



Capacitação em JavaScript

▲ O conteúdo a seguir faz parte da capacitação em JavaScript da CIMATEC jr.

O modelo conta com vídeos e links organizados de maneira sequencial para o melhor entendimento do conteúdo. Ademais, é passado um desafio prático para aplicação dos conteúdos aprendidos. A consulta de outros materiais complementares é recomendada e muito bem-vinda. E, por fim, é válido frisar que qualquer dúvida sobre o conteúdo ou desafio pode e deve ser perguntada. Então, não existem em perguntar e boa capacitação!

Introdução

JavaScript (JS) é uma linguagem de programação **multiparadigma, dinâmica, interpretada** e de **alto nível**. É a linguagem padrão dos navegadores juntamente com o HTML e CSS. Além disso, é uma linguagem extremamente versátil que possui respaldo de diversas bibliotecas que a permitem transitar em diversos ambientes, sendo para atuar no Front-End, no Back-End ou até mesmo para desenvolvimento Mobile.



Bem, o que significam cada um dos termos destacados? E como exatamente esses conceitos afetam na hora de desenvolver códigos e aplicações?

1. Multiparadigma

Os paradigmas de programação são basicamente formas de abordar e resolver problemas ao escrever código. Eles são como conjuntos de regras que guiam os desenvolvedores. Cada linguagem de programação é criada com base nessas regras. Existem diversos paradigmas de programação e o JavaScript é considerado multiparadigma por ser facilmente utilizado por essas várias formas de programar. Alguns dos paradigmas mais comuns pelos quais os desenvolvedores utilizam o JavaScript são os: Paradigma Orientado a Objetos, Paradigma Imperativo e o Paradigma Funcional.

2. Dinâmica

O JavaScript é uma linguagem dinâmica de tipagem fraca, isso significa que uma variável que outrora era do tipo *String* pode ter um dado do tipo *Inteiro* atribuído a ela.

```
int x = 23;
x = 'c'; // x = 99
```

Nesse caso, na linguagem C, haveria uma conversão de 'c' para um valor inteiro, já que estamos tentando atribuir um *Character* a uma variável do tipo *Inteiro*. O objetivo era que o valor de x fosse o carácter 'c' e não 99, que é o valor de 'c' convertido para inteiro de acordo com a tabela ASCII. Vamos ver como seria em JavaScript, mas lembremos desse exemplo depois.

```
let x = 23;
x = "Novo valor" // x = "Novo valor"
```

Já em JavaScript, a variável que era do tipo *Number* (mesmo não existindo em nenhum lugar essa declaração de forma explícita) pôde receber uma *String*. Isso graças a linguagem ser de tipagem dinâmica, em contraste ao C, que é de tipagem estática.

Ademais, a característica tipagem fraca que foi comentada refere-se a como a linguagem se comporta quando há um tipo inesperado de dado. Lembrando do exemplo da linguagem C:

```
int x = 23;
x = 'c'; // x = 99
```

Apesar de ser de tipagem estática, o C também possui tipagem fraca como o JavaScript. Por isso, ele converteu automaticamente o carácter 'c' para 99 usando a tabela ASCII.

```
const num = "10";
const operacao = num + 10; // operacao = "1010"
```

O programa não gerou erros, pois o JavaScript fez uma **conversão implícita** do valor 10 de *Number* para *String*, para então concatenar com a *String* "10".

3. Interpretada

Dizer que JavaScript é uma linguagem interpretada significa que seu código é executado diretamente pelo interpretador, linha por linha, sem a necessidade de compilação prévia. Isso difere das linguagens compiladas, como C ou Java, que exigem a conversão do código-fonte para um código de máquina antes da execução.

No caso do JavaScript, os navegadores desempenham o papel de interpretador, processando o código no momento da execução. Essa característica proporciona grande flexibilidade durante o desenvolvimento, como a possibilidade de testar alterações rapidamente, mas também pode impactar negativamente no desempenho em alguns casos, especialmente em aplicações muito complexas.

Vantagens da interpretação:

- Portabilidade, as linguagens interpretadas são mais portáteis, pois o mesmo código pode ser executado em diferentes plataformas.

Desvantagens:

- Execução mais lenta comparada às linguagens compiladas em alguns cenários.
- Possibilidade de erros só serem detectados durante a execução.

4. Alto nível

Alto nível ou baixo nível são termos que se referem ao nível de abstração de uma linguagem. O JavaScript é uma linguagem de alto nível por ser mais próxima de uma língua real e transcrever essas instruções para a máquina. Diferentemente, linguagens de baixo nível são muito mais parecidas com a linguagem de máquina e há um esforço maior do programador para definir qual forma e como será feita a ação.

Exemplo em JavaScript:

```
console.log('Hello, world!')
```

Exemplo em Assembly:

```
section .text
    global _start
_start:
    mov     edx, len
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     0x80
    mov     eax, 1
    int     0x80
section .data
msg db     'Hello, world!',0xa
len equ $ - msg
```

Objetivos de Aprendizagem da Seção

Esta seção introdutória serve apenas para familiarizar com o JavaScript. Estas características serão recorrentemente lembradas à medida que os primeiros códigos e projetos forem feitos com essa linguagem. Portanto, não é necessária uma preocupação muito grande com esses conceitos, eles são importantes para prover um pequeno contexto à parte seguinte do material.

