017+).



Callbacks em JavaScript: Analogias e Exemplos Práticos

Callbacks são **funções que passamos como argumento para outras funções**, para que sejam executadas posteriormente, quando um evento ou ação específica acontecer. Essa é a base da programação assíncrona em JavaScript, permitindo que o código lide com tarefas que levam tempo, como buscar dados da internet ou esperar um temporizador.

Analogias do Cotidiano

Fazer um Pedido em um Restaurante:

Você faz o pedido ao garçom.	Chama a função.
O garçom entrega o pedido à cozinha.	Passa a função callback.
A cozinha prepara a comida.	Executa a tarefa assíncrona.
Quando a comida fica pronta, a cozinha avisa o garçom.	Chama o callback.
O garçom traz a comida até você.	Executa a função callback, trazendo o resultado.

Agendar uma Consulta Médica:

Agendar uma Consulta Médica:

Você liga para o consultório e agenda a consulta.	Chama a função.
A recepcionista anota seu nome e horário.	Passa a função callback.
No dia da consulta, a recepcionista te chama.	Chama o callback.
Você entra no consultório para a consulta.	Executa a função callback.

Exemplos de Código em JavaScript

INICIAR >



Temporizador (setTimeout)

JavaScript

```
function cumprimentar(nome) {  
    console.log("Olá, " + nome + "!");  
}
```

```
setTimeout(cumprimentar, 3000, "João"); // Chama a função 'cumprimentar' após 3  
segundos
```



```
function cumprimentar(nome) {  
    console.log("Olá, " + nome + "!");  
}
```

```
setTimeout(cumprimentar, 3000, "João"); // Chama a função 'cumprimentar' após 3  
segundos
```



Use o código com cuidado.

- setTimeout é a função que recebe o callback (cumprimentar) e o tempo de espera (3000 milissegundos = 3 segundos).
- Após 3 segundos, o callback é executado, imprimindo a mensagem no console.

Buscar dados de uma API (fetch)

JavaScript

```
function exibirDados(dados) {  
    console.log(dados);  
}  
  
fetch('https://api.example.com/dados')  
    .then(response => response.json())  
    .then(exibirDados)  
    .catch(error => console.error('Erro ao buscar dados:', error));
```

```
fetch('https://api.example.com/dados')

.then(response => response.json())

.then(exibirDados)

.catch(error => console.error('Erro ao buscar dados:', error));
```

Use o código com cuidado.

- fetch busca dados da API.
- .then() encadeia o tratamento da resposta, convertendo-a para JSON.
- O segundo .then() recebe o callback exibirDados, que será executado quando os dados estiverem prontos.
- .catch() lida com possíveis erros durante a busca.



Conclusão

Callbacks são uma ferramenta essencial para lidar com assincronicidade em JavaScript. Ao entender o conceito de passar funções como argumentos para serem executadas posteriormente, você estará pronto para criar aplicações mais interativas e eficientes, que lidam com tarefas demoradas sem travar a interface do usuário.

[INICIAR NOVAMENTE](#)



1 2

Promises

Em essência, uma Promise é como um recibo que você recebe ao fazer um pedido. Esse recibo representa a promessa de que seu pedido será atendido no futuro. O recibo pode estar em três estados:

Analogias do Cotidiano

Pedir um prato em um Restaurante:

Você faz o pedido e recebe um recibo.	A Promise.
Enquanto a comida é preparada, o recibo está pendente.	
Quando o prato está pronto, o recibo é resolvido e você o troca pela comida.	
Se algum problema ocorrer na cozinha, o recibo é rejeitado e você é informado.	

Comprar um produto Online:

Você faz a compra e recebe um número de pedido.

A Promise.

Enquanto o produto é processado e enviado, o pedido está pendente.

Quando o produto chega, o pedido é resolvido.

Se houver algum problema com o pedido, ele é rejeitado e você recebe um aviso.

Exemplos de Código em JavaScript

INICIAR >



Promise Básica

JavaScript

```
const minhaPromise = new Promise((resolve, reject) => {  
    // Simulando uma tarefa assíncrona (pode ser um fetch, setTimeout, etc.)  
    setTimeout(() => {  
        const sucesso = true; // ou false, para simular um erro  
  
        if (sucesso) {  
            resolve("Tarefa concluída com sucesso!");  
        } else {  
            reject("Ocorreu um erro na tarefa.");  
        }  
    }, 2000);  
});
```



```
if (sucesso) {  
    resolve("Tarefa concluída com sucesso!");  
}  
else {  
    reject(new Error("Algo deu errado na tarefa."));  
}  
}, 2000); // Simulando um atraso de 2 segundos  
});
```

minhaPromise

```
.then(resultado => console.log(resultado)) // Executado se a Promise for resolvida  
.catch(erro => console.error(erro)); // Executado se a Promise for rejeitada
```

Encadeando Promises (exemplo com fetch)

JavaScript

```
fetch('https://api.example.com/dados')

.then(response => {

  if (!response.ok) {

    throw new Error('Erro na resposta da rede!');

  }

  return response.json();

})

.then(dados => console.log(dados))
```

```
        throw new Error('Erro na resposta da rede!');

    }

    return response.json();

})

.then(dados => console.log(dados))

.catch(erro => console.error('Erro ao buscar dados:', erro));
```



- fetch retorna uma Promise.
- O primeiro .then() verifica a resposta e a converte para JSON se estiver OK, ou lança um erro.
- O segundo .then() recebe os dados (se a Promise anterior foi resolvida) e os exibe.
- .catch() captura qualquer erro ocorrido em qualquer etapa anterior.



Conclusão

Promises são uma ferramenta poderosa para lidar com assincronicidade em JavaScript, tornando o código mais organizado e legível. Ao entender o conceito de "promessa de um resultado futuro" e os três estados possíveis, você estará pronto para escrever código assíncrono mais robusto e fácil de manter.

