



Fundamentos de JavaScript

Diego Nascimento

✓ Introdução ao JavaScript

O que é JavaScript e para que serve

Como o JS funciona no navegador (Engine V8, por exemplo)

Onde escrever (console, , editores como VS Code)

✓ Sintaxe Básica

Variáveis (let, const, var)

Tipos de dados (string, number, boolean, null, undefined, symbol, object)

Operadores (aritméticos, lógicos, comparação)

✓ Controle de Fluxo

Condicionais (if, else, switch)

Loops (for, while, do...while, for...of, for...in)

✓ Funções

Declaração e expressão

Arrow functions

Parâmetros padrão, rest/spread

Escopo e closures

✓ Objetos e Arrays

Criação, acesso e manipulação

Métodos importantes: .map(), .filter(), .reduce(), .forEach()

✓ DOM (Document Object Model)

Seleção de elementos (getElementById, querySelector)

Manipulação de conteúdo e atributos

Eventos (addEventListener)



Fundamentos de JavaScript



Objetivo Geral

Entender como JavaScript funciona em sua forma pura, sem frameworks ou bibliotecas externas, focando na base da linguagem e na manipulação do DOM.



Módulo 1: Introdução ao JavaScript



O que é JavaScript e para que serve

- Linguagem de programação interpretada.
- Utilizada principalmente para desenvolvimento web (client-side).
- Também usada no backend (Node.js).
- Permite criar interações dinâmicas.




Como o JS funciona no navegador

- O navegador possui uma **engine** (ex: V8 no Chrome).
- Interpreta e executa o JS em tempo real.
- Interage com o DOM.

Onde escrever JS

- Console do navegador (DevTools).
- Tag `<script>` no HTML.
- Arquivos `.js` externos (linkados ao HTML).
- Editores como **VS Code**.

 **Atividade:** Criar um HTML com um `<script>` simples que mostra `console.log("Olá, JavaScript!")`.

Módulo 2: Sintaxe Básica

Variáveis

- `var`, `let`, `const` (diferenças de escopo e mutabilidade).

Tipos de Dados

- Primitivos: `string`, `number`, `boolean`, `null`, `undefined`, `symbol`
- Estruturados: `object`, `array`

Operadores

- Aritméticos: `+`, `-`, `*`, `/`, `%`
- Comparação: `==`, `===`, `!=`, `!==`, `<`, `>`, `<=`, `>=`

- Lógicos: `&&`, `||`, `!`



Atividade: Criar variáveis com diferentes tipos e operar entre elas.



Módulo 3: Controle de Fluxo



Condicionais

- `if, else, else if`
- `switch`



Loops

- `for, while, do...while`
- `for...of, for...in`



Atividade: Criar um loop que percorre de 1 a 10 e imprime se o número é par ou ímpar.



Módulo 4: Funções



Tipos de função


- Declaração tradicional
- Expressão de função
- Arrow function (`=>`)

Parâmetros

- Padrão
- `...rest`
- `...spread`

Escopo e Closures

- Escopo léxico
- Funções dentro de funções

 **Atividade:** Criar uma função que soma N números usando `rest` e `reduce`.


Módulo 5: Objetos e Arrays

Objetos

- Criação e acesso (`obj.chave` ou `obj['chave']`)
- Modificação e exclusão

Arrays

- Métodos: `.push()`, `.pop()`, `.shift()`, `.unshift()`
- Iteração: `.map()`, `.filter()`, `.reduce()`, `.forEach()`

 **Atividade:** Criar um array de pessoas e usar `.filter()` para selecionar maiores de idade.

Módulo 6: DOM – Document Object Model

Selecionar elementos


- `getElementById`, `getElementsByClassName`
- `querySelector`, `querySelectorAll`

Manipular conteúdo e atributos

- `.textContent`, `.innerHTML`, `.value`
- `.setAttribute()`, `.getAttribute()`

Eventos

- `addEventListener('click', callback)`
- Eventos comuns: `click`, `input`, `mouseover`, `submit`

 **Atividade:** Criar um botão que muda o texto de um parágrafo ao ser clicado.

✓ Módulo 1: Introdução ao JavaScript

📌 O que é JavaScript e para que serve

🟡 Definição

JavaScript é uma linguagem de programação **interpretada, dinâmica e baseada em protótipos**, criada para trazer **interatividade** às páginas web.

- Foi criada em 1995 por **Brendan Eich**.
- Inicialmente chamada de **LiveScript**, depois renomeada para JavaScript por motivos de marketing.
- Apesar do nome, **não tem relação direta com Java**.

🟢 Para que serve o JavaScript?

O JavaScript é uma das **linguagens fundamentais da web**, junto com **HTML** e **CSS**. Ele é usado principalmente para:

✓ Navegadores (Front-end)

- Tornar sites **interativos**: cliques, animações, validação de formulários.
- Modificar conteúdo da página **dinamicamente** (DOM).
- Comunicar-se com servidores sem recarregar a página (AJAX, fetch API).

✓ Servidores (Back-end)

- Utilizando **Node.js**, o JS pode ser usado no servidor para:
 - Criar APIs
 - Manipular bancos de dados
 - Criar sistemas completos (full-stack)

✓ Outras aplicações

- Apps mobile (React Native)
 - Games em navegador
 - Automação de tarefas
 - Aplicações de desktop (Electron)
-

● Por que aprender JavaScript?

- É uma das linguagens **mais populares do mundo**.
- É **essencial** para quem quer trabalhar com **desenvolvimento web**.
- Tem uma **curva de aprendizado acessível**.
- Tem uma **comunidade enorme** e muitos recursos gratuitos.

Exemplo simples

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8">
    <title>Primeiro JS</title>
  </head>
  <body>
    <h1>Olá, mundo!</h1>
    <script>
      alert("Bem-vindo ao JavaScript!");
      console.log("Olá do console!");
    </script>
  </body>
</html>
```

Atividade rápida

Abra o console do navegador (F12 → aba Console) e digite:

```
console.log("Testando JS!");
```

Deseja que eu continue com o próximo item (**Como o JS funciona no navegador**)?

Como o JavaScript funciona no navegador

O papel do navegador

Quando você abre uma página da web, o navegador carrega e interpreta três tecnologias principais:

1. **HTML** – estrutura do conteúdo
2. **CSS** – aparência visual
3. **JavaScript** – comportamento e interatividade

O JavaScript é executado diretamente no navegador, sem a necessidade de compilação, graças a algo chamado **engine JavaScript**.

O que é uma Engine JavaScript?

Uma **engine** é o mecanismo interno do navegador responsável por interpretar e executar o código JavaScript.

Cada navegador possui sua própria engine:

Navegador	Engine JS
-----------	-----------

Chrome, Edge	V8
-----------------	-----------

Firefox	SpiderMonkey
---------	---------------------

Safari	JavaScriptCore
--------	-----------------------

Opera **V8**

A engine lê seu código linha por linha, converte em código de máquina e executa. A V8, por exemplo, é extremamente rápida e também é usada no Node.js (fora do navegador).

Etapas da execução no navegador

1. **HTML é lido primeiro**
2. Quando encontra uma tag `<script>`, o navegador:
 - Pausa o carregamento do HTML
 - Executa o código JS
 - Depois continua com o HTML
3. O JavaScript pode:
 - Modificar o conteúdo (DOM)
 - Reagir a eventos (cliques, teclas)
 - Fazer requisições a servidores

Interação com o DOM

O **DOM (Document Object Model)** é uma representação em árvore de todos os elementos HTML da página. O JS pode acessar, modificar e reagir a esse DOM em tempo real.

Exemplo:

```
document.body.style.backgroundColor = "lightblue";
```



Demonstração prática

Abra o **console do navegador** e digite:

```
document.querySelector('h1').textContent = 'Texto alterado via JS!';
```



Resumo

- O navegador executa JS através de uma **engine** como a V8.
 - O JS interage com a página por meio do **DOM**.
 - A execução é **interpretada em tempo real**.
 - O código JS pode ser escrito em **tags** `<script>`, arquivos `.js`, ou diretamente no **console do navegador**.
-

Onde escrever JavaScript

O JavaScript pode ser escrito e executado de várias formas, dependendo do objetivo. Vamos ver as principais maneiras:

1. No console do navegador (para testes rápidos)


 Onde encontrar:

- Clique com o botão direito na página → **"Inspecionar"** → Aba **Console**
- Ou pressione **F12** (Windows/Linux) ou **Cmd + Option + I** (Mac)

 Exemplo:

```
console.log("Olá do console!");
```

 **Vantagem:** rápido para testar trechos de código.

 **Limitação:** código não fica salvo nem afeta o HTML diretamente (a menos que você use DOM).

✅ 2. Dentro de uma tag `<script>` no HTML

🔧 Exemplo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <title>Exemplo com Script Interno</title>
</head>
<body>
  <h1>Olá</h1>

  <script>
    document.querySelector("h1").textContent = "Olá com JS!";
  </script>
</body>
</html>
```

✅ **Vantagem:** simples para projetos pequenos ou aprendizado inicial.