Declarando Variáveis

Os tópicos a seguir são muito melhor compreendidos com exemplos práticos e demonstrações, assim ao lado de cada tópico há ao menos um vídeo que o ilustra bem. A partir daqui, a playlist de JavaScript do Matheus Battisti é altamente recomendada. [PLAYLIST](#)

- Declarando Variáveis → [Vídeo explicativo #5](#)
- Tipos de Dados → [Vídeo explicativo #06](#), [Vídeo explicativo #07](#), [Vídeo explicativo #08](#) e [Vídeo explicativo #09](#)

Declarando Variáveis

1. Usando `let`

```
let idade = 25;
```

Aqui estamos dizendo:

"Crie uma caixinha chamada `idade` e guarde o número 25 dentro dela."

- `let` é usado quando você sabe que o valor da variável pode mudar depois.

Por exemplo:

```
let idade = 25;
idade = 26; // Agora a variável "idade" tem o valor 26.
```

2. Usando `const`

```
const nome = "Ana";
```

Aqui estamos dizendo:

"Crie uma caixinha chamada `nome` e guarde o texto `Ana` dentro dela."

Mas atenção:

- `const` significa **constante**. O valor de uma variável criada com `const` não pode ser mudado depois.


Por exemplo:

```
const nome = "Ana";
nome = "Carlos"; // Isso vai dar erro, porque "nome" foi declarado com const
```

3. Usando `var`

```
var cidade = "São Paulo";
```

Antigamente, usávamos `var` para declarar variáveis, mas hoje preferimos usar `let` e `const` porque são mais seguras. Ainda assim, você pode encontrar `var` em códigos mais antigos.

 **Detalhe Importante:** Esta é uma introdução sobre os `let`, `const` e `var`. Uma explicação mais detalhada se encontra mais abaixo, depois de explicados o que são os escopos.

Tipos de Dados

Nas variáveis, você pode guardar diferentes tipos de valores. Vamos conhecer os principais:

- **Números:**

```
let idade = 30; // Um número inteiro.
let altura = 1.75; // Um número decimal.
```

- **Textos (Strings):**

```
let nome = "Maria"; // Um texto sempre vai entre aspas (simples ou duplas).
```

- **Booleanos** (verdadeiro ou falso):

```
let maiorDeIdade = true; // true significa "sim".
let possuiCNH = false; // false significa "não".
```

Regras para Nomes de Variáveis

1. O nome da variável deve começar com uma letra, underline (_) ou cifrão (\$).

Exemplos válidos:

```
let nome;  
let _idade;  
let $salario;
```

2. Não pode começar com números.

```
let 1numero; // ERRADO  
let numero1; // CERTO
```


3. Não pode usar palavras reservadas da linguagem, como `let`, `const`, `if`, etc.

```
let let; // ERRADO
```

Primeiro Exemplo Completo

Vamos juntar tudo o que aprendemos até agora:

```
let nome = "João"; // Guardando um texto na variável "nome".  
let idade = 20;    // Guardando um número na variável "idade".  
const cidade = "Salvador"; // Guardando um texto em uma constante.  
  
console.log("Meu nome é " + nome); // Exibe: Meu nome é João.  
console.log("Tenho " + idade + " anos."); // Exibe: Tenho 20 anos.  
console.log("Eu moro em " + cidade); // Exibe: Eu moro em Salvador.
```

 **Teste:** Tente imaginar mentalmente o que seria exibido, mas também reproduza esses códigos no <https://playcode.io/javascript>.

Objetivos de Aprendizagem da Seção

Esta seção introdutória serve apenas para familiarizar com a declaração de variáveis, os tipos de dados e os primeiros passos na linguagem JavaScript. Estas características serão recorrentemente lembradas à medida que os primeiros códigos e projetos forem desenvolvidos com essa linguagem. Portanto, não é necessária uma preocupação muito grande com esses conceitos neste momento, pois eles são importantes apenas para prover um pequeno contexto à parte seguinte do material.

Arrays e Objetos

- Objetos → [Vídeo explicativo #10](#)
- Arrays → [Vídeo explicativo #11](#)

Arrays

Arrays (ou vetores) são listas ordenadas de valores. Eles permitem armazenar múltiplos itens em uma única variável. Cada valor em um array está associado a uma **posição numérica** (chamada de índice), que começa em **zero**. Use arrays para guardar valores que têm uma sequência ou lista lógica.

Exemplo 1: Trabalhando com Arrays Simples

```
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas[0]); // "maçã" (primeiro item)
console.log(frutas[2]); // "laranja" (terceiro item)
```

Aqui, criamos um array chamado `frutas` que contém três itens. Usamos os índices (números que começam no zero) para acessar os valores.

Exemplo 2: Adicionando e Removendo Itens

```
let numeros = [1, 2, 3];
numeros.push(4); // Adiciona 4 ao final do array
console.log(numeros); // [1, 2, 3, 4]

numeros.pop(); // Remove o último item
console.log(numeros); // [1, 2, 3]
```

Aqui usamos o método `.push()` para adicionar um item ao final do array e `.pop()` para remover o último item. Os métodos de Array serão explicados futuramente neste documento, então não é necessário se preocupar com isso por enquanto.

Exemplo 3: Alterando Valores

```
let cores = ["vermelho", "azul", "verde"];
cores[1] = "amarelo"; // Substitui "azul" por "amarelo"
console.log(cores); // ["vermelho", "amarelo", "verde"]
```

Os arrays permitem alterar itens diretamente por meio de seus índices.

Exemplo 4: Misturando Tipos de Dados

```
let misto = [42, "texto", true, null];
console.log(misto); // [42, "texto", true, null]
```

Os arrays podem armazenar diferentes tipos de dados, como números, textos, valores booleanos e até outros arrays.

Objetos

Objetos são coleções de **pares chave-valor**. Enquanto arrays usam índices numéricos para organizar seus valores, objetos permitem usar **nomes descritivos** (chamados de **chaves**) para identificar cada valor. Eles são úteis para armazenar características ou propriedades de algo.