[INICIAR NOVAMENTE](#)



1 2

Async/Await

Imagine que você está em uma fila de espera. Em vez de ficar parado esperando sua vez, `async/await` permite que você faça outras coisas enquanto aguarda. Quando sua vez chega, você é "acordado" e pode continuar de onde parou.

- **async:** marca uma função como assíncrona, indicando que ela lidará com Promises e poderá usar `await`. É como dizer: "Ei, essa função pode precisar esperar um pouco!".
- **await:** pausa a execução da função assíncrona até que a Promise seja resolvida ou rejeitada. É como dizer: "Vou esperar aqui até que esteja pronto para continuar".

Vantagens do Async/Await:

Vantagens do Async/Await:



Código mais limpo: a sintaxe se assemelha ao código síncrono, facilitando a leitura e compreensão.



Analogias do Cotidiano:

Fila de espera em um Banco:

Você pega uma senha e aguarda ser chamado.

Função async.

Vantagens do Async/Await:



Tratamento de erros simplificado: use try...catch para lidar com erros de forma clara e concisa.



Analogias do Cotidiano:

Fila de espera em um Banco:

Você pega uma senha e aguarda ser chamado.

Função async.

Vantagens do Async/Await:



Manutenção facilitada: código mais organizado e fácil de entender, mesmo em projetos complexos.



Analogias do Cotidiano:

Fila de espera em um Banco:

Você pega uma senha e aguarda ser chamado.

Função async.

Analogias do Cotidiano:

Fila de espera em um Banco:

Você pega uma senha e aguarda ser chamado.	Função async.
Enquanto espera, você pode ler uma revista ou usar o celular.	Await permite que outras tarefas sejam executadas.
Quando sua senha é chamada, você se levanta e vai ao caixa.	A Promise é resolvida e a execução continua.

Preparar um café da Manhã:

Você coloca a água para ferver.	Inicia uma tarefa assíncrona.

Preparar um café da Manhã:

Você coloca a água para ferver.	Inicia uma tarefa assíncrona.
Enquanto a água ferve, você prepara o pão e pega a manteiga.	Wait permite realizar outras tarefas.
Quando a água ferve, você prepara o café.	A Promise é resolvida e a execução continua.

Exemplos de Código em JavaScript

INICIAR >



Função Assíncrona

JavaScript

```
async function buscarDados() {  
  try {  
    const response = await fetch('https://api.example.com/dados');  
    const dados = await response.json();  
    console.log(dados);  
  } catch (error) {  
    console.error('Erro ao buscar dados:', error);  
  }  
}
```



```
console.log(dados);

} catch (error) {

    console.error('Erro ao buscar dados:', error);

}

buscarDados();
```



- `async` marca a função como assíncrona.
- `await` pausa a execução até que `fetch` retorne a resposta e, em seguida, até que `response.json()` converta os dados para JSON.
- `try...catch` lida com possíveis erros durante o processo.

Encadeando Operações Assíncronas

JavaScript

```
async function processarDados() {  
  try {  
    const dados = await buscarDados();  
  
    const dadosProcessados = await processar(dados);  
  
    console.log(dadosProcessados);  
  
  } catch (error) {  
    console.error('Erro durante o processamento:', error);  
  }  
}
```

```
const dados = await buscarDados(),  
  
    const dadosProcessados = await processar(dados);  
  
    console.log(dadosProcessados);  
  
} catch (error) {  
  
    console.error('Erro durante o processamento:', error);  
  
}  
}  
  
processarDados();
```



- await garante que buscarDados() seja concluída antes de chamar processar(dados).
- O código flui de forma sequencial, facilitando a leitura e compreensão.

Dica

Lembre-se de que `async/await` funciona em conjunto com Promises. Use-o sempre que precisar lidar com operações assíncronas e desejar um código mais estruturado e legível.

1 2 3 ✓



Conclusão

Async/await oferece uma maneira elegante e intuitiva de trabalhar com operações assíncronas em JavaScript. Ao combinar a clareza do código síncrono com o poder das Promises, você pode escrever código assíncrono mais limpo, legível e fácil de manter.

[INICIAR NOVAMENTE](#)



1 2 3 A blue circular button with a white checkmark inside, indicating the current step.

Exercícios de Programação Assíncrona em JavaScript

1

Simulador de Temporizador (Callback)

1

Crie uma função esperar(tempo, callback) que simule um temporizador usando setTimeout.

2

A função deve receber um tempo em milissegundos e uma função callback.

3

Após o tempo especificado, a função callback deve ser executada.

4

Exemplo de uso:

JavaScript

```
function esperar(tempo, callback) {  
    setTimeout(callback, tempo);  
}
```

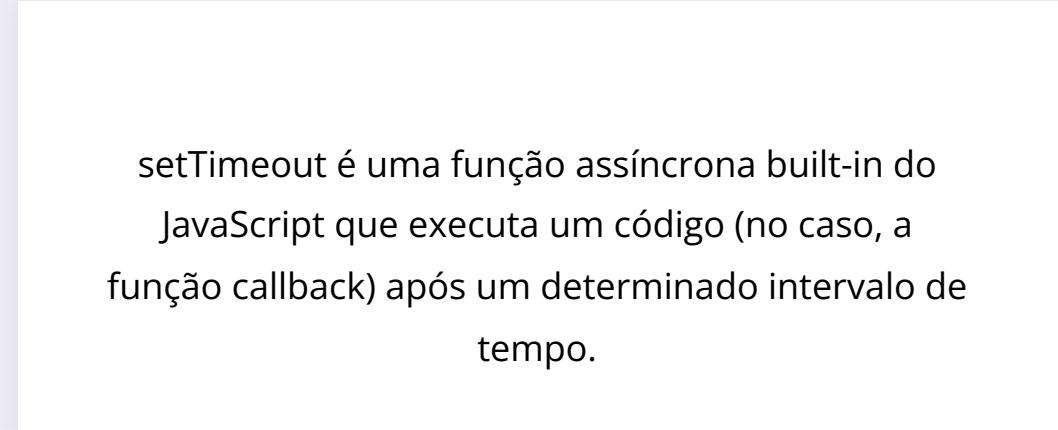
Explicação:

A função esperar utiliza setTimeout para agendar a execução da função callback após o tempo (em milissegundos).