❌ **Limitação:** código JS fica misturado com o HTML.

✅ 3. Em arquivos JavaScript externos (`.js`)

📁 Exemplo:

Crie um arquivo chamado `script.js` com este conteúdo:

```
console.log("JS externo funcionando!");
```

E no HTML:

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

✓ Vantagem:

- Melhor organização
- Reutilização de código
- Manutenção mais fácil



Importante:

A tag `<script src="...">` deve vir **no final do <body>** ou com `defer` no `<head>`, para garantir que o HTML esteja carregado antes da execução do JS.

✓ 4. Em editores de código (como o VS Code)

- **VS Code** é uma das melhores ferramentas para escrever JavaScript.
- Suporte a **syntax highlight**, **formatação automática**, **plugins**, **Live Server** etc.



Recomendado instalar:

- [Live Server Extension](#): recarrega a página automaticamente quando você salva.

Resumo prático

Local		Ideal para
Console navegador	do	Testes rápidos, aprendizado
<code><script></code> interno		Projetos simples
Arquivo externo	<code>.js</code>	Organização profissional
Editor como VS Code	VS	Desenvolvimento real/prático

Módulo 2: Sintaxe Básica

Variáveis

O que são variáveis?

Variáveis são **espaços na memória** onde você pode **armazenar dados** para usar e manipular depois.

Pense como uma **caixa com um nome**, onde você guarda um valor.

Formas de declarar variáveis em JavaScript

var

- Antiga forma de declarar variáveis.
- Tem **escopo de função**, não respeita blocos (`{ }`).
- Pode ser **redeclarada**.

```
var nome = "Ana";  
var nome = "Maria"; // permitido
```


let

- Introduzida no ES6 (2015).
- Tem **escopo de bloco**.
- Pode ser alterada, mas **não redeclarada no mesmo escopo**.

```
let idade = 25;  
idade = 26; // permitido
```

const

- Também do ES6.
- Tem **escopo de bloco**.
- **Não pode ser reatribuída** (imutável).

```
const cidade = "São Paulo";  
// cidade = "Rio";  erro
```

Resumo das diferenças

Tip o	Esco po	Reatribui ção	Redeclara ção
<code>var</code>	Funç ão	Sim	Sim
<code>let</code>	Bloco	Sim	Não
<code>const</code>	Bloco	Não	Não

Boas práticas

- Evite `var`: tem comportamento confuso em blocos.
 - Use `const` sempre que possível.
 - Use `let` quando o valor for mudar.
-

Exemplos práticos

```
let nome = "Carlos";  
const idade = 30;
```

```
console.log(nome, idade);
```

```
nome = "João"; // ok  
// idade = 31; // erro
```

```
if (true) {  
  let dentroDoBloco = "visível aqui";  
  console.log(dentroDoBloco);  
}
```

```
}  
// console.log(dentroDoBloco); // erro
```

Atividade prática

Abra o console do navegador ou um arquivo JS e declare:

1. Uma variável `let` chamada `nota` com o valor `9.5`.
 2. Uma `const` chamada `aprovado` com o valor `true`.
 3. Tente alterar `aprovado` e observe o erro.
-
-

Tipos de Dados

Em JavaScript, os **tipos de dados** determinam o tipo de valor que uma variável pode armazenar. A linguagem é **dinamicamente tipada**, ou seja, você não precisa declarar o tipo — o JavaScript identifica automaticamente.

Tipos Primitivos

São imutáveis e armazenam um único valor.

1. `string` – Texto

```
let nome = "Ana";  
let frase = 'Olá, mundo!';
```

2. **number** – Números inteiros ou decimais

```
let idade = 30;  
let altura = 1.75;
```

3. **boolean** – Verdadeiro ou falso

```
let aprovado = true;  
let reprovado = false;
```

4. **undefined** – Variável declarada, mas sem valor

```
let nota;  
console.log(nota); // undefined
```

5. **null** – Intencionalmente sem valor

```
let desconto = null;
```

6. **symbol** – Identificador único (uso mais avançado)

```
const id = Symbol("id");
```

Tipos de Referência (Estruturados)

1. **object** – Conjunto de pares chave:valor

```
let pessoa = {  
  nome: "Carlos",  
  idade: 28  
};
```

2. **array** – Lista ordenada de valores

```
let frutas = ["maçã", "banana", "uva"];
```

Arrays **também são objetos**, mas com propriedades especiais.

typeof: identificando tipos

Você pode usar **typeof** para saber o tipo de uma variável:

```
console.log(typeof "Olá");    // string
console.log(typeof 42);       // number
console.log(typeof true);     // boolean
console.log(typeof null);     // ⚠ retorna 'object' (bug histórico do JS)
console.log(typeof undefined); // undefined
console.log(typeof {});       // object
console.log(typeof []);       // object
```

Atenção especial ao **null**

`typeof null === "object" // true` (isso é um bug conhecido do JS)

Exemplo prático

```
let nome = "Julia";    // string
let idade = 22;        // number
let ativo = true;      // boolean
let endereco;         // undefined
let curso = null;      // null
let simbolo = Symbol(); // symbol
```

```
let aluno = { nome, idade };
let notas = [8, 9.5, 10];
```



Atividade prática

1. Crie variáveis com cada tipo de dado mencionado.
 2. Use `typeof` para verificar os tipos.
 3. Teste no console ou no seu arquivo `.js`.
-
-



Operadores em JavaScript

Operadores são símbolos que realizam operações em valores (variáveis ou literais). Em JavaScript, eles estão divididos em grupos:



1. Operadores Aritméticos

Usados para realizar cálculos matemáticos.

Operador	Descrição	Exemplo
+	Adição	$5 + 3 \rightarrow 8$
-	Subtração	$5 - 2 \rightarrow 3$
*	Multiplicação	$4 * 2 \rightarrow 8$

/	Divisão	8 / 2 → 4
%	Resto da divisão	10 % 3 → 1
**	Exponenciação	2 ** 3 → 8
++	Incremento	x++ → x + 1
--	Decremento	x-- → x - 1

2. Operadores de Comparação

Comparam dois valores e retornam `true` ou `false`.

Operador	Significado	Exemplo
<code>==</code>	Igual (valor)	"5" == 5 → <code>true</code>
<code>===</code>	Igual (valor e tipo)	"5" === 5 → <code>false</code>
<code>!=</code>	Diferente (valor)	"5" != 5 → <code>false</code>
<code>!==</code>	Diferente (valor e tipo)	"5" !== 5 → <code>true</code>
<code>></code>	Maior que	7 > 5 → <code>true</code>

<	Menor que	3 < 1 → false
>=	Maior ou igual	4 >= 4 → true
<=	Menor ou igual	2 <= 1 → false

⚠ Use **===** e **!==** sempre que possível, pois evitam conversões implícitas de tipo.

3. Operadores Lógicos

Trabalham com valores booleanos (**true** ou **false**).

Operador	Nome	Exemplo	Resultado
&&	E (AND)	true && false false	
,		,	Ou (OR)
!	Não (NOT)	!true	false



4. Operadores de Atribuição

Usados para atribuir valores a variáveis.

Operador	Exemplo	Equivalente
=	$x = 10$	Atribuição direta
+=	$x += 5$	$x = x + 5$
-=	$x -= 2$	$x = x - 2$
*=	$x *= 3$	$x = x * 3$
/=	$x /= 2$	$x = x / 2$
%=	$x \% 2$	$x = x \% 2$

Exemplo prático

```
let a = 10;
```

```
let b = 3;
```

```
console.log(a + b); // 13
```

```
console.log(a > b); // true
```

```
console.log(a === 10); // true
```

```
console.log(!(a < b)); // true
```

```
a += 5; // a agora é 15
```

```
console.log(a); // 15
```

Atividade prática

1. Crie duas variáveis com valores numéricos.
 2. Teste os operadores aritméticos entre elas.
 3. Use `if` com operadores de comparação.
 4. Combine operadores lógicos para validar múltiplas condições.
-

✓ Módulo 3: Controle de Fluxo

📌 Condicionais

Condicionais permitem que você execute diferentes blocos de código dependendo do **valor** ou **condição** em questão. Elas são fundamentais para controlar o comportamento do programa com base em decisões lógicas.

🟡 Estrutura do **if** e **else**

A estrutura mais básica de uma condicional em JavaScript é o **if**:

Sintaxe:

```
if (condição) {  
  // Código executado se a condição for verdadeira  
} else {  
  // Código executado se a condição for falsa  
}
```

Exemplo:

```
let idade = 18;  
  
if (idade >= 18) {  
  console.log("Você é maior de idade.");  
} else {  
  console.log("Você é menor de idade.");  
}
```

else if – Testar múltiplas condições

Se você tiver mais de duas possibilidades, pode usar **else if** para testar várias condições:

Sintaxe:

```
if (condição1) {  
    // Código executado se condição1 for verdadeira  
} else if (condição2) {  
    // Código executado se condição2 for verdadeira  
} else {  
    // Código executado se nenhuma das condições anteriores for verdadeira  
}
```

Exemplo:

```
let nota = 7;  
  
if (nota >= 9) {  
    console.log("Aprovado com distinção");  
} else if (nota >= 6) {  
    console.log("Aprovado");  
} else {  
    console.log("Reprovado");  
}
```

● Operadores Lógicos nas Condicionais

Você pode usar **operadores lógicos** (**&&**, **||**, **!**) para combinar múltiplas condições em uma única verificação.