Exemplo 1: Criando e Acessando um Objeto

```
let pessoa = { nome: "João", idade: 25, cidade: "São Paulo" };
console.log(pessoa.nome); // "João" (acessando pela chave 'nome')
console.log(pessoa.idade); // 25 (acessando pela chave 'idade')
```

No exemplo, o objeto `pessoa` representa informações sobre alguém. Podemos acessar seus valores usando o nome da chave.

Exemplo 2: Adicionando e Alterando Propriedades

```
let carro = { marca: "Toyota", modelo: "Corolla" };
carro.ano = 2020; // Adiciona a chave 'ano' com o valor 2020
carro.modelo = "Camry"; // Altera o valor da chave 'modelo'
console.log(carro); // { marca: "Toyota", modelo: "Camry", ano: 2020 }
```

Você pode adicionar novas propriedades a um objeto ou alterar as existentes.

Exemplo 3: Acessando Valores com Colchetes

```
let produto = { nome: "Notebook", preco: 2500 };
console.log(produto["nome"]); // "Notebook"
```

Além do ponto (`.`), também podemos usar colchetes para acessar valores de um objeto.

Exemplo 4: Objetos Dentro de Objetos

```
let estudante = {
  nome: "Ana",
  curso: {
    nome: "Engenharia",
    duracao: 5,
  },
};
console.log(estudante.curso.nome); // "Engenharia"
```

Objetos podem conter outros objetos dentro de si, permitindo uma organização mais complexa.

Comparação entre Arrays e Objetos

Resumindo:

- **Arrays** são listas organizadas por índices (números). Use arrays para armazenar conjuntos de valores ordenados, como uma lista de frutas ou números.
- **Objetos** são coleções de pares chave-valor. Use objetos para armazenar dados mais estruturados e descritivos, como informações sobre uma pessoa ou um produto.

Exemplo Comparativo

```
// Array
let frutas = ["maçã", "banana", "laranja"];
console.log(frutas[1]); // "banana" (acessando pelo índice)

// Objeto
let fruta = { tipo: "maçã", cor: "vermelha" };
console.log(fruta.tipo); // "maçã" (acessando pela chave)
```

Enquanto usamos **índices numéricos** para acessar itens em arrays, usamos **nomes descritivos (chaves)** para acessar valores em objetos.

Sintaxe:

- **Array:** Use **colchetes** `[]` para declarar e acessar itens.
- **Objeto:** Use **chaves** `{ }` para declarar e acessa-se valores com `.` ou `[]`.

Objetivos de Aprendizagem da Seção

Esta seção serve para familiarizar com a estrutura e o uso de **arrays** e **objetos** no JavaScript. Esses conceitos básicos são fundamentais para organizar e manipular dados de maneira eficiente e serão frequentemente aplicados em projetos e exercícios futuros. Não é necessário memorizar todos os métodos ou detalhes agora, pois esses tópicos serão retomados e reforçados ao longo do material. O objetivo principal é compreender a diferença entre arrays e objetos e como utilizá-los em situações simples.

Operadores Relacionais e Lógicos

Operadores Relacionais

Os operadores relacionais comparam dois valores e retornam um **valor booleano**: `true` ou `false`.

Eles são usados para verificar **relações** como maior, menor, igual, etc.

1. Igualdade (`==`)

- Verifica se dois valores são iguais, **sem considerar o tipo**.
- Exemplo:

```
console.log(5 == '5'); // true (os valores são iguais, mesmo com tipos diferentes)
```

2. Igualdade estrita (`===`)

- Verifica se dois valores são iguais, **considerando o tipo**.
- Exemplo:

```
console.log(5 === '5'); // false (os tipos são diferentes: número e string)
```

3. Diferente (`!=`)

- Verifica se dois valores são diferentes, **sem considerar o tipo**.
- Exemplo:

```
console.log(5 != '5'); // false (os valores são iguais)
```

4. Diferente estrito (`!==`)

- Verifica se dois valores são diferentes, **considerando o tipo**.
- Exemplo:

```
console.log(5 !== '5'); // true (os tipos são diferentes)
```

5. Maior que (`>`)

- Verifica se o valor da esquerda é maior que o da direita.
- Exemplo:

```
console.log(10 > 5); // true
```

6. Menor que (`<`)

- Verifica se o valor da esquerda é menor que o da direita.
- Exemplo:

```
console.log(10 < 5); // false
```

7. Maior ou igual (`>=`)

- Verifica se o valor da esquerda é maior **ou igual** ao da direita.
- Exemplo:


```
console.log(5 >= 5); // true
```

8. Menor ou igual (<=)

- Verifica se o valor da esquerda é menor **ou igual** ao da direita.
- Exemplo:

```
console.log(3 <= 5); // true
```

Operadores Lógicos

Os operadores lógicos combinam expressões booleanas e retornam um **valor booleano**.

1. E lógico (&&)

- Retorna `true` se **todas** as condições forem verdadeiras.
- Exemplo:

```
console.log(true && true); // true
console.log(5 > 3 && 10 < 20); // true
console.log(5 > 3 && 10 > 20); // false
```

2. Ou lógico (||)

- Retorna `true` se **pelo menos uma** das condições for verdadeira.
- Exemplo:

```
console.log(true || false); // true
console.log(5 > 3 || 10 > 20); // true
console.log(5 < 3 || 10 > 20); // false
```

3. Não lógico (!)

- Inverte o valor de uma condição:
 - `true` vira `false` e vice-versa.
- Exemplo:

```
console.log(!true); // false
console.log(!(5 > 3)); // false
```

Objetivos de Aprendizagem da Seção

Esta seção é muito importante para os tópicos seguintes, que são Estruturas Condicionais e Estruturas de Repetição. Só podemos adicionar uma condição para um código se esse mesmo código puder julgar corretamente a situação condicional que impusermos. Desta forma, conhecer bem os operadores relacionais e lógicos nos permite fazer pares `if` e `else` mais complexos, assim como loops mais precisos com as estruturas de repetição. Vale destacar que essa seção nos ajuda a compreender melhor os booleans, tipo de dado extremamente importante e que baseia completamente os ramos da computação e da elétrica.