JavaScript

```
function esperar(tempo, callback) {  
    setTimeout(callback, tempo);  
}
```

Explicação:

<  >
setTimeout é uma função assíncrona built-in do JavaScript que executa um código (no caso, a função callback) após um determinado intervalo de tempo.

Calculadora de Promessas (Promise)

1

Crie uma função calcular(operacao, num1, num2) que retorne uma Promise.

2

A Promise deve ser resolvida com o resultado da operação matemática especificada (operacao pode ser "somar", "subtrair", "multiplicar" ou "dividir") entre num1 e num2.

3

Se a operação for inválida ou ocorrer algum erro, a Promise deve ser rejeitada com uma mensagem de erro.

4

Exemplo de uso:

JavaScript

```
function calcular(operacao, num1, num2) {
```

```
    return new Promise((resolve, reject) => {
```

```
        switch (operacao) {
```

```
            case "somar":
```

```
                resolve(num1 + num2);
```

```
                break;
```

```
            case "subtrair":
```

```
                resolve(num1 - num2);
```

```
                break;
```

```
    resolve(num1 - num2);
```

```
    break;
```

```
    case "multiplicar":
```

```
        resolve(num1 * num2);
```

```
    break;
```

```
    case "dividir":
```

```
        if (num2 === 0) {
```

```
            reject("Divisão por zero!");
```

```
        } else {
```

```
            resolve(num1 / num2);
```

```
    resolve(num1 / num2);
```

```
}
```

```
break;
```

```
default:
```

```
    reject("Operação inválida!");
```

```
}
```

```
});
```

```
}
```

Explicação:

}

});

}

Explicação:



A função calcular cria uma nova Promise.



}

});

}

Explicação:

Dentro da Promise, um switch verifica a operação desejada e realiza o cálculo correspondente.



}

});

}

Explicação:



Se a operação for bem-sucedida, resolve é
chamado com o resultado.



```
}
```

```
});
```

```
}
```

Explicação:

Se houver um erro (divisão por zero ou operação inválida), reject é chamado com uma mensagem de erro.



Busca de Dados da API (Fetch com Promises)

1

Crie uma função buscarDados(url) que utilize fetch para buscar dados de uma API.

2

A função deve retornar uma Promise que será resolvida com os dados em formato JSON se a requisição for bem-sucedida.

3

Caso ocorra algum erro na requisição, a Promise deve ser rejeitada com uma mensagem de erro.

4

Exemplo de uso:

JavaScript

JavaScript

```
function buscarDados(url) {  
  
    return fetch(url)  
  
    .then(response => {  
  
        if (!response.ok) {  
  
            throw new Error("Erro na resposta da rede: " + response.status);  
  
        }  
  
        return response.json();  
  
    });
```

```
return response.json();
```

```
});
```

```
}
```

Explicação:

A função buscarDados utiliza fetch para fazer uma requisição à API na URL especificada.



```
return response.json();
```

```
});
```

```
}
```

Explicação:

fetch retorna uma Promise que é resolvida com a resposta da requisição.



```
return response.json();
```

```
});
```

```
}
```

Explicação:



O primeiro .then() verifica se a resposta foi bem-sucedida (`response.ok`). Se não for, lança um erro.



```
    return response.json();
```

```
});
```

```
}
```

Explicação:

Se a resposta for bem-sucedida, response.json() é chamado para analisar o corpo da resposta como JSON.



```
    return response.json();
```

```
});
```

```
}
```

Explicação:

O resultado de response.json() (outra Promise) é
retornado, permitindo encadear mais .then() para
processar os dados.



Simulador de Login (Async/Await)

1

Crie uma função assíncrona `verificarLogin(usuario, senha)` que simule a verificação de login.

2

Utilize `setTimeout` para simular um atraso na resposta do servidor.

3

Se o usuário e senha forem válidos ("admin" e "123456"), a função deve retornar uma Promise resolvida com a mensagem "Login bem-sucedido!".

4

Caso contrário, a Promise deve ser rejeitada com a mensagem "Usuário ou senha inválidos".

5

Exemplo de uso:

```
async function verificarLogin(usuario, senha) {
```

```
    return new Promise((resolve, reject) => {
```

```
        setTimeout(() => {
```

```
            if (usuario === "admin" && senha === "123456") {
```

```
                resolve("Login bem-sucedido!");
```

```
            } else {
```

```
                reject("Usuário ou senha inválidos");
```

```
}
```

```
, 2000); // Simulando um atraso de 2 segundos
```

```
});
```

```
}, 2000); // Simulando um atraso de 2 segundos  
});  
}
```

Explicação:

A função verificarLogin é assíncrona, indicando que
utiliza Promises.



```
}, 2000); // Simulando um atraso de 2 segundos  
});  
}
```

Explicação:

Ela retorna uma nova Promise que simula um atraso de 2 segundos usando setTimeout.



```
}, 2000); // Simulando um atraso de 2 segundos
```

```
});
```

```
}
```

Explicação:

Após o atraso, verifica as credenciais e resolve ou rejeita a Promise de acordo.



```
}, 2000); // Simulando um atraso de 2 segundos
```

```
});
```

```
}
```

Explicação:

O uso de `async/await` na função `tentarLogin` permite aguardar a resolução da Promise de forma mais síncrona e legível.



```
}, 2000); // Simulando um atraso de 2 segundos
```

```
});
```

```
}
```

Explicação:

O resultado de response.json() (outra Promise) é
retornado, permitindo encadear mais .then() para
processar os dados.



Carregamento de Imagens em Sequência (Async/Await)

1

Crie uma função assíncrona carregarImagens(urls) que receba um array de URLs de imagens.