Exemplo com **&&** (E lógico):

```
let idade = 20;
let temCarteiraDeIdentidade = true;

if (idade >= 18 && temCarteiraDeIdentidade) {
  console.log("Você pode votar.");
} else {
  console.log("Você não pode votar.");
}
```

Exemplo com **||** (OU lógico):

```
let idade = 16;
let temResponsavel = true;

if (idade >= 18 || temResponsavel) {
  console.log("Você pode fazer o cadastro.");
} else {
  console.log("Você não pode fazer o cadastro.");
}
```

Exemplo com **!** (NÃO lógico):

```
let temCarro = false;

if (!temCarro) {
  console.log("Você precisa de um carro.");
} else {
  console.log("Você tem um carro.");
}
```

O uso do **switch**

O **switch** é uma alternativa ao **if** quando você tem muitas condições a verificar, especialmente quando a condição envolve valores discretos (como valores fixos).

Sintaxe:

```
switch (expressão) {  
  case valor1:  
    // Código executado se a expressão for igual a valor1  
    break;  
  case valor2:  
    // Código executado se a expressão for igual a valor2  
    break;  
  default:  
    // Código executado se nenhum dos casos for verdadeiro  
}
```

Exemplo:

```
let dia = 3;  
  
switch (dia) {  
  case 1:  
    console.log("Domingo");  
    break;  
  case 2:  
    console.log("Segunda-feira");  
    break;  
  case 3:  
    console.log("Terça-feira");  
    break;  
  default:  
    console.log("Dia inválido");  
}
```

Exemplo prático

```
let numero = 10;
```

```
if (numero % 2 === 0) {  
  console.log("Número par");  
} else {  
  console.log("Número ímpar");  
}
```

```
let idade = 17;
```

```
if (idade >= 18) {  
  console.log("Você pode entrar na festa!");  
} else {  
  console.log("Você não pode entrar na festa.");  
}
```

```
let cor = "azul";
```

```
switch (cor) {  
  case "vermelho":  
    console.log("Cor escolhida: Vermelho");  
    break;  
  case "azul":  
    console.log("Cor escolhida: Azul");  
    break;  
  case "verde":  
    console.log("Cor escolhida: Verde");  
    break;  
  default:  
    console.log("Cor inválida");  
}
```



Atividade prática

1. Crie uma variável de idade e utilize uma condicional para verificar se a pessoa é maior de idade.
 2. Utilize um `switch` para verificar o dia da semana, dado um número de 1 a 7.
 3. Use operadores lógicos para validar múltiplas condições, como idade e se a pessoa tem autorização.
-
-



Loops (Laços de Repetição)

Loops permitem que você execute um bloco de código várias vezes, com base em uma condição. Eles são úteis quando você precisa repetir tarefas, como percorrer listas ou executar uma ação repetidamente.



O `for` loop – Repetição com número definido de vezes

O `for` é utilizado quando você sabe **quantas vezes** deseja repetir o código.

Sintaxe:

```
for (inicialização; condição; incremento) {  
    // Código a ser repetido  
}
```

Exemplo:

```
for (let i = 0; i < 5; i++) {  
  console.log(i); // Vai imprimir de 0 a 4  
}
```

Explicação:

- **inicialização**: define a variável inicial (**i = 0**).
- **condição**: enquanto **i < 5**, o loop continua.
- **incremento**: após cada iteração, **i** aumenta de 1 (**i++**).

● O **while** loop – Repetição enquanto a condição for verdadeira

O **while** é utilizado quando você não sabe exatamente quantas vezes o código vai ser repetido, mas sabe a condição que deve ser atendida para continuar.

Sintaxe:

```
while (condição) {  
  // Código a ser repetido enquanto a condição for verdadeira  
}
```

Exemplo:

```
let i = 0;

while (i < 5) {

  console.log(i); // Vai imprimir de 0 a 4

  i++;

}
```

O **do...while** loop – Executa o código pelo menos uma vez

O **do...while** é semelhante ao **while**, mas a condição é verificada **após** a execução do código, garantindo que o código seja executado pelo menos uma vez.

Sintaxe:

```
do {

  // Código a ser repetido

} while (condição);
```

Exemplo:

```
let i = 0;

do {

  console.log(i); // Vai imprimir de 0 a 4

  i++;

} while (i < 5);
```

● O **for...of** loop – Percorrendo elementos de um array

O **for...of** é utilizado para percorrer os **elementos de um array** ou outro objeto iterável (como strings).

Sintaxe:

```
for (let item of array) {  
    // Código a ser repetido para cada item  
}
```

Exemplo:

```
let frutas = ["maçã", "banana", "laranja"];  
for (let fruta of frutas) {  
    console.log(fruta); // Vai imprimir "maçã", "banana", "laranja"  
}
```

● O **for...in** loop – Percorrendo as propriedades de um objeto

O **for...in** é utilizado para percorrer as **propriedades** de um **objeto**.

Sintaxe:

```
for (let chave in objeto) {  
    // Código a ser repetido para cada chave do objeto  
}
```

Exemplo:

```
let pessoa = {
```

```
  nome: "Carlos",
```

```
  idade: 28,
```

```
  cidade: "São Paulo"
```

```
};
```

```
for (let chave in pessoa) {
```

```
  console.log(chave + ": " + pessoa[chave]); // Vai imprimir "nome: Carlos", "idade: 28", "cidade: São Paulo"
```

```
}
```



Exemplo prático

```
// Usando o 'for'
```

```
for (let i = 0; i < 3; i++) {
```

```
  console.log("Número: " + i);
```

```
}
```

```
// Usando o 'while'
```

```
let i = 0;
```

```
while (i < 3) {
```

```
  console.log("Número (while): " + i);
```

```
  i++;
```

```
}
```

```
// Usando o 'do...while'
```

```
let j = 0;
```

```
do {
```

```
  console.log("Número (do...while): " + j);
```

```
  j++;
```

```
} while (j < 3);
```

```
// Usando o 'for...of'
```

```
let animais = ["cachorro", "gato", "passarinho"];
```

```
for (let animal of animais) {
```

```
  console.log("Animal: " + animal);
```

```
}
```

```
// Usando o 'for...in'
```

```
let carro = {
```

```
  marca: "Ford",
```

```
  modelo: "F-150",
```

```
  ano: 2021
```

```
};
```

```
for (let propriedade in carro) {
```

```
  console.log(propriedade + ": " + carro[propriedade]);
```

```
}
```



Atividade prática

1. Crie um loop `for` para imprimir os números de 1 a 10.
 2. Use um `while` para contar de 1 a 10, mas apenas os números **ímpares**.
 3. Com o `for...of`, percorra um array de frutas e imprima cada fruta em maiúsculas.
 4. Use o `for...in` para percorrer um objeto representando uma pessoa e exibir suas propriedades.
-
-



Módulo 4: Funções (40 min)



Tipos de Função

Em JavaScript, **funções** são blocos de código que realizam uma tarefa específica. Elas podem ser chamadas e executadas em diferentes momentos do seu código, o que torna a reutilização de código mais eficiente e facilita a organização.

Existem alguns **tipos principais de funções** em JavaScript:

● Funções Declarativas (Funções Normais)

As funções declarativas são definidas usando a palavra-chave **function**. Elas podem ser chamadas antes ou depois de serem declaradas devido ao **hoisting** (elevação).

Sintaxe:

```
function nomeDaFuncao(parametro1, parametro2) {  
    // Código a ser executado  
    return resultado; // Opcional, mas permite retornar um valor  
}
```

Exemplo:

```
function soma(a, b) {  
    return a + b;  
}  
  
console.log(soma(5, 3)); // 8
```

● Funções Anônimas

Funções anônimas são funções que não possuem um nome. Elas são frequentemente usadas em expressões, como em callbacks e funções de ordem superior (como **map**, **filter**, etc.).

Sintaxe:

```
const nomeDaFuncao = function(parametro1, parametro2) {  
  // Código a ser executado  
  return resultado;  
};
```

Exemplo:

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

```
console.log(multiplicar(4, 2)); // 8
```

Funções Arrow (Funções de seta)

As **funções arrow** são uma forma mais curta e concisa de escrever funções anônimas. Elas também têm algumas diferenças importantes no comportamento de **escopo** e **this**, mas, em termos de sintaxe, são muito mais compactas.