Estruturas Condicionais

As estruturas condicionais são formas de um algoritmo executar determinadas instruções de acordo com diferentes situações que aconteçam. Isso permite uma maior robustez do que esteja sendo desenvolvido e também traz uma maior dinamicidade. De forma grosseira, essas estruturas servem para executar instruções que só queremos que aconteçam se algum requisito/condição for atendido. Dessa forma, o código se comportará diferentemente conforme a situação apresentada.

- `if` → [Vídeo explicativo #12](#)
- `else` e `else if` → [Vídeo explicativo #14](#)

1. O que é o `if` ?

O `if` significa "se". Ele executa um bloco de código **se uma condição for verdadeira (`true`)**. Dessa forma, é escrita uma condição entre parênteses ao lado do `if` e se esta condição retornar o valor booleano `true` um bloco de código entre chaves que será posto depois dos parênteses da condição irá ser executado pelo algoritmo. Porém, **se a condição for falsa (`false`)**, esse mesmo bloco de código será totalmente ignorado.

Estrutura básica:

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

Exemplo:

```
let idade = 18;  
  
if (idade >= 18) {  
    console.log("Você é maior de idade!");  
}  
// Saída: "Você é maior de idade!"
```

A comparação `idade >= 18` retorna o valor booleano `true`, pois idade vale 18 e o 18 é propriamente um valor maior ou igual ao outro 18. Dessa forma, o bloco de código que a estrutura `if` comporta será executado e a frase `"Você é maior de idade!"` é exibida.

2. O que é o `else` ?

O `else` significa "senão". Ele é usado para executar um bloco de código **se a condição do `if` for falsa**.

Estrutura básica:


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

Exemplo:

```
let idade = 16;  
  
if (idade >= 18) {  
    console.log("Você é maior de idade!");  
}
```

```
else {
  console.log("Você é menor de idade!");
}
// Saída: "Você é menor de idade!"
```

A comparação `idade >= 18` retorna o valor booleano `false`, pois idade vale 16 e o 16 não é propriamente um valor maior ou igual ao 18, visto que é menor. Dessa forma, o bloco de código que a estrutura `if` comporta não será executado. No entanto, nesse momento que o `else` entra em ação. Quando o `if` não tem sua condição atendida, o bloco de código do `else` é executado. Dessa forma, a frase `"Você é menor de idade!"` é exibida.

 **Detalhe Importante:** O `else` não é acompanhado de uma condição entre parênteses. Isso ocorre simplesmente porque a condição para o `else` ser executado é que o `if` não tenha sido. Portanto, a condição implícita do `else` é a não execução de um `if` anterior.

3. O que é o `else if` ?

O `else if` significa "senão se". Ele é usado quando você tem **mais de uma condição para verificar**. Você pode usar vários `else if` seguidos, mas o **primeiro que for verdadeiro será executado**.

Estrutura básica:

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

Exemplo:

```
let nota = 85;

if (nota >= 90) {
  console.log("Você tirou um A!");
}
else if (nota >= 80) {
  console.log("Você tirou um B!");
}
else if (nota >= 70) {
  console.log("Você tirou um C!");
}
else {
  console.log("Você precisa melhorar!");
}
// Saída: "Você tirou um B!"
```

A comparação `nota >= 90` retorna o valor booleano `false`, pois nota vale 85 e o 85 não é propriamente um valor maior ou igual ao 90, visto que é menor. Dessa forma, o bloco de código que a estrutura `if` comporta não será executado. No entanto, nesse momento que o `else if` entra em ação. Quando o `if` não tem sua condição atendida, o `else if` terá sua condição avaliada para que o algoritmo entenda se esse bloco de código do `else if` será executado ou se o próximo `else if` será avaliado. A comparação `nota >= 80` retorna o valor booleano `true`, pois nota vale 85 e o 85 é um valor propriamente maior ou igual a 80. Portanto, esse bloco de código será executado, a frase `"Você tirou um B!"` exibida e todos os `else if` seguintes e o `else` serão ignorados.

 **Detalhe Importante:** O `else` só seria executado é uma situação que o `if` e também todos os `else if` fossem **falsos**.

Como eles funcionam juntos?

Você pode combinar o `if`, `else if` e `else` para cobrir todas as possibilidades:

Exemplo com múltiplas condições:

```
let hora = 14;

if (hora < 12) {
  console.log("Bom dia!");
}
else if (hora < 18) {
  console.log("Boa tarde!");
}
else {
  console.log("Boa noite!");
}
```



Teste: Qual seria a saída nesse caso? Tente imaginar mentalmente o que seria exibido. Para conferir a resposta aplique esse código completo em um editor de código como o <https://playcode.io/javascript>.

Regras Importantes

1. **A condição deve ser um valor que pode ser avaliado como `true` ou `false`.**
 - Por exemplo, expressões como `idade >= 18` ou `nota >= 90` funcionam.
2. **A ordem importa no `else if`.**
 - O JavaScript para na **primeira condição verdadeira**.
3. **O `else` é opcional.**
 - Use-o apenas se precisar de um "plano B" para os casos em que nenhuma condição for atendida.