2

Utilize um loop for...of e await para carregar cada imagem em sequência, simulando um atraso com setTimeout.

3

A cada imagem carregada, exiba uma mensagem no console indicando o progresso.

4

Se ocorrer algum erro durante o carregamento de uma imagem, trate-o adequadamente e continue o processo com as próximas imagens.

5

Exemplo de uso:

```
async function carregarImagens(urls) {  
  
    for (const url of urls) {  
  
        try {  
  
            const response = await fetch(url);  
  
            if (!response.ok) {  
  
                throw new Error("Erro ao carregar imagem: " + response.status);  
  
            }  
  
            console.log(`Imagem ${url} carregada com sucesso!`);  
  
        } catch (error) {  
  
            console.error(`Erro ao carregar imagem ${url}:`, error);  
        }  
    }  
}
```

```
        console.error(`Erro ao carregar imagem ${url}:`, error);  
  
    }  
  
}  
  
}
```

Explicação:

A função carregarImagens é assíncrona e recebe
um array de URLs de imagens.



```
        console.error(`Erro ao carregar imagem ${url}:`, error);  
  
    }  
  
}  
  
}
```

Explicação:



Ela itera sobre as URLs usando um loop for...of.



```
        console.error(`Erro ao carregar imagem ${url}:`, error);  
  
    }  
  
}  
  
}
```

Explicação:

Para cada URL, fetch é usado para buscar a imagem, e await pausa a execução até que a Promise seja resolvida.



```
        console.error(`Erro ao carregar imagem ${url}:`, error);

    }

}

}
```

Explicação:

Se a resposta for bem-sucedida, uma mensagem de sucesso é exibida no console.



```
        console.error(`Erro ao carregar imagem ${url}:`, error);  
  
    }  
  
}  
  
}
```

Explicação:

Se ocorrer um erro, ele é capturado pelo bloco catch e uma mensagem de erro é exibida, mas o loop continua para as próximas imagens.



Dê o play!

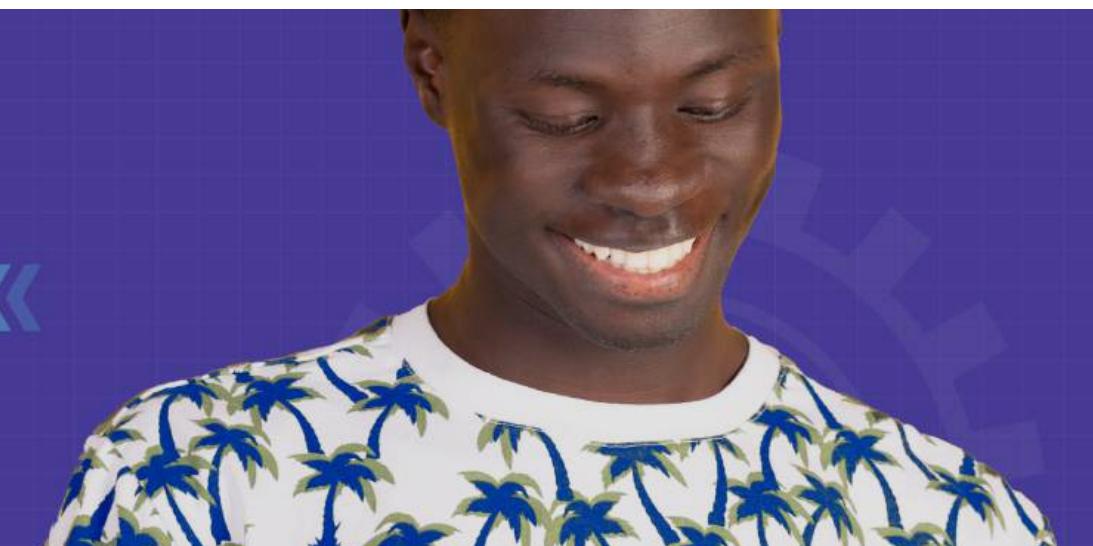
AULA 6 - PROMISES



Consumo de APIs com Fetch API

- Fundamentos e Práticas

A Fetch API é uma interface moderna em JavaScript para realizar requisições HTTP assíncronas, permitindo a





comunicação com serviços web e a obtenção de dados de forma estruturada.



Conceitos-Chave

FETCH()

PROMISES

.THEN()

.CATCH()

Função principal da API, inicia uma requisição HTTP para a URL especificada e Retorna uma Promise que se resolve com um objeto Response contendo os detalhes da resposta do servidor.



comunicação com serviços web e a obtenção de dados de forma estruturada.



Conceitos-Chave

FETCH()

PROMISES

.THEN()

.CATCH()

Objetos que representam o resultado futuro de uma operação assíncrona. Permitem lidar com o sucesso ou falha da requisição de forma organizada.



comunicação com serviços web e a obtenção de dados de forma estruturada.



Conceitos-Chave

FETCH()

PROMISES

.THEN()

.CATCH()

Método encadeado a uma Promise, executa uma função callback quando a Promise é resolvida com sucesso, recebendo o valor resolvido como argumento.



comunicação com serviços web e a obtenção de dados de forma estruturada.



Conceitos-Chave

FETCH()

PROMISES

.THEN()

.CATCH()

Método encadeado a uma Promise, executa uma função callback quando a Promise é rejeitada, recebendo o motivo da rejeição (erro) como argumento.

Exemplos Práticos

INICIAR >



Requisição GET

JavaScript

```
fetch('https://api.example.com/data')

.then(response => {

  if (response.ok) {

    return response.json();

  } else {

    throw new Error('Erro na requisição: ' + response.status);

  }

})
```



```
    } else {  
  
        throw new Error('Erro na requisição: ' + response.status);  
  
    }  
  
})  
  
.then(data => {  
  
    console.log(data);  
  
})  
  
.catch(error => {  
  
    console.error('Erro:',  
error);  
  
});
```



Este código JavaScript utiliza a Fetch API para realizar uma requisição GET à URL <https://api.example.com/data> e processar a resposta do servidor. Vamos analisar cada parte:



1. `fetch('https://api.example.com/data'):`

- Inicia uma requisição HTTP do tipo GET para a URL especificada.
- Retorna uma Promise que será resolvida quando a resposta do servidor estiver disponível.