Sintaxe:

```
const nomeDaFuncao = (parametro1, parametro2) => {  
  // Código a ser executado  
  return resultado;  
};
```

Exemplo:

```
const soma = (a, b) => a + b;
```

```
console.log(soma(5, 3)); // 8
```

Nota: Se a função tiver apenas uma linha de código, o **return** pode ser omitido.

Funções Auto-invocáveis (IIFE)

As funções **auto-invocáveis** são aquelas que são executadas imediatamente após a sua definição. Elas são frequentemente usadas para criar **escopos isolados** e evitar conflitos de variáveis no escopo global.

Sintaxe:

```
(function() {  
    // Código a ser executado imediatamente  
})();
```

Exemplo:

```
(function() {  
    let mensagem = "Olá, mundo!";  
    console.log(mensagem); // "Olá, mundo!"  
})(); // Executa automaticamente
```

Funções Recursivas

Funções recursivas são aquelas que **chamam a si mesmas**. Elas são úteis para resolver problemas que podem ser divididos em subproblemas menores (como na busca em estruturas de dados como árvores).

Sintaxe:

```
function nomeDaFuncao() {  
  if (condiçãoBase) {  
    // Caso base  
    return;  
  } else {  
    nomeDaFuncao(); // Chamada recursiva  
  }  
}
```

Exemplo:

```
function contarDeTrásParaFrente(numero) {  
  if (numero <= 0) {  
    return;  
  } else {  
    console.log(numero);  
    contarDeTrásParaFrente(numero - 1); // Chamada recursiva  
  }  
}  
  
contarDeTrásParaFrente(5); // 5, 4, 3, 2, 1
```

Exemplo prático de todos os tipos de funções

// Função declarativa

```
function saudacao(nome) {  
    return "Olá, " + nome + "!";  
}  
  
console.log(saudacao("Carlos")); // "Olá, Carlos!"
```

// Função anônima

```
const subtrair = function(a, b) {  
    return a - b;  
};  
  
console.log(subtrair(10, 4)); // 6
```

// Função arrow

```
const dividir = (a, b) => a / b;  
  
console.log(dividir(10, 2)); // 5
```

// Função auto-invocável (IIFE)

```
(function() {  
    let resultado = 7 * 3;  
    console.log(resultado); // 21  
})();
```

```
// Função recursiva

function contarAte(numero) {

  if (numero === 0) return;

  console.log(numero);

  contarAte(numero - 1);

}

contarAte(3); // 3, 2, 1
```



Atividade prática

1. Crie uma função declarativa que receba dois números e retorne o maior deles.
 2. Escreva uma função anônima que calcule o quadrado de um número.
 3. Crie uma função **arrow** que recebe uma string e retorna a mesma string em letras maiúsculas.
 4. Faça uma função recursiva que conte de um número até 1, imprimindo cada número.
-

Parâmetros Padrão, Rest e Spread

Em JavaScript, as funções podem receber parâmetros de diversas maneiras. Vamos explorar três características importantes:

Parâmetros Padrão (Default Parameters)

Parâmetros padrão permitem que você defina um valor para um parâmetro caso ele **não seja passado** ou **seja undefined**. Isso evita que você precise verificar manualmente se o valor foi fornecido, tornando o código mais limpo e legível.

Sintaxe:

```
function nomeDaFuncao(parametro1 = valorPadrao) {  
    // Código da função  
}
```

Exemplo:

```
function saudacao(nome = "Visitante") {  
    console.log("Olá, " + nome + "!");  
}  
  
saudacao("Carlos"); // "Olá, Carlos!"  
  
saudacao();         // "Olá, Visitante!"
```

No exemplo acima, se **nome** não for fornecido, o valor **"Visitante"** será usado como padrão.

● Parâmetro Rest (...)

O **parâmetro rest** permite que você passe um número **indefinido** de argumentos para uma função, que serão tratados como um **array**. Ele é útil quando você não sabe de antemão quantos argumentos serão passados para a função.

Sintaxe:

```
function nomeDaFuncao(...parametros) {  
    // Código que utiliza os parâmetros como um array  
}
```

Exemplo:

```
function somar(...numeros) {  
    return numeros.reduce((soma, num) => soma + num, 0);  
}
```

```
console.log(somar(1, 2, 3)); // 6
```

```
console.log(somar(5, 10, 20, 30)); // 65
```

Neste exemplo, **...numeros** coleta todos os valores passados para a função e os transforma em um array, permitindo que você os manipule como um conjunto.

● **Parâmetro Spread (. . .)**

O **parâmetro spread** é usado para **expandir** elementos de um array ou objeto. Ele permite passar os elementos de um array como argumentos de uma função ou copiar/combinar arrays e objetos de maneira eficiente.

Sintaxe (para arrays):

```
const novoArray = [...arrayOriginal];
```

Sintaxe (para objetos):

```
const novoObjeto = {...objetoOriginal};
```

Exemplo (com arrays):

```
let numeros = [1, 2, 3];
```

```
let maisNumeros = [0, ...numeros, 4, 5];
```

```
console.log(maisNumeros); // [0, 1, 2, 3, 4, 5]
```

Exemplo (com objetos):

```
let pessoa = { nome: "Carlos", idade: 28 };
```

```
let endereco = { cidade: "São Paulo", estado: "SP" };
```

```
let pessoaCompleta = { ...pessoa, ...endereco };
```

```
console.log(pessoaCompleta);
```



```
// { nome: "Carlos", idade: 28, cidade: "São Paulo", estado: "SP" }
```

O **spread** facilita a combinação de objetos e arrays sem modificar os originais.

Exemplo prático com os três tipos de parâmetros

// Parâmetros padrão

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

```
saudacao(); // "Olá, Visitante!"
```

```
saudacao("Carlos"); // "Olá, Carlos!"
```

// Parâmetro Rest

```
function mostrarNumeros(...numeros) {  
    console.log(numeros); // Exibe os números como um array  
}
```

```
mostrarNumeros(1, 2, 3, 4); // [1, 2, 3, 4]
```

// Parâmetro Spread

```
let frutas = ["maçã", "banana", "laranja"];
```

```
let maisFrutas = ["melancia", ...frutas, "morango"];
```

```
console.log(maisFrutas); // ["melancia", "maçã", "banana", "laranja",  
"morango"]
```



Atividade prática

1. Crie uma função que tenha um parâmetro padrão para o nome e uma mensagem. Se a mensagem não for fornecida, use um valor padrão como "Bem-vindo!".
 2. Crie uma função que receba vários números com o parâmetro rest e calcule a média deles.
 3. Utilize o spread para combinar dois arrays de frutas e imprimir o resultado.
 4. Crie um objeto de uma pessoa com nome e idade e depois crie um novo objeto, adicionando a cidade usando o spread.
-

Escopo e Closures

Escopo (Scope)

Escopo refere-se ao **contexto** em que uma variável ou função é definida e acessada. Em JavaScript, o escopo determina a **visibilidade** e o **tempo de vida** de variáveis e funções.

Existem dois tipos principais de escopo:

1. Escopo Global

Quando uma variável ou função é definida fora de qualquer função, ela tem **escopo global**, o que significa que pode ser acessada de qualquer lugar no código.

```
let nome = "Carlos"; // Escopo global
```

```
function saudacao() {  
    console.log(nome); // Acessando a variável global 'nome'  
}
```

```
saudacao(); // "Carlos"
```

2. Escopo Local

Variáveis e funções definidas dentro de uma função possuem **escopo local**, ou seja, só podem ser acessadas dentro dessa função.

```
function saudacao() {  
    let nome = "Carlos"; // Escopo local
```

```
console.log(nome); // "Carlos"
}
```

```
saudacao();
```

```
// console.log(nome); // Erro: 'nome' is not defined
```

No exemplo acima, a variável **nome** é definida dentro da função **saudacao()** e não pode ser acessada fora dela.

Escopo de Bloco

Introduzido no ES6, o **let** e o **const** têm escopo de bloco, ou seja, são acessíveis apenas dentro do bloco onde foram declarados (como em loops ou condicionais).

```
if (true) {
  let idade = 28; // Escopo de bloco
  console.log(idade); // "28"
}
```

```
// console.log(idade); // Erro: 'idade' is not defined
```

Hoisting

Hoisting é um comportamento do JavaScript em que **declarações de variáveis e funções** são "elevadas" para o topo do seu escopo de execução, antes de o código ser executado. Isso se aplica apenas a **funções declarativas** e **variáveis declaradas com `var`**.

Exemplo com função declarativa:

```
saudacao(); // "Olá"
```

```
function saudacao() {  
  console.log("Olá");  
}
```

Exemplo com `var`:

```
console.log(nome); // undefined (não erro!)
```

```
var nome = "Carlos";
```

Note que, enquanto a variável `nome` é elevada ao topo, seu valor é atribuído apenas quando o código realmente passa por aquela linha. Portanto, ao tentar acessá-la antes da atribuição, o valor é `undefined`.