Objetivos de Aprendizagem da Seção

Dessa seção, é importante se entender tudo de forma minuciosa. Isso porque essas estruturas vão aparecer constantemente em todo projeto minimamente robusto que forem fazer. Somando-se ao uso dos operadores relacionais e lógicos, os conteúdos dessa seção permitem construir um número muito grande de programas e adaptá-los a condições críticas.

Estruturas de Repetição

As estruturas de repetição são essenciais para criar códigos eficazes e eficientes ao permitir executar uma mesma ação diversas vezes escrevendo poucas linhas. Imagina ter que escrever mil vezes uma mesma linha de código, o código ficaria desnecessariamente extenso. Por isso, existem essas estruturas que garantem essa economia de linhas e também um poder maior para o algoritmo.

- `while` → [Vídeo explicativo #19](#)
- `for` → [Vídeo explicativo #21](#)

🚧 **Detalhe:** As estruturas mais importantes são o `while` e o `for`, porém estruturas como `do...while`, `for...of` e `for...in` são muito úteis e ficam como bônus aqui nessa seção.

1. `for` (para laços controlados)

O `for` é usado quando você sabe **quantas vezes** quer repetir o código.

Estrutura:

```
for (inicialização; condição; incremento) {  
  // Código a ser executado em cada repetição  
}
```

- **Inicialização:** Declara e inicializa uma variável de controle (executada uma vez no início).
- **Condição:** Verifica se o loop deve continuar (é avaliada antes de cada repetição).
- **Incremento:** Atualiza a variável de controle a cada repetição.

Exemplo:

```
for (let i = 0; i < 5; i++) {  
  console.log("Repetição número:", i);  
}  
// Saída: 0, 1, 2, 3, 4
```

2. `while` (enquanto)

O `while` é usado quando **não sabemos exatamente o número de repetições**, mas queremos repetir enquanto uma condição for verdadeira.

Estrutura:

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

Exemplo:


```
let contador = 0;  
  
while (contador < 3) {  
  console.log("Contador é:", contador);  
  contador++; // Incrementa o valor do contador  
}  
// Saída: 0, 1, 2  
//Contador é: 0  
//Contador é: 1  
//Contador é: 2
```

⚠ **Cuidado com loops infinitos:** Se a condição nunca se tornar falsa, o código ficará preso no loop.

3. `do...while` (faça enquanto)

O `do...while` é parecido com o `while`, mas ele **executa o código pelo menos uma vez** antes de verificar a condição.

Estrutura:



```
do {  
  // Código a ser executado  
} while (condição);
```

Exemplo:

```
let numero = 0;  
  
do {  
  console.log("Número é:", numero);  
  numero++;  
} while (numero < 3);  
// Saída:  
//Número é: 0  
//Número é: 1  
//Número é: 2
```



4. **for...of** (para valores em objetos iteráveis)

O **for...of** é usado para percorrer **valores** de objetos iteráveis, como arrays, strings, ou objetos do tipo Map/Set.

Estrutura:

```
for (const valor of iteravel) {  
  // Código a ser executado para cada valor  
}
```

Exemplo com Array:

```
const frutas = ["Maçã", "Banana", "Uva"];  
  
for (const fruta of frutas) {  
  console.log("Fruta:", fruta);  
}  
// Saída:  
//Fruta: Maçã  
//Fruta: Banana  
//Fruta: Uva
```

Exemplo com String:

```
const palavra = "JavaScript";  
  
for (const letra of palavra) {  
  console.log(letra);  
}  
// Saída: J, a, v, a, S, c, r, i, p, t
```

5. **for...in** (para propriedades de objetos)

O **for...in** é usado para iterar sobre as **propriedades** de um objeto.

Estrutura:

```
for (const propriedade in objeto) {  
  // Código a ser executado para cada propriedade  
}
```

Exemplo:

```
const pessoa = { nome: "Ana", idade: 25, cidade: "Salvador" };  
  
for (const chave in pessoa) {  
  console.log(chave + ":", pessoa[chave]);  
}  
// Saída:  
// nome: Ana  
// idade: 25  
// cidade: Salvador
```

O código define um objeto `pessoa` com três propriedades: `nome`, `idade` e `cidade`. Em seguida, é utilizado um loop `for...in` para percorrer todas as chaves do objeto. A cada iteração, a chave e seu valor correspondente são exibidos no console. No exemplo, o loop percorre as chaves `nome`, `idade` e `cidade`, e para cada chave, o valor é acessado usando `pessoa[chave]`.

Objetivos de Aprendizagem da Seção

Dessa seção, é importante se entender tudo de forma minuciosa. Isso porque essas estruturas vão aparecer constantemente em todo projeto minimamente robusto que forem fazer. Somando-se ao uso dos operadores relacionais e lógicos, os conteúdos dessa seção permitem construir um número muito grande de programas e executar ações muito extensas com poucas linhas de código escritas.

Funções e Arrow Functions

As **funções** são **blocos de códigos reutilizáveis** e são fundamentais para **encurtar o número de linhas de um código, torná-lo mais simples e facilitar sua manutenção**. Uma função que soma dois números, por exemplo, pode ser útil em diferentes outras partes de um algoritmo e declará-la como uma função possibilita não ter que escrever novamente as mesmas linhas de código para executar essa ação. Além disso, essa mesma função pode ser alterada em todos os locais em que ela aparece, apenas mudando as instruções de um único lugar (que é o local onde a função foi declarada). Outro aspecto importante das funções é que **elas precisam ser chamadas**, ou seja, não basta simplesmente declará-la, **é preciso usar um comando para que ela comece a ser executada em seu algoritmo**.