2. `.then(response => { ... }):`

- O primeiro `.then()` é encadeado à Promise retornada por `fetch()`.

- O primeiro .then() é encadeado à Promise retornada por fetch().
- A função callback dentro dele recebe o objeto response, que contém os detalhes da resposta do servidor.
- Dentro da função: if (response.ok): Verifica se o status da resposta está no intervalo 200-299 (sucesso).
 - return response.json(): se a resposta for bem-sucedida, chama o método json() do objeto response para analisar o corpo da resposta como JSON e retorna uma nova Promise que será resolvida com os dados em formato de objeto JavaScript.
 - else: se a resposta não for bem-sucedida, lança um novo erro com a mensagem "Erro na requisição" e o código de status.

3. **.then(data => { ... }):**

- O segundo .then() é encadeado à Promise retornada pelo primeiro .then().
- A função callback dentro dele recebe os data (objeto JavaScript) resultantes da

- O segundo .then() é encadeado à Promise retornada pelo primeiro .then().
- A função callback dentro dele recebe os data (objeto JavaScript) resultantes da análise do JSON.
- Dentro da função: console.log(data): imprime os dados no console do navegador.

4. **.catch(error => { ... }):**

- O .catch() é encadeado à cadeia de Promises para lidar com qualquer erro que ocorra durante a requisição ou o processamento da resposta.
- A função callback dentro dele recebe o objeto error que representa o erro ocorrido.
- Dentro da função: console.error('Erro:', error): Imprime uma mensagem de erro no console, juntamente com os detalhes do erro.

Requisição POST

JavaScript

```
const newData = {  
    name: "Novo Item",  
    value: 123  
};  
  
fetch('https://api.example.com/data', {  
    method: 'POST',  
    headers: {  
        'Content-Type': 'application/json'  
    },  
    body: JSON.stringify(newData)  
});
```



'Content-Type': 'application/json'

},

body: JSON.stringify(newData)

)

.then(response => {

if (response.ok) {

return response.json();

} else

{

throw new Error('Erro na requisição: ' + response.status);

}

)

.then(responseData => {

console.log("Dados enviados com sucesso!", responseData);



```
    } else

    {

        throw new Error('Erro na requisição: ' + response.status);

    }

}

.then(responseData => {

    console.log("Dados enviados com sucesso!", responseData);

})

.catch(error => {

    console.error("Erro no envio dos dados:", error);

});
```



Este código JavaScript utiliza a Fetch API para enviar dados para um servidor usando o método POST.
Vamos analisar cada parte:



1. `const newData = { ... };`:

- Cria um objeto JavaScript chamado newData que contém os dados que serão enviados para o servidor.
- No exemplo, o objeto tem duas propriedades: name com o valor "Novo Item" e value com o valor 123.

2. `fetch('https://api.example.com/data', { ... });`:

- Inicia uma requisição HTTP do tipo POST para a URL https://api.example.com/data.

- Inicia uma requisição HTTP do tipo POST para a URL `https://api.example.com/data`.

- O segundo argumento é um objeto de configuração com as seguintes propriedades:

- `method: 'POST'`: especifica que a requisição é do tipo POST, usada para enviar dados para o servidor.

- `headers: { 'Content-Type': 'application/json' }`: define o cabeçalho Content-Type como application/json, indicando que o corpo da requisição contém dados no formato JSON.

- `body: JSON.stringify(newData)`: Converte o objeto newData em uma string JSON e a define como o corpo da requisição.

3. `.then(response => { ... })`:

- O primeiro `.then()` é encadeado à Promise retornada por `fetch()`.

- A função callback dentro dele recebe o objeto `response`, que contém os detalhes da resposta do servidor.



- Dentro da função: if (response.ok): verifica se o status da resposta está no intervalo 200-299 (sucesso).
- return response.json(): se a resposta for bem-sucedida, chama o método json() do objeto response para analisar o corpo da resposta como JSON e retorna uma nova Promise que será resolvida com os dados em formato de objeto JavaScript.
- else: se a resposta não for bem-sucedida, lança um novo erro com a mensagem "Erro na requisição" e o código de status.



4. **.then(responseData => { ... }):**

- O segundo .then() é encadeado à Promise retornada pelo primeiro .then().
- A função callback dentro dele recebe os responseData (objeto JavaScript) resultantes da análise do JSON da resposta do servidor.
- Dentro da função: console.log("Dados enviados com sucesso!", responseData): imprime uma mensagem de sucesso no console, juntamente com os dados recebidos do servidor.

A função callback dentro dele recebe os `responseData` (objeto JavaScript) resultantes da análise do JSON da resposta do servidor.

- Dentro da função: `console.log("Dados enviados com sucesso!", responseData)`: imprime uma mensagem de sucesso no console, juntamente com os dados recebidos do servidor.

5. `.catch(error => { ... })`:

- O `.catch()` é encadeado à cadeia de Promises para lidar com qualquer erro que ocorra durante a requisição ou o processamento da resposta.
- A função callback dentro dele recebe o objeto `error` que representa o erro ocorrido.
- Dentro da função: `console.error("Erro no envio dos dados:", error)`: imprime uma mensagem de erro no console, juntamente com os detalhes do erro.