Closures

Closures (ou **fechamentos**) ocorrem quando uma função **lembra** e tem acesso a variáveis do seu **escopo léxico**, mesmo após o término da execução da função externa. Em outras palavras, uma closure é criada quando uma função interna **acessa** variáveis da função externa, mesmo depois de a função externa ter terminado sua execução.

Sintaxe:

```
function externa() {  
  
    let nome = "Carlos"; // Variável no escopo externo  
  
    return function interna() {  
  
        console.log(nome); // Acessando a variável do escopo externo  
  
    };  
}  
  
const closure = externa(); // A função interna é "lembrada"  
closure(); // "Carlos"
```

Neste exemplo:

- A função **interna** é uma closure porque ela tem acesso à variável **nome** da função **externa**, mesmo depois que **externa()** foi executada.

Exemplo com contador:

Closures são comumente usadas para criar **funções com estado**, como um contador:

```
function contador() {  
  let count = 0; // variável dentro do escopo da função externa  
  return function() {  
    count++; // A função interna tem acesso à variável 'count'  
    return count;  
  };  
}
```

```
const contador1 = contador();  
console.log(contador1()); // 1  
console.log(contador1()); // 2  
console.log(contador1()); // 3
```

A função `contador` retorna uma função que continua "lembrando" e acessando a variável `count`, mesmo após `contador` ter terminado sua execução.

● Importância das Closures

- **Privacidade de dados:** Elas permitem que você crie variáveis privadas (inacessíveis de fora da função).
- **Funções de ordem superior:** Elas são frequentemente usadas em funções que retornam outras funções, como `map()`, `filter()`, e `reduce()`.
- **Funções com estado:** Como vimos no exemplo do contador, closures ajudam a criar funções que mantêm o estado entre as execuções.

Exemplo prático de Escopo e Closures

// Escopo global

let nome = "Carlos"; // Acessível globalmente

function saudacao() {

let saudacao = "Olá"; // Escopo local

console.log(saudacao + ", " + nome); // Acessa 'nome' do escopo global

}

saudacao(); // "Olá, Carlos"

// Closure

function criaMultiplicador(fator) {


```
return function(numero) {  
    return numero * fator; // Acessa 'fator' da função externa  
};  
}
```

```
const multiplicarPorDois = criaMultiplicador(2);  
console.log(multiplicarPorDois(5)); // 10
```

```
const multiplicarPorTres = criaMultiplicador(3);  
console.log(multiplicarPorTres(5)); // 15
```



Atividade prática

1. Crie uma função que defina uma variável no escopo local e outra no escopo global, e tente acessar ambas a partir de diferentes partes do código.
 2. Escreva uma função que retorna outra função, utilizando **closures** para manter o estado entre as execuções.
 3. Use o conceito de **hoisting** para declarar uma variável com **var** e uma função, e observe como o comportamento de cada um é diferente.
 4. Crie um exemplo de **closure** que conta quantas vezes uma função é chamada.
-



Módulo 5: Objetos e Arrays



Objetos

Em JavaScript, **objetos** são estruturas que permitem armazenar dados na forma de **pares de chave-valor**. Eles são extremamente úteis para organizar e representar dados mais complexos e podem conter qualquer tipo de valor, como strings, números, arrays, funções e até outros objetos.



Criando Objetos

Existem duas formas principais de criar objetos em JavaScript:

1. Notação Literal

A maneira mais comum e direta de criar objetos é usando a notação literal, onde você define o objeto com chaves `{ }` e os pares chave-valor separados por dois pontos `:`.

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28,  
  profissao: "Desenvolvedor"  
};
```

```
console.log(pessoa);
```

2. Usando a palavra-chave `new Object()`

Você também pode criar um objeto utilizando o construtor `Object()`, mas a notação literal é mais utilizada por ser mais concisa.

```
const pessoa = new Object();  
  
pessoa.nome = "Carlos";  
  
pessoa.idade = 28;  
  
pessoa.profissao = "Desenvolvedor";  
  
  
console.log(pessoa);
```

Acessando Propriedades do Objeto

As propriedades de um objeto podem ser acessadas de duas formas:

1. Notação de Ponto

A maneira mais comum e intuitiva de acessar uma propriedade de um objeto.

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28  
};  
  
  
console.log(pessoa.nome); // "Carlos"  
  
console.log(pessoa.idade); // 28
```

2. Notação de Colchetes

A notação de colchetes permite acessar uma propriedade usando uma string como chave. Ela é útil quando o nome da propriedade é armazenado em uma variável ou contém caracteres especiais.

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28  
};
```

```
console.log(pessoa["nome"]); // "Carlos"
```

```
console.log(pessoa["idade"]); // 28
```

Adicionando e Modificando Propriedades

Você pode adicionar novas propriedades ou modificar as existentes de um objeto facilmente:

Exemplo de adição e modificação:

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28  
};
```

```
// Adicionando uma nova propriedade
```

```
pessoa.profissao = "Desenvolvedor";
```

```
// Modificando uma propriedade existente
```

```
pessoa.idade = 29;
```

```
console.log(pessoa);
```

```
// { nome: "Carlos", idade: 29, profissao: "Desenvolvedor" }
```

Deletando Propriedades

Para excluir uma propriedade de um objeto, usamos a palavra-chave **delete**.

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28,  
  profissao: "Desenvolvedor"  
};  
  
delete pessoa.profissao;  
  
console.log(pessoa);  
// { nome: "Carlos", idade: 28 }
```

● Métodos de Objetos

Objetos também podem conter **métodos**, que são funções associadas a um objeto. Esses métodos podem acessar e modificar as propriedades do próprio objeto.

Exemplo de método:

```
const pessoa = {  
  nome: "Carlos",  
  idade: 28,  
  saudacao: function() {  
    console.log("Olá, meu nome é " + this.nome);  
  }  
};  
  
pessoa.saudacao(); // "Olá, meu nome é Carlos"
```

No exemplo acima, o método `saudacao` usa a palavra-chave `this` para se referir ao próprio objeto `pessoa`.

● Objetos Aninhados

Você também pode ter objetos dentro de objetos. Isso é útil para representar dados mais complexos, como um endereço de uma pessoa ou um produto com várias características.

Exemplo de objeto aninhado:

```
const pessoa = {
```

```
nome: "Carlos",
endereço: {
  rua: "Rua A",
  número: 100,
  cidade: "São Paulo"
}
};

console.log(pessoa.endereço.cidade); // "São Paulo"
```

Desestruturando Objetos (Destructuring)

A **desestruturação** de objetos permite extrair valores das propriedades de um objeto e atribuí-los a variáveis de forma concisa.

Exemplo de desestruturação:

```
const pessoa = {
  nome: "Carlos",
  idade: 28,
  profissão: "Desenvolvedor"
};

// Desestruturando as propriedades do objeto
const { nome, idade } = pessoa;
```



```
console.log(nome); // "Carlos"
```

```
console.log(idade); // 28
```

Você também pode renomear as variáveis durante a desestruturação, caso as variáveis não tenham o mesmo nome da propriedade:

```
const pessoa = {
```

```
  nome: "Carlos",
```

```
  idade: 28
```

```
};
```

```
const { nome: nomePessoa, idade: idadePessoa } = pessoa;
```

```
console.log(nomePessoa); // "Carlos"
```

```
console.log(idadePessoa); // 28
```



Exemplo prático de objetos

```
// Criando um objeto de pessoa
```

```
const pessoa = {
```

```
  nome: "Carlos",
```

```
  idade: 28,
```

```
  profissao: "Desenvolvedor",
```

```
saudacao: function() {  
    console.log("Olá, meu nome é " + this.nome);  
}  
};
```

// Acessando as propriedades do objeto

```
console.log(pessoa.nome); // "Carlos"
```

```
console.log(pessoa["profissao"]); // "Desenvolvedor"
```

// Adicionando uma nova propriedade

```
pessoa.cidade = "São Paulo";
```

// Modificando uma propriedade

```
pessoa.idade = 29;
```

// Chamando um método

```
pessoa.saudacao(); // "Olá, meu nome é Carlos"
```

// Deletando uma propriedade

```
delete pessoa.profissao;
```

```
console.log(pessoa); // { nome: "Carlos", idade: 29, cidade: "São Paulo"  
}
```

```
// Desestruturando o objeto  
  
const { nome, idade } = pessoa;  
  
console.log(nome, idade); // "Carlos", 29
```



Atividade prática

1. Crie um objeto `carro` com propriedades como `marca`, `modelo`, `ano` e `cor`. Acesse essas propriedades e altere o valor de algumas delas.
 2. Crie um objeto `livro` com propriedades como `titulo`, `autor` e `anoPublicacao`. Em seguida, adicione uma nova propriedade chamada `genero` e modifique o `anoPublicacao`.
 3. Utilize a desestruturação para extrair o título e o autor de um livro e imprima essas informações no console.
 4. Crie um objeto `endereco` dentro do objeto `pessoa` que tenha as propriedades `rua`, `bairro`, `cidade` e `cep`. Acesse e imprima o nome da rua e o nome do bairro.
-
-

Arrays

Arrays em JavaScript são estruturas de dados que permitem armazenar uma **lista ordenada de elementos**. Esses elementos podem ser de qualquer tipo, como números, strings, objetos e até outros arrays. Eles são uma das ferramentas mais poderosas em JavaScript para trabalhar com coleções de dados.

Criando Arrays

Você pode criar arrays de diferentes maneiras:

1. Notação Literal

A forma mais comum de criar um array é usando colchetes `[]`:

```
const frutas = ["maçã", "banana", "laranja"];  
console.log(frutas); // ["maçã", "banana", "laranja"]
```

2. Usando o construtor `new Array()`

Embora a notação literal seja mais usada, você pode criar um array utilizando o construtor `Array()`:

```
const frutas = new Array("maçã", "banana", "laranja");  
console.log(frutas); // ["maçã", "banana", "laranja"]
```

● Acessando e Modificando Elementos

Você pode acessar os elementos de um array através de seus **índices**, que começam de 0 (o primeiro elemento tem índice 0, o segundo tem índice 1 e assim por diante).

Exemplo de acesso:

```
const frutas = ["maçã", "banana", "laranja"];
```

```
console.log(frutas[0]); // "maçã"
```

```
console.log(frutas[2]); // "laranja"
```

Exemplo de modificação:

```
frutas[1] = "abacaxi"; // Alterando "banana" para "abacaxi"
```

```
console.log(frutas); // ["maçã", "abacaxi", "laranja"]
```

● Propriedades e Métodos dos Arrays

1. **length**

O **length** retorna o número de elementos em um array.

```
const frutas = ["maçã", "banana", "laranja"];
```

```
console.log(frutas.length); // 3
```

2. **push()**

O método **push()** adiciona um ou mais elementos ao final do array.

```
const frutas = ["maçã", "banana"];
frutas.push("laranja");
console.log(frutas); // ["maçã", "banana", "laranja"]
```

3. **pop()**

O método **pop()** remove o último elemento do array.

```
const frutas = ["maçã", "banana", "laranja"];
frutas.pop();
console.log(frutas); // ["maçã", "banana"]
```

4. **shift()**

O método **shift()** remove o primeiro elemento do array.

```
const frutas = ["maçã", "banana", "laranja"];
frutas.shift();
console.log(frutas); // ["banana", "laranja"]
```

5. **unshift()**

O método **unshift()** adiciona um ou mais elementos ao início do array.

```
const frutas = ["banana", "laranja"];
frutas.unshift("maçã");
console.log(frutas); // ["maçã", "banana", "laranja"]
```

● Métodos Importantes para Manipulação de Arrays

1. `map()`

O método `map()` cria um novo array com os resultados de aplicar uma função a cada elemento do array original.

```
const numeros = [1, 2, 3, 4];  
  
const quadrados = numeros.map(num => num * num);
```

```
console.log(quadrados); // [1, 4, 9, 16]
```

2. `filter()`

O método `filter()` cria um novo array contendo todos os elementos que passam em um teste.

```
const numeros = [1, 2, 3, 4, 5];  
  
const pares = numeros.filter(num => num % 2 === 0);
```

```
console.log(pares); // [2, 4]
```

3. `reduce()`

O método `reduce()` aplica uma função em cada elemento do array (da esquerda para a direita) para reduzir a lista a um único valor, como uma soma ou produto.

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

```
const soma = numeros.reduce((acumulador, num) => acumulador + num, 0);
```

```
console.log(soma); // 10
```

4. **forEach()**

O método **forEach()** executa uma função para cada elemento do array, mas **não retorna um novo array**.

```
const frutas = ["maçã", "banana", "laranja"];
```

```
frutas.forEach(fruta => {
```

```
    console.log(fruta);
```

```
});
```

```
// "maçã"
```

```
// "banana"
```

```
// "laranja"
```

Acessando Arrays Multidimensionais

Arrays também podem conter outros arrays dentro deles, formando uma estrutura de **arrays multidimensionais**.

```
const matriz = [
```

```
    [1, 2, 3],
```

```
    [4, 5, 6],
```

```
    [7, 8, 9]
```



```
];
```

```
console.log(matriz[0][0]); // 1
```

```
console.log(matriz[1][2]); // 6
```

Desestruturação de Arrays (Destructuring)

A desestruturação de arrays permite extrair valores de um array e atribuí-los a variáveis de forma concisa.

Exemplo de desestruturação de arrays:

```
const frutas = ["maçã", "banana", "laranja"];
```

```
const [primeira, segunda] = frutas;
```

```
console.log(primeira); // "maçã"
```

```
console.log(segunda); // "banana"
```

Você também pode usar desestruturação para ignorar alguns valores ou para pegar o restante dos elementos.

```
const frutas = ["maçã", "banana", "laranja", "abacaxi"];
```

```
const [primeira, , terceira] = frutas;
```

```
console.log(primeira); // "maçã"
```

```
console.log(terceira); // "laranja"
```

Exemplo prático de arrays

// Criando e manipulando um array

```
const frutas = ["maçã", "banana", "laranja"];
```

// Acessando elementos

```
console.log(frutas[0]); // "maçã"
```

```
console.log(frutas[2]); // "laranja"
```

// Modificando elementos

```
frutas[1] = "morango";
```

```
console.log(frutas); // ["maçã", "morango", "laranja"]
```

// Usando métodos

```
frutas.push("abacaxi");
```

```
console.log(frutas); // ["maçã", "morango", "laranja", "abacaxi"]
```

```
frutas.pop();
```

```
console.log(frutas); // ["maçã", "morango", "laranja"]
```

// Desestruturação de arrays

```
const [primeiraFruta, segundaFruta] = frutas;
```

```
console.log(primeiraFruta); // "maçã"
```

```
console.log(segundaFruta); // "morango"
```

```
// Usando map() para criar um novo array
```

```
const frutasMaiusculas = frutas.map(fruta => fruta.toUpperCase());
```

```
console.log(frutasMaiusculas); // ["MAÇÃ", "MORANGO", "LARANJA"]
```



Atividade prática

1. Crie um array de números e utilize o método `map()` para criar um novo array com o quadrado de cada número.
 2. Crie um array de palavras e use o método `filter()` para selecionar apenas as palavras que têm mais de 5 letras.
 3. Crie um array de objetos representando livros. Cada livro deve ter `titulo`, `autor` e `anoPublicacao`. Use `map()` para criar um novo array contendo apenas os títulos dos livros.
 4. Utilize a desestruturação para acessar o segundo e o quarto elemento de um array de frutas.
-



Módulo 6: DOM – Document Object Model



Selecionar Elementos

O **DOM** (Document Object Model) é uma interface de programação para documentos HTML e XML. Ele representa a página web como uma estrutura de árvore, onde cada elemento HTML é um nó. Com o DOM, você pode acessar, modificar e interagir com o conteúdo da página web.

A primeira coisa que você precisa saber ao trabalhar com o DOM é **como selecionar os elementos** que você deseja manipular.



Métodos para Seleção de Elementos

1. `getElementById()`

Esse método seleciona um único elemento com base no **ID**. O ID deve ser único dentro do documento.

```
<!-- HTML -->
```

```
<div id="exemplo">Este é um exemplo de ID</div>
```

```
// Selecionando o elemento pelo ID
```

```
const elemento = document.getElementById("exemplo");
```

```
console.log(elemento); // <div id="exemplo">Este é um exemplo de ID</div>
```

- **Nota:** O ID de um elemento é único, ou seja, você pode ter apenas um elemento com o mesmo ID em toda a página.

2. `getElementsByClassName()`

Esse método seleciona todos os elementos com uma determinada **classe**. Ele retorna uma coleção de elementos (HTMLCollection), o que significa que você pode acessar vários elementos que possuem a mesma classe.

```
<!-- HTML -->
```

```
<p class="mensagem">Mensagem 1</p>
```

```
<p class="mensagem">Mensagem 2</p>
```

```
// Selecionando elementos pela classe
```

```
const mensagens = document.getElementsByClassName("mensagem");
```

```
console.log(mensagens);           //      HTMLCollection      [<p  
class="mensagem">Mensagem          1</p>,      <p  
class="mensagem">Mensagem 2</p>]
```

- **Nota:** `getElementsByClassName()` retorna uma coleção dinâmica, o que significa que, se a classe for modificada no DOM, a coleção também será atualizada automaticamente.

3. `getElementsByTagName()`

Esse método seleciona todos os elementos de uma determinada **tag** HTML. Ele retorna uma coleção de elementos (HTMLCollection), como o método anterior.

```
<!-- HTML -->
```

```
<ul>
```

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

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

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

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

```
const itens = document.getElementsByTagName("li");
```

```
console.log(itens); // HTMLCollection [<li>Item 1</li>, <li>Item 2</li>,
<li>Item 3</li>]
```

4. **querySelector()**

Esse método é um dos mais versáteis e modernos para selecionar elementos, permitindo a seleção de um único elemento usando **seletores CSS**. Ele retorna o **primeiro** elemento que corresponde ao seletor fornecido.