- Funções → [Vídeo explicativo #23](#)
- Arrow Functions → [Vídeo explicativo sobre Arrow Functions](#)

Como declarar uma função

1. Função tradicional

A forma mais comum e antiga de declarar funções:

```
function nomeDaFuncao(param1, param2, ...) {  
  // Corpo da função
```

```
    return resultado; // Opcional
}
```

Exemplo:

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

console.log(somar(2, 3)); // 5
```

2. Função anônima

Funções que não possuem um nome e geralmente são atribuídas a uma variável.

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

console.log(somar(2, 3)); // 5
```

3. Arrow Function

Uma forma mais curta e moderna de declarar funções (introduzida no ES6). Haverá maior destaque para as Arrow Functions abaixo, pois ela são muito utilizadas para simplificar a escrita de instruções em alguns métodos de array como `map`, `filter` e `forEach`.

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

console.log(somar(2, 3)); // 5
```

A função `somar` é uma arrow function que recebe dois parâmetros, `a` e `b`, e retorna a soma de ambos. No exemplo, a função é chamada com os valores `2` e `3`, então a soma realizada é `2 + 3`, resultando em `5`. Esse valor é exibido no console.

Componentes de uma função

1. Parâmetros

Os **parâmetros** são valores que você passa para uma função para que ela possa usá-los.

Exemplo:

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

console.log(cumprimentar("João")); // "Olá, João!"
```

- Você pode definir valores **padrão** para parâmetros:

```
function cumprimentar(nome = "Visitante") {
    return `Olá, ${nome}!`;
}
```

```
console.log(cumprimentar()); // "Olá, Visitante!"
```

A função `cumprimentar` possui um parâmetro chamado `nome` que, caso não seja informado durante a chamada da função, assume o valor padrão `"Visitante"`. A função retorna a frase `"Olá, ${nome}!"`, onde `${nome}` será substituído pelo valor do parâmetro fornecido. No exemplo, a função é chamada sem nenhum argumento, então o parâmetro `nome` assume o valor padrão `"Visitante"`. Por isso, a frase `"Olá, Visitante!"` é exibida no console.

2. Retorno (`return`)

O `return` é usado para enviar o resultado de uma função para quem a chamou.

Exemplo:

```
function quadrado(x) {  
  return x * x;  
}  
  
console.log(quadrado(4)); // 16
```

A função `quadrado` recebe um parâmetro `x` e retorna o valor de `x` multiplicado por ele mesmo, ou seja, o quadrado de `x`. No exemplo, a função é chamada com o argumento `4`, então o cálculo realizado é `4 * 4`, resultando em `16`. Esse valor é exibido no console.

Se não usar o `return`, a função retornará `undefined` por padrão.

Arrow Functions → Exemplos de utilização

Abaixo alguns exemplos de uso de Arrow Functions e é possível perceber porque elas são tão utilizadas para deixar o código mais curto e elegante, porém com a mesma eficiência. No penúltimo exemplo, a Arrow Function está acompanhando o Método de Array `.map()`, que será explicado mais tarde nesse material. O último exemplo apresenta a Arrow Function sendo utilizada juntamente ao operador ternário, que é uma forma mais enxuta de escrever um par `if` e `else`.

```
const cumprimentar = () => "Olá, visitante!";  
  
console.log(cumprimentar()); // "Olá, visitante!"
```

Quando a função não precisa de parâmetros, podemos usar parênteses vazios na arrow function. No exemplo, a função retorna uma saudação fixa sem necessidade de entrada.

```
const dobrar = numero => numero * 2;  
  
console.log(dobrar(4)); // 8
```

Quando há apenas um parâmetro, podemos omitir os parênteses. Aqui, a função `dobrar` recebe um número e retorna o dobro dele.

```
const calcularArea = (base, altura) => {  
  const area = (base * altura) / 2;  
  return area;  
};
```

```
console.log(calcularArea(10, 5)); // 25
```

Para funções com várias linhas de código, usamos chaves (`{ }`) para criar um corpo de bloco. Aqui, a função calcula a área de um triângulo e retorna o resultado.

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

console.log(dobrados); // [2, 4, 6, 8]
```

No exemplo, o método `map` aplica uma Arrow Function a cada elemento do array, retornando um novo array com os números dobrados.

```
const verificarMaioridade = idade => idade >= 18 ? "Maior de idade" : "Menor de idade";

console.log(verificarMaioridade(20)); // "Maior de idade"
console.log(verificarMaioridade(16)); // "Menor de idade"
```

Aqui, a arrow function usa o operador ternário para verificar a maioridade. Se a idade for maior ou igual a 18, retorna `"Maior de idade"`, caso contrário, retorna `"Menor de idade"`.

Objetivos de Aprendizagem da Seção

Esta seção de funções é extremamente importante, visto que as funções são uma das estruturas mais básicas e importantes na programação. Escrever uma função e depois escrever uma função passando seus parâmetros é um exercício fundamental que deve ser feito. Com o domínio das funções juntamente com os conteúdos anteriores, um(a) programador(a) ganha poder para realizar quase que qualquer projeto, mesmo que este seja excessivamente trabalhoso. No que tange as Arrow Functions, seu entendimento total não é o foco. É importante mencioná-la e explicá-la porque ela é muito usada no JavaScript em diversos contextos, como verão mais adiante. Ter noção de que ela é uma função com sintaxe mais simples e de como declará-la já é extremamente válido nesse ponto.

Escopos e `let` **x** `const` **x** `var`

- Escopos → [Vídeo explicativo #25](#)
- `let`, `const` → [Vídeo explicativo #26](#)

Escopos: **Global** **x** **Locais**

O escopo é um conceito que define o comportamento do algoritmo de um bloco de instruções no código de acordo com onde estão localizadas essas instruções no código. De maneira mais simples, de acordo com o escopo, os mesmos comandos e até variáveis de mesmo nome podem agir de maneira diferente. Existem dois tipos de escopos: o **escopo global** e o **escopo local**.

1. O **escopo global** é todo o espaço que existe em um arquivo de código. As linhas, comandos e variáveis escritas em um programa pertencem ao escopo global. Existe apenas um escopo global para cada arquivo. No exemplo abaixo, uma variável `numeroDeCachorros` é criada no escopo global e exibida com o comando `console.log()`.