Requisição PUT (Atualização de Dados)

```
const updatedData = {  
  id: 1,  
  name: "Item Atualizado",  
  value: 456  
};  
  
fetch('https://api.example.com/data/1', { // URL inclui o ID do item a ser atualizado  
  method: 'PUT',  
  headers: {  
    'Content-Type': 'application/json'  
  }  
}
```



```
'Content-Type': 'application/json'

},
body: JSON.stringify(updatedData)

})

.then(response => {

if (response.ok) {

return response.json();

}

} else {

throw new Error('Erro na atualização: ' + response.status);

}

})

.then(responseData => {
```



```
    } else {  
  
        throw new Error('Erro na atualização: ' + response.status);  
  
    }  
  
})  
  
.then(responseData => {  
  
    console.log("Dados atualizados com sucesso!", responseData);  
  
})  
  
.catch(error => {  
  
    console.error("Erro na atualização dos dados:", error);  
  
});
```

Requisição PUT

Imagine que você tem uma caixa de ferramentas mágica na internet (<https://api.example.com/data>). Dentro dessa caixa, há várias ferramentas, cada uma com seu próprio número de identificação (ID).



Você quer modificar a ferramenta de número 1 (/data/1). Para isso, você:

1. Prepara a ferramenta modificada:

- Pega uma nova ferramenta (updatedData) com as características desejadas:
- ID: 1 (mantém o mesmo ID para identificar a ferramenta correta)
- Nome: "Item Atualizado"
- Valor: 456

- Valor: 456

2. Envia a ferramenta modificada para a caixa:

- Usa um serviço de entrega especial (fetch) para levar a ferramenta até a caixa mágica.
- Informa ao serviço de entrega:
 - **Destino:** a caixa de ferramentas na internet (<https://api.example.com/data/1>)
 - **Método de entrega:** substituir a ferramenta existente pela nova (PUT)
 - **Conteúdo da embalagem:** a ferramenta está embalada em um formato especial que a caixa mágica entende (Content-Type: application/json)
 - **Conteúdo da caixa:** a nova ferramenta (body: JSON.stringify(updatedData))

3. Aguarda a confirmação de entrega:



3. Aguarda a confirmação de entrega:

- O serviço de entrega te envia uma mensagem (response) quando a entrega é concluída.
- Você verifica a mensagem:
 - **Se a entrega foi bem-sucedida:**
 - Abre a mensagem para ver os detalhes da ferramenta atualizada na caixa (response.json()).
 - Comemora: "Dados atualizados com sucesso!" e mostra os detalhes da ferramenta (responseData).
 - **Se houve algum problema na entrega:**
 - Identifica o erro e avisa: "Erro na atualização" com os detalhes do problema (Error).



4. Em caso de imprevistos:

- Abre a mensagem para ver os detalhes da ferramenta atualizada na caixa (response.json()).
- Comemora: "Dados atualizados com sucesso!" e mostra os detalhes da ferramenta (responseData).

- **Se houve algum problema na entrega:**

- Identifica o erro e avisa: "Erro na atualização" com os detalhes do problema (Error).

4. Em caso de imprevistos:

- Se algo inesperado acontecer durante o processo (endereço errado, caixa mágica fechada, etc.), você recebe um alerta (catch) com a descrição do problema (error) e toma as medidas necessárias.

Requisição DELETE (Remoção de Dados)

```
fetch('https://api.example.com/data/1', { // URL inclui o ID do item a ser removido
  method: 'DELETE'
})
.then(response => {
  if (response.ok) {
    console.log("Item removido com sucesso!");
  } else {
    throw new Error('Erro na remoção: ' + response.status);
  }
})
```



)

```
.then(response => {  
  
    if (response.ok) {  
  
        console.log("Item removido com sucesso!");  
  
    } else {  
  
        throw new Error('Erro na remoção: ' + response.status);  
    }  
  
})  
  
.catch(error => {  
  
    console.error("Erro na remoção do item:", error);  
  
});
```



Analogia para o código de Requisição DELETE

Imagine que você tem uma estante de livros na internet (<https://api.example.com/data>). Cada livro tem um número de identificação único (ID).

Você decidiu se desfazer do livro de número 1 (/data/1). Para isso, você:

1. Envia um pedido de remoção:

- Usa um serviço de descarte de livros online (fetch) para remover o livro da estante.
- Informa ao serviço:
 - **Livro a ser removido:** o livro de número 1 na estante (<https://api.example.com/data/1>)
 - **Ação:** remover o livro (DELETE)

- 
- **Ação:** remover o livro (DELETE)

2. Aguarda a confirmação:

- O serviço de descarte te envia uma mensagem (response) quando a remoção é concluída.

- Você verifica a mensagem:

- **Se a remoção foi bem-sucedida:**

- Comemora: "Item removido com sucesso!"

- **Se houve algum problema na remoção:**

- Identifica o erro e avisa: "Erro na remoção" com os detalhes do problema (Error).

3. Em caso de imprevistos:

- Se algo inesperado acontecer durante o processo (livro não encontrado, estante

- 
- Identifica o erro e avisa: "Erro na remoção" com os detalhes do problema (Error).

3. Em caso de imprevistos:

- Se algo inesperado acontecer durante o processo (livro não encontrado, estante inacessível, etc.), você recebe um alerta (catch) com a descrição do problema (error) e toma as medidas necessárias.

Em resumo, o código:

- Remove um item específico de um servidor remoto.
- Utiliza o método DELETE da Fetch API para excluir os dados.
- Lida com a resposta do servidor, tanto em caso de sucesso quanto de erro.

Dê o play!

AULA 7 - FETCH API



Módulos em JavaScript

Imagine que você está construindo
uma casa. Em vez de fazer tudo de uma
vez, você divide o projeto em partes



menores, como fundação, paredes e telhado. Cada parte tem uma função específica e, juntas, formam a casa completa.



Em JavaScript, os módulos funcionam de maneira semelhante. Eles permitem que você divida seu código em arquivos separados, cada um com seu próprio conjunto de funções, variáveis e classes. Isso torna seu código mais organizado, reutilizável e fácil de manter.

Analogia da Caixa de Ferramentas

Pense em um módulo como uma caixa de ferramentas especializada. Cada caixa contém ferramentas específicas para uma determinada tarefa, como marcenaria, encanamento ou eletricidade. Quando você precisa realizar uma tarefa específica, você pega a caixa de ferramentas apropriada e usa as ferramentas dentro dela.