```
<!-- HTML -->
```

```
<div class="caixa">
```

```
<p>Texto dentro da caixa</p>
```

```
</div>
```

```
// Selecionando o primeiro elemento com a classe 'caixa'
```

```
const caixa = document.querySelector(".caixa");
```

```
console.log(caixa); // <div class="caixa">...</div>
```

- **Nota:** O `querySelector()` é muito poderoso porque pode aceitar seletores complexos, como IDs, classes, tags, e até combinações entre eles.

5. `querySelectorAll()`

Esse método é semelhante ao `querySelector()`, mas ao invés de retornar apenas o primeiro elemento encontrado, ele retorna **todos** os elementos que correspondem ao seletor fornecido. O resultado será uma **NodeList** (uma lista de nós).

```
<!-- HTML -->
```

```
<p class="alerta">Alerta 1</p>
```

```
<p class="alerta">Alerta 2</p>
```

```
<p class="alerta">Alerta 3</p>
```

```
// Selecionando todos os elementos com a classe 'alerta'
```

```
const alertas = document.querySelectorAll(".alerta");
```

```
console.log(alertas); // NodeList [<p class="alerta">Alerta 1</p>, <p class="alerta">Alerta 2</p>, <p class="alerta">Alerta 3</p>]
```

- **Nota:** A **NodeList** retornada por `querySelectorAll()` é semelhante a um array, mas nem todos os métodos de array podem ser utilizados diretamente. Porém, você pode usar o método `forEach()` em uma NodeList.
-

● Manipulando Elementos Seleccionados

Uma vez que você seleciona um ou mais elementos do DOM, você pode **manipular suas propriedades**, como conteúdo, atributos, estilo, etc.

1. Alterando o conteúdo de um elemento

Para alterar o conteúdo de um elemento selecionado, você pode usar a propriedade **innerHTML** ou **textContent**.

```
const elemento = document.getElementById("exemplo");  
  
elemento.innerHTML = "Novo conteúdo";
```

- **Nota:** **innerHTML** altera o HTML dentro de um elemento, permitindo adicionar tags HTML. Já **textContent** altera apenas o texto, sem interpretar tags HTML.

2. Alterando atributos de um elemento

Você pode alterar os atributos de um elemento com o método **setAttribute()**.

```
const link = document.querySelector("a");  
  
link.setAttribute("href", "https://www.novosite.com");
```

3. Alterando o estilo de um elemento

Você pode alterar o estilo diretamente através da propriedade **style**.

```
const caixa = document.querySelector(".caixa");  
  
caixa.style.backgroundColor = "blue";  
  
caixa.style.color = "white";
```

🟡 Eventos no DOM

A manipulação do DOM também envolve a **interação com o usuário**, como cliques, passagens de mouse e outros tipos de eventos. Você pode adicionar **event listeners** a elementos para escutar eventos e reagir a eles.

1. `addEventListener()`

O método `addEventListener()` permite que você adicione um ou mais ouvintes de eventos a um elemento. Esse método aceita dois parâmetros: o tipo de evento (como `click`, `mouseover`, etc.) e a função que será chamada quando o evento ocorrer.

```
const botao = document.getElementById("meuBotao");
```

```
botao.addEventListener("click", function() {  
    alert("O botão foi clicado!");  
});
```

🟢 Exemplo Prático de Seleção e Manipulação de Elementos

Vamos criar um exemplo onde o usuário clica em um botão para alterar o conteúdo e o estilo de um elemento.

HTML:

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

```
<head>

  <meta charset="UTF-8">

      <meta      name="viewport"      content="width=device-width,
initial-scale=1.0">

  <title>Exemplo DOM</title>

</head>

<body>

  <div id="mensagem">Clique no botão abaixo para mudar este
texto.</div>

  <button id="meuBotao">Clique aqui!</button>


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

</body>

</html>
```

JavaScript (**script.js**):

```
// Selecionando o botão e o div

const botao = document.getElementById("meuBotao");
const mensagem = document.getElementById("mensagem");


// Adicionando um ouvinte de evento ao botão

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

  // Alterando o conteúdo e o estilo da mensagem
```

```
mensagem.innerHTML = "Você clicou no botão! O conteúdo foi  
alterado.";

mensagem.style.color = "blue";

mensagem.style.fontWeight = "bold";

});
```

Atividade Prática

1. Crie um botão na página e, ao clicar nele, altere o texto de um parágrafo e mude sua cor.
 2. Selecione uma lista de itens (como uma lista de compras) e altere o estilo de cada item (adicione cor de fundo, mude o tamanho da fonte).
 3. Use o método `querySelectorAll()` para selecionar todos os itens de uma lista e adicione um evento de clique em cada um deles, fazendo com que o item seja marcado como "completo" (alterando a cor ou o estilo).
 4. Crie uma página com um campo de texto e um botão. Quando o botão for clicado, altere o conteúdo de um parágrafo com base no que foi digitado no campo de texto.
-

Manipular Conteúdo e Atributos

Depois de selecionar elementos do DOM, é hora de **manipular** o conteúdo e os **atributos** desses elementos. O DOM oferece várias maneiras de alterar o que é exibido na página e modificar as propriedades dos elementos.

Manipulando Conteúdo de um Elemento

1. **innerHTML**

A propriedade **innerHTML** permite acessar ou definir o conteúdo HTML de um elemento. Isso significa que você pode inserir ou alterar tags HTML dentro de um elemento.

// Selecionando o elemento

```
const elemento = document.getElementById("exemplo");
```

// Alterando o conteúdo HTML

```
elemento.innerHTML = "<strong>Texto em negrito!</strong>";
```

- **Nota:** O **innerHTML** interpreta tags HTML. Ao usá-lo, você pode inserir não apenas texto, mas também elementos HTML dentro do seu conteúdo.

2. **textContent**

A propriedade **textContent** é similar ao **innerHTML**, mas ela manipula apenas o **texto** do elemento, sem interpretar tags HTML.

```
const elemento = document.getElementById("exemplo");
```

```
// Alterando o conteúdo de texto
```

```
elemento.textContent = "Este é o novo conteúdo de texto.";
```

- **Nota:** Se você usar `textContent`, as tags HTML serão tratadas como texto e não serão interpretadas como elementos HTML.

3. `innerText`

A propriedade `innerText` é parecida com `textContent`, mas ela também leva em consideração a formatação CSS e pode não retornar todos os textos (como os que estão escondidos).

```
const elemento = document.getElementById("exemplo");
```

```
// Alterando o texto visível
```

```
elemento.innerText = "Texto visível alterado.";
```

Manipulando Atributos de um Elemento

A manipulação de atributos pode ser feita através das propriedades `getAttribute()`, `setAttribute()`, `removeAttribute()`, e `hasAttribute()`.

1. `getAttribute()`

O método `getAttribute()` permite que você leia o valor de um atributo de um elemento.

```
// Selecionando o elemento
```

```
const link = document.getElementById("meuLink");

// Obtendo o valor do atributo 'href'

const href = link.getAttribute("href");

console.log(href); // Exemplo: "https://www.exemplo.com"
```

2. **setAttribute()**

O método **setAttribute()** altera o valor de um atributo existente ou cria um novo atributo caso ele ainda não exista.

```
// Seleccionando o elemento

const link = document.getElementById("meuLink");

// Modificando o valor do atributo 'href'

link.setAttribute("href", "https://www.novosite.com");
```

3. **removeAttribute()**

O método **removeAttribute()** remove um atributo de um elemento.

```
// Seleccionando o elemento

const link = document.getElementById("meuLink");

// Removendo o atributo 'href'

link.removeAttribute("href");
```

4. `hasAttribute()`

O método `hasAttribute()` verifica se um determinado atributo existe em um elemento.

```
// Selecionando o elemento

const link = document.getElementById("meuLink");

// Verificando se o atributo 'href' existe

const temHref = link.hasAttribute("href");

console.log(temHref); // true ou false
```

Manipulando Estilos de um Elemento

Você pode também manipular o **estilo** de um elemento diretamente, utilizando a propriedade `style`.

1. Alterando Estilos

A propriedade `style` permite que você altere as propriedades CSS de um elemento diretamente em JavaScript.