Da mesma forma, cada módulo JavaScript contém código específico para uma determinada funcionalidade. Quando você precisa usar essa funcionalidade em seu programa principal, você importa o módulo e usa as funções, variáveis ou classes que ele exporta.

Exemplos Conceituais e Código

Vamos explorar alguns conceitos chave de módulos com exemplos simples e didáticos.

INICIAR >



Exportando e Importando

Exportando: É como colocar ferramentas dentro da caixa. Você usa a palavra-chave export para tornar funções, variáveis ou classes disponíveis para outros módulos.

JavaScript

```
// arquivo: matematica.js

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

export function subtrair(a, b) {
    return a - b;
}
```

}

Importando: É como pegar a caixa de ferramentas e usar as ferramentas dentro dela. Você usa a palavra-chave import para trazer as funcionalidades exportadas para o seu código principal.

// arquivo: programa.js

```
import { somar, subtrair } from './matematica.js';
```

```
let resultadoSoma = somar(5, 3); // resultadoSoma será 8
```

```
let resultadoSubtracao = subtrair(10, 4); // resultadoSubtracao será 6
```

Use o código com cuidado.

Export Default

Às vezes, você quer exportar apenas uma coisa principal de um módulo. É como ter uma caixa de ferramentas com uma única ferramenta grande e importante.



```
// arquivo: calculadora.js

function calcular(a, b, operacao) {

    if (operacao === '+') {

        return a + b;

    } else if (operacao === '-') {

        return a - b;

    } else {

        return "Operação inválida";
    }
}
```

```
    return a - b;  
  } else {  
    return "Operação inválida";  
  }  
}  
  
export default calcular;
```



Para importar um export default, você pode dar qualquer nome a ele.

```
// arquivo: programa.js  
  
import minhaCalculadora from './calculadora.js';  
  
let resultado = minhaCalculadora(7, 2, '+'); // resultado será 9
```

Organização e Reutilização

- Os módulos ajudam a manter seu código organizado, dividindo-o em partes menores e mais focadas.
- Você pode reutilizar módulos em diferentes partes do seu projeto, evitando duplicação de código.
- Módulos também facilitam o trabalho em equipe, permitindo que diferentes pessoas trabalhem em partes separadas do código sem conflitos.

Escopo de Módulo

- Cada módulo tem seu próprio escopo. Isso significa que as variáveis, funções e classes definidas dentro de um módulo não estão automaticamente disponíveis em outros módulos, a menos que sejam explicitamente exportadas.
- Isso ajuda a evitar conflitos de nomes e torna seu código mais seguro e previsível.

Evolução dos Módulos

- Os módulos são um recurso relativamente novo em JavaScript. Antes, os desenvolvedores usavam outras técnicas, como padrões de módulo e bibliotecas externas, para alcançar modularidade.
- Atualmente, os módulos são suportados nativamente pela maioria dos navegadores modernos e pelo Node.js.

Exemplo Completo de Módulos ES6+ com Configuração Passo a Passo

Vamos criar um exemplo prático para ilustrar o uso de módulos ES6+ em JavaScript, incluindo a configuração necessária para habilitar esse recurso em sua aplicação.

Estrutura do Projeto

```
projeto-modulos/
    ├── calculadora.js
    ├── utilitarios.js
    └── app.js
```

Conteúdo dos Arquivos

- calculadora.js (Módulo de Cálculos)

JavaScript

```
export function somar(a, b) {  
    return a + b;  
}  
  
export function subtrair(a, b) {  
    return a - b;  
}  
  
export function multiplicar(a, b) {  
    return a * b;  
}  
  
export function dividir(a, b) {  
    if (b === 0) {  
        throw new Error("Divisão por zero!");  
    }  
}
```

```
    return a / b;  
}
```

- utilitarios.js (Módulo de Utilitários)

```
export function saudacao(nome) {  
    return `Olá, ${nome}!`;  
}
```

```
export function dataAtual() {  
    const hoje = new Date();  
    return hoje.toLocaleDateString();  
}
```

- app.js (Arquivo Principal)

- app.js (Arquivo Principal)

```
import { somar, subtrair } from './calculadora.js';
import { saudacao, dataAtual } from './utilitarios.js';

console.log(saudacao("Alice")); // Saída: Olá, Alice!
console.log("Data de hoje:", dataAtual());

const resultadoSoma = somar(10, 5);
console.log("Resultado da soma:", resultadoSoma);

const resultadoSubtracao = subtrair(20, 8);
console.log("Resultado da subtração:", resultadoSubtracao);
```

Configuração para Módulos ES6+

Para habilitar o uso de módulos ES6+ em sua aplicação, você precisa seguir estes passos:

1

Especificar o tipo de módulo no arquivo HTML (para aplicações web):

No seu arquivo HTML principal, adicione o atributo type="module" à tag <script> que carrega seu arquivo JavaScript principal (app.js). Exemplo:

HTML

```
<!DOCTYPE html>

<html>

<head>

<title>Exemplo de Módulos</title>

</head>

<body>
```

```
<script type="module" src="app.js"></script>
```

```
</body>
```

```
</html>
```

2

Usar um empacotador de módulos (para Node.js ou projetos mais complexos):

- Se você estiver executando seu código no Node.js ou se seu projeto tiver muitos módulos e dependências, é recomendado usar um empacotador de módulos como o Webpack ou o Rollup.js.
- Esses empacotadores permitem que você organize seus módulos de forma eficiente, lide com dependências e otimize o código para produção.
- A configuração específica para usar um empacotador varia dependendo da ferramenta escolhida. Consulte a documentação do empacotador para obter instruções detalhadas.

Executando o Exemplo

No navegador

- Crie os arquivos `calculadora.js`, `utilitarios.js` e `app.js` em uma pasta.
- Crie o arquivo HTML conforme descrito acima.
- Abra o arquivo HTML no seu navegador. Você deverá ver as saídas no console do navegador.

No Node.js (com empacotador)

- Configure seu projeto Node.js para usar um empacotador de módulos.
- Execute o comando de build do empacotador para gerar um arquivo JavaScript empacotado.
- Execute o arquivo empacotado usando o Node.js.

Dê o play!

AULA 8 - MODULARIZAÇÃO



HORA DA PRÁTICA!

—
Laboratório
de Prática



Olá, Bem-vindo ao Laboratório de Práticas

Nesse espaço você encontrará alguns exercícios para praticar o conteúdo estudado neste curso. Depois de praticar, não esqueça de conferir o gabarito com a explicação dos exercícios!

Vamos começar?

Atividades

Prepare-se para colocar a mão na massa!

Vamos lá?

[INICIAR >](#)



Exercício 1: Biblioteca de Operações Matemáticas

Objetivo: criar um módulo JavaScript que realiza operações matemáticas básicas.

Instruções:

- Crie um arquivo chamado operacoes.js.
- Dentro do arquivo, defina as seguintes funções:
 - somar(a, b): recebe dois números como parâmetros e retorna a soma deles.
 - subtrair(a, b): recebe dois números como parâmetros e retorna a subtração do primeiro pelo segundo.
 - multiplicar(a, b): recebe dois números como parâmetros e retorna a multiplicação deles.
 - dividir(a, b): recebe dois números como parâmetros e retorna a divisão do primeiro pelo segundo. Caso o segundo número seja zero, lance um erro (throw



new Error("Divisão por zero!").

- Exporte cada uma das funções usando export.

Exemplo de uso:

/ arquivo: main.js

```
import { somar, subtrair, multiplicar, dividir } from './operacoes.js';
```



```
console.log(somar(5, 3)); // Saída: 8
```

```
console.log(subtrair(10, 4)); // Saída: 6
```

```
console.log(multiplicar(2, 5)); // Saída: 10
```

```
console.log(dividir(10, 2)); // Saída: 5
```

```
// console.log(dividir(10, 0)); // Lança um erro
```

content_copyUse code with caution.JavaScript

Exercício 2: Conversor de Unidades

Objetivo: Criar um módulo JavaScript que converte unidades de medida.

Instruções:

- Crie um arquivo chamado conversor.js.
- Dentro do arquivo, defina as seguintes funções:
 - celsiusParaFahrenheit(celsius): recebe uma temperatura em Celsius como parâmetro e retorna a temperatura equivalente em Fahrenheit.
 - fahrenheitParaCelsius(fahrenheit): recebe uma temperatura em Fahrenheit como parâmetro e retorna a temperatura equivalente em Celsius.
 - metrosParaPés(metros): recebe uma distância em metros como parâmetro e retorna a distância equivalente em pés.
 - pésParaMetros(pés): recebe uma distância em pés como parâmetro e retorna a



distância equivalente em metros.

- Exporte cada uma das funções usando export.

Exemplo de uso:

```
// arquivo: main.js  
  
import { celsiusParaFahrenheit, fahrenheitParaCelsius, metrosParaPés,  
pésParaMetros } from './conversor.js';
```

```
console.log(celsiusParaFahrenheit(25)); // Saída: 77
```

```
console.log(fahrenheitParaCelsius(77)); // Saída: 25
```

```
console.log(metrosParaPés(10)); // Saída: 32.80839895
```

```
console.log(pésParaMetros(32.80839895)); // Saída: 10
```

```
content_copyUse code with caution.JavaScript
```



Gabarito

Agora chegou a hora de conferir!

INICIAR >



Gabarito - Exercício 1

```
// arquivo: operacoes.js

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

export function subtrair(a, b) {
    return a - b;
}

export function multiplicar(a, b) {
    return a * b;
}
```



```
}
```

```
export function multiplicar(a, b) {
```

```
    return a * b;
```

```
}
```

```
export function dividir(a, b) {
```

```
    if (b === 0) {
```

```
        throw new Error("Divisão por zero!");
```

```
}
```

```
    return a / b;
```

```
}
```



Gabarito - Exercício 2

```
// arquivo: conversor.js

export function celsiusParaFahrenheit(celsius) {
    return (celsius * 9 / 5) + 32;
}

export function fahrenheitParaCelsius(fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
}

export function metrosParaPés(metros) {
    return metros * 3.280839895;
}
```



```
    return (celsius * 9 / 5) + 32;  
}  
  
export function fahrenheitParaCelsius(fahrenheit) {  
    return (fahrenheit - 32) * 5 / 9;  
}  
  
export function metrosParaPés(metros) {  
    return metros * 3.280839895;  
}  
  
export function pésParaMetros(pés) {  
    return pés / 3.280839895;  
}
```



Você finalizou o Laboratório de Prática! Lembre-se! Você
sempre pode voltar para treinar mais um pouco e rever os
gabaritos. Siga adiante!

CONTINUAR

Finalizando

Parabéns por ter chegado até aqui! Você concluiu o conteúdo de Javascript ES6+.

Ao final deste curso, você estará pronto para usar JavaScript ES6+ em seus projetos web, tornando-os mais dinâmicos e interativos. Você aprendeu a:

- Compreender e aplicar métodos auxiliares para manipular arrays e utilizar

eficientemente o laço for...of para percorrer coleções.

- Desenvolver a habilidade de acessar, modificar e criar elementos na estrutura do DOM para construir e manipular a interface da página web.
 - Utilizar corretamente callbacks, promises e o padrão async/await para realizar tarefas assíncronas e lidar com operações que envolvem o consumo de APIs externas usando a Fetch API.
 - Aprender a dividir código em módulos reutilizáveis para melhorar a manutenção, legibilidade e escalabilidade dos projetos em JavaScript.
-

Não se esqueça de praticar e implementar esses
conhecimentos em seus projetos web.

Siga em frente e boa sorte!

Siga em frente e boa sorte!

Até a próxima!

Feche a janela do seu navegador
para finalizar o módulo.