```
const box = document.getElementById("minhaCaixa");

// Alterando o fundo e a cor do texto

box.style.backgroundColor = "blue";

box.style.color = "white";
```

- **Nota:** As propriedades de estilo são modificadas em **camelCase**, como **backgroundColor**, **fontSize**, etc.

2. Modificando Vários Estilos ao Mesmo Tempo

Embora você possa alterar um estilo de cada vez, uma boa prática é usar **classes** CSS e manipulá-las, ao invés de alterar os estilos diretamente.

// Alterando a classe do elemento para adicionar um estilo

```
box.classList.add("estiloNovo"); // Adiciona uma nova classe
```

```
box.classList.remove("estiloAntigo"); // Remove uma classe existente
```

- **Nota:** Usar **classList** é mais eficiente e mantém o código mais limpo, já que você pode definir várias propriedades de estilo no CSS e manipulá-las com a adição ou remoção de classes.

Exemplo Prático: Manipulando Conteúdo e Atributos

Vamos criar um exemplo em que manipulamos conteúdo e atributos com base em um evento de clique.

HTML:

```
<!DOCTYPE html>
```

```
<html lang="pt-br">
```

```
<head>
```

```
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```



```
<title>Exemplo de Manipulação de DOM</title>

</head>

<body>

  <h2 id="titulo">Bem-vindo ao site!</h2>

  <p id="descricao">Clique no botão para alterar o conteúdo e
  atributos.</p>

  <button id="alterarBtn">Alterar Conteúdo e Atributos</button>

  <a      id="meuLink"      href="https://www.exemplo.com"
  target="_blank">Visitar Exemplo</a>

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

</body>

</html>
```

JavaScript (**script.js**):

```
// Selecionando os elementos

const botao = document.getElementById("alterarBtn");
const titulo = document.getElementById("titulo");
const descricao = document.getElementById("descricao");
const link = document.getElementById("meuLink");

// Adicionando evento de clique

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

  // Alterando o conteúdo do título
```

```
titulo.textContent = "Conteúdo Alterado!";

// Alterando o conteúdo da descrição

descricao.innerHTML = "<strong>O conteúdo da descrição foi  
atualizado.</strong>";

// Alterando o link

link.setAttribute("href", "https://www.novosite.com");

link.textContent = "Visitar Novo Site";

// Alterando o estilo

descricao.style.color = "green";

descricao.style.fontSize = "18px";

});
```



Atividade Prática

1. Crie uma página com um botão. Quando o botão for clicado, altere o conteúdo de um parágrafo e modifique o estilo do texto (exemplo: cor, tamanho da fonte).
2. Crie uma lista de itens e altere o conteúdo de um item específico quando um botão for clicado.
3. Adicione um link na página e use JavaScript para alterar seu **href** e texto.

4. Crie uma imagem em HTML e use JavaScript para alterar o `src` da imagem ao clicar em um botão.
-
-

Eventos no DOM

Eventos são interações do usuário com a página web, como cliques, pressionamentos de teclas, movimentos do mouse, entre outros. O **JavaScript** permite que você responda a esses eventos e execute funções específicas quando eles ocorrem.

O que são eventos?

Eventos são **ações** que ocorrem no navegador e podem ser disparadas por interações do usuário (como um clique), mudanças na página (como carregamento de conteúdo) ou até mesmo ações do próprio sistema (como um erro de rede).

Exemplos de eventos:

- Clique em um botão: `click`
- Passagem do mouse sobre um elemento: `mouseover`
- Pressionar uma tecla: `keydown`, `keypress`, `keyup`
- Carregamento da página: `load`
- Envio de um formulário: `submit`

Esses eventos são importantes para que a página seja interativa. Vamos ver como podemos lidar com esses eventos em JavaScript.

● Como adicionar eventos a elementos

Você pode adicionar eventos aos elementos utilizando o método **`addEventListener()`**, que permite associar uma função de callback a um evento. A sintaxe básica é a seguinte:

```
elemento.addEventListener(tipoDeEvento, função, [fase]);
```

- **elemento**: O elemento ao qual o evento será adicionado.
 - **tipoDeEvento**: O tipo de evento que você quer ouvir (como `click`, `mouseover`, `keydown`, etc.).
 - **função**: A função que será executada quando o evento ocorrer.
 - **fase (opcional)**: Define em que fase o evento será capturado (fase de captura ou fase de propagação).
-

● Exemplo Básico de Evento

Vamos ver um exemplo simples de como adicionar um evento de clique a um botão e executar uma função quando o evento ocorrer.

HTML:

```
<button id="meuBotao">Clique em mim!</button>
```

```
<p id="mensagem">Aguardando clique...</p>
```

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

JavaScript (**script.js**):

```
// Selecionando os elementos
```

```
const botao = document.getElementById("meuBotao");
```

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

```
// Adicionando o evento de clique
```

```
botao.addEventListener("click", function() {
```

```
    mensagem.textContent = "Você clicou no botão!";
```

```
});
```

Neste exemplo, ao clicar no botão, a mensagem do parágrafo será alterada para "Você clicou no botão!".

Tipos Comuns de Eventos

Aqui estão alguns dos **tipos de eventos** mais comuns que você pode usar em JavaScript:

1. **click**

O evento **click** ocorre quando um usuário clica em um elemento (geralmente um botão, link ou imagem).

```
botao.addEventListener("click", function() {
```

```
    alert("Você clicou no botão!");
```

```
});
```

2. **mouseover** e **mouseout**

O evento **mouseover** ocorre quando o mouse passa sobre um elemento, e o evento **mouseout** ocorre quando o mouse sai do elemento.

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

```
caixa.addEventListener("mouseover", function() {  
    caixa.style.backgroundColor = "lightblue";  
});
```

```
caixa.addEventListener("mouseout", function() {  
    caixa.style.backgroundColor = "white";  
});
```

3. **keydown**, **keypress** e **keyup**

Esses eventos estão relacionados ao teclado:

- **keydown** ocorre quando uma tecla é pressionada.
- **keypress** ocorre quando uma tecla é pressionada e mantida.
- **keyup** ocorre quando a tecla é solta.

```
document.addEventListener("keydown", function(event) {  
    console.log("Tecla pressionada: " + event.key);  
});
```

4. **submit**

O evento **submit** é disparado quando um formulário é enviado.

```
const formulario = document.getElementById("form");

formulario.addEventListener("submit", function(event) {
    event.preventDefault(); // Impede o envio do formulário
    alert("Formulário enviado!");
});
```

- **Nota:** `event.preventDefault()` é usado para evitar que o comportamento padrão do evento ocorra (como o envio de um formulário).

5. **load**

O evento **load** ocorre quando a página (ou um recurso, como uma imagem) é completamente carregado.

```
window.addEventListener("load", function() {
    console.log("A página foi totalmente carregada!");
});
```

● O Objeto **event**

O **objeto event** contém informações sobre o evento que foi disparado. Ele é passado automaticamente para a função de callback e contém propriedades úteis, como o tipo de evento, o elemento alvo, as teclas pressionadas, etc.

Exemplo:

```
document.addEventListener("click", function(event) {  
    console.log(event.type); // Exibe "click"  
    console.log(event.target); // Exibe o elemento que foi clicado  
});
```

- **event.type**: O tipo do evento (ex: "click", "keydown").
- **event.target**: O elemento que disparou o evento.

● Remover um Evento

Se você quiser **remover** um evento, pode usar o método **removeEventListener()**. Ele requer a mesma função de callback que foi passada para o **addEventListener()**.

```
const botao = document.getElementById("meuBotao");
```

```
function minhaFuncao() {  
    alert("Você clicou no botão!");  
}
```


// Adicionando o evento

```
botao.addEventListener("click", minhaFuncao);
```

// Removendo o evento

```
botao.removeEventListener("click", minhaFuncao);
```

- **Nota:** O método `removeEventListener()` só funciona se você usar uma referência para a função que foi passada para `addEventListener()`. Funções anônimas não podem ser removidas dessa forma.

🟡 Eventos de Propagação: Captura e Bubbling

O DOM tem dois tipos principais de propagação de eventos: **captura** e **bubbling**.

- **Captura:** O evento começa a ser processado no elemento mais externo e vai em direção ao elemento mais interno.
- **Bubbling:** O evento começa no elemento mais interno e sobe para os elementos mais externos.

A forma padrão de propagação é o **bubbling**, mas você pode especificar a fase de captura ao adicionar o evento.

// Exemplo de captura

```
elemento.addEventListener("click", minhaFuncao, true); // true para captura
```

🟡 Exemplo Prático de Evento de Clique

Vamos criar um exemplo com um botão e um campo de texto. Quando o botão for clicado, o conteúdo do campo de texto será alterado.

HTML:

```
<input type="text" id="campoTexto" placeholder="Digite algo...">
```

```
<button id="meuBotao">Alterar Texto</button>
```

```
<p id="mensagem">O texto será alterado ao clicar no botão.</p>
```

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

JavaScript (**script.js**):

```
const botao = document.getElementById("meuBotao");
```

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

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

```
botao.addEventListener("click", function() {
```

```
    mensagem.textContent = "Você digitou: " + campoTexto.value;
```

```
});
```



Atividade Prática

1. Crie um campo de entrada de texto e um botão. Quando o botão for clicado, o texto da página deve mudar para o que foi digitado no campo.
 2. Crie um exemplo de evento `mouseover` em que a cor de fundo de um elemento mude quando o mouse passar sobre ele.
 3. Use o evento `submit` para validar um formulário e mostrar uma mensagem ao usuário sem enviar o formulário.
-