```
var numeroDeCachorros = 3

console.log(numeroDeCachorros) // 3
```

2. O **escopo local**, por outro lado, é todo que existe dentro de uma função. Podem existir infinitos escopos locais em um arquivo, inclusive podem existir escopos locais dentro de outros escopos locais. No exemplo abaixo, uma variável `numeroDeCachorros` é criada dentro de uma função, ou seja, essa variável pertence ao escopo local da função.

```
function funcaoCachorros(){


var numeroDeCachorros = 3
console.log(numeroDeCachorros) // 3

}

funcaoCachorros() // Será printado o valor '3' ao chamar essa função

console.log(numeroDeCachorros) // ReferenceError: numeroDeCachorros is not defined
```

Ao declarar a função `funcaoCachorros`, é criado um escopo local onde a variável `numeroDeCachorros` existe e possui valor. Quando a função é chamada, o número `3`, que corresponde ao valor da variável, é printado. Porém, ao executarmos o comando para printar a variável `numeroDeCachorros` fora da função, o programa não reconhece a variável. Isso ocorre porque a variável simplesmente não existe no escopo global.

 **Detalhe Importante:** É possível perceber que não é possível acessar de fora uma variável declarada dentro de um escopo local. Porém, no próximo exemplo, ficará claro que é possível acessar uma variável de escopo global de dentro de uma função (escopo local).

```
var numeroDeCachorros = 3

function funcaoCachorros(){

var numeroDeCachorros = 7
console.log(numeroDeCachorros) // 7

}

console.log(numeroDeCachorros) // Será printado o valor '3'

funcaoCachorros() // Será printado o valor '7' ao chamar essa função

console.log(numeroDeCachorros) // Será printado o valor '3'
```

A variável `numeroDeCachorros` é declarada e lhe é atribuída o valor `3`. Dentro da função `funcaoCachorros`, uma variável com o mesmo nome da declarada anteriormente tem o valor `7` atribuído. Dessa forma, o que acontece aqui é que existem duas variáveis `numeroDeCachorros` com valores distintos, uma existindo no escopo global e a outra no escopo local da função `funcaoCachorros`. Abaixo é feito um comando para printar o valor da variável, que será `7` por conta da atribuição feita logo no início da função. Depois da função, um outro comando para printar o valor da variável `numeroDeCachorros` é feito e o valor printado é `3`. Isso ocorre, claro, porque a função não foi chamada ainda, mas também por conta que o JavaScript entende apenas a existência da variável declarada no escopo global. A segunda variável com mesmo nome só existe dentro da função `funcaoCachorros`, isso ficará mais claro a seguir. Ao ser executado o comando para chamar a função `funcaoCachorros`, o valor printado será `7`, pois, lá dentro da função, a variável local `numeroDeCachorros` que possui valor `7` é reconhecida. Por último, é feito mais comando para printar a variável `numeroDeCachorros` fora da função e, novamente, o programa reconhece apenas a variável global `numeroDeCachorros` e não a homônima variável local `numeroDeCachorros`.

Enfim, estes conceitos são muito melhor entendidos com a prática e com a realização de testes. Por isso, é recomendado o uso do <https://playcode.io/javascript> para testar os exemplos acima e também para visualizar os efeitos diferentes que podem surgir de situações muito parecidas. Uma situação diferente vai ser deixada logo abaixo e é importante que testem e procurem perguntar a alguém sobre o motivo que causa a diferença no resultado.

```
var numeroDeCachorros = 3

function funcaoCachorros(){
  numeroDeCachorros = 7
  console.log(numeroDeCachorros)
}
console.log(numeroDeCachorros)

funcaoCachorros()

console.log(numeroDeCachorros)
```

let x const x var

1. var

O `var` é o jeito mais antigo de declarar variáveis no JavaScript (antes do ES6). Ele tem algumas características que podem causar comportamentos inesperados.

Características:

1. Escopo global ou de função:

- Se declarada fora de uma função, a variável `var` será **global**.
- Se declarada dentro de uma função, estará disponível apenas naquela função (escopo local).
- **Não respeita escopo de bloco** (`if`, `for`, etc.).

2. Pode ser redeclarada:

- Você pode declarar a mesma variável com `var` várias vezes sem erro.

3. Hoisting:

- As variáveis declaradas com `var` são "movidas para o topo" do escopo durante a execução, mas sem o valor.

Exemplos:

```
// Escopo global
var nome = "Ana";
console.log(nome); // Ana

// Redeclaração é permitida
var nome = "João";
console.log(nome); // João

// Hoisting
console.log(x); // undefined (a declaração é movida para o topo, mas o valor não)
var x = 5;
```

2. let

O `let` foi introduzido no ES6 (2015) para corrigir problemas do `var`. Ele é mais seguro e moderno.

Características:

1. Escopo de bloco:

- A variável só existe dentro do **bloco** onde foi declarada (entre `{ }`).
- É mais previsível, já que respeita o escopo.

2. Não pode ser redeclarada no mesmo escopo:

- Isso evita erros acidentais.

3. Hoisting (parcial):

- As variáveis `let` são "movidas para o topo" do escopo, mas **não podem ser usadas antes da declaração**.

Exemplos:

```
// Escopo de bloco
if (true) {
  let idade = 25;
  console.log(idade); // 25
}
// console.log(idade); // Erro: idade não está definida (fora do bloco)

// Não permite redeclaração
let cidade = "Salvador";
let cidade = "São Paulo"; // Erro: cidade já foi declarada

// Hoisting parcial
// console.log(a); // Erro: Não é possível acessar 'a' antes da inicialização
let a = 10;
```

3. `const`

O `const` também foi introduzido no ES6 e é usado para declarar **constantes** (valores que não mudam). Mas tem algumas particularidades.

Características:

1. Escopo de bloco:

- Igual ao `let`, as variáveis `const` respeitam o escopo de bloco.

2. Imutabilidade (parcial):

- O valor atribuído a uma `const` **não pode ser reatribuído**.
- Mas, se for um objeto ou array, você **pode alterar suas propriedades ou itens**.

3. Não pode ser redeclarada:

- Assim como `let`, evita erros acidentais.

4. Hoisting (parcial):

- Comportamento igual ao do `let`: não pode ser usada antes da declaração.

Exemplos:

```
// Escopo de bloco
if (true) {
  const nome = "Carlos";
  console.log(nome); // Carlos
```

```
}
// console.log(nome); // Erro: nome não está definido (fora do bloco)

// Imutabilidade
const pi = 3.14;
pi = 3.1415; // Erro: não pode reatribuir valor a uma constante

// Objetos e Arrays podem ser alterados
const pessoa = { nome: "Ana", idade: 25 };
pessoa.idade = 30; // Isso é permitido
console.log(pessoa); // { nome: "Ana", idade: 30 }

const lista = [1, 2, 3];
lista.push(4); // Isso também é permitido
console.log(lista); // [1, 2, 3, 4]
```

Comparação Resumida

Propriedade	var	let	const
Escopo	Global ou função	Bloco	Bloco
Redeclaração	Permitida	Não permitida	Não permitida
Reatribuição	Permitida	Permitida	Não permitida
Hoisting	Sim (valor undefined)	Sim (mas não inicializada)	Sim (mas não inicializada)

Quando usar?

- **const** : Use sempre que possível. É a escolha padrão para valores que não precisam ser reatribuídos.
- **let** : Use quando o valor pode mudar durante o programa.
- **var** : Evite usar, a menos que precise de compatibilidade com código muito antigo.

Objetivos de Aprendizagem da Seção

O entendimento mais aprofundado do que são escopos é essencial tanto no JavaScript como em qualquer outra linguagem. É uma característica de como são construídas as linguagens que dão segurança para cada trecho do código. Isso evita que variáveis de mesmo nome ou funções de mesmo nome entrem em conflito. No entanto, claro que muitas vezes um código precisa ser escrito de uma maneira diferente por conta das limitações de escopo. Desta seção, é importante levar que, se algo não está funcionando no seu código mesmo tendo uma função ou variável declarada, o problema pode ser por conta do escopo. Essa noção te ajudará a encontrar as falhas no seu código.

Metódos de Array

Os Métodos de Array são formas de se trabalhar e manipular os elementos de um array (vetor). Eles são extremamente úteis para obter o número de elementos que o vetor possui, para adicionar um novo elemento ao final, adicionar no início, remover no final, remover no início, saber se existe um determinado elemento naquele vetor e/ou inverter a ordem dos elementos. Além disso, é possível também criar um novo vetor (array) a partir dos elementos filtrados de um primeiro array ou então aplicar uma condição a todos os elementos desse novo array. Bem, essas são apenas algumas das possibilidades e melhor explicadas abaixo.

⚠ **IMPORTANTE:** São vários métodos e muitos detalhes. Não há a necessidade de decorar todos, na verdade, não há necessidade de decorar praticamente nenhum. O que interessa é entender como funciona o uso de métodos, saber da existência deles e, quando for preciso, escolher e saber aprender como usá-los para resolver o seu problema.

- Métodos de Array básicos → [Vídeo Explicativo #30](#)
- Métodos de Array mais avançados → [Vídeo Explicativo sobre Métodos de Array](#)

🚧 Detalhe Importante: A distinção feita aqui entre métodos básicos e mais avançados não existe de fato. Os métodos que aqui foram chamados de “Métodos de Array básicos” são aqueles que não utilizam uma função na hora de serem escritas, enquanto os “mais avançados” utilizam uma função para efetuar suas ações. Outro importante é que essas funções são normalmente escritas com Arrow Functions como será demonstrado em breve.

Metódos de Array básicos

1. `.length`

O `.length` é uma propriedade que retorna o **número de elementos** em um array.

Exemplo:

```
const numeros = [10, 20, 30, 40];
console.log(numeros.length); // 4
```

A propriedade `.length` retorna o número de elementos presentes no array. No exemplo, o array `[10, 20, 30, 40]` possui quatro elementos, então `.length` retorna o valor `4`.

2. `.push()`

O método `.push()` **adiciona um ou mais elementos** ao final de um array e retorna o novo comprimento do array.

Exemplo:

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

O método `.push()` adiciona um ou mais elementos ao final do array, modificando-o e retornando o novo comprimento do array. No exemplo, `"laranja"` é adicionado ao array `["maçã", "banana"]`, resultando no array `["maçã", "banana", "laranja"]`.

3. `.pop()`

O método `.pop()` **remove o último elemento** de um array e o retorna. O array original é modificado.

Exemplo:

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

O método `.pop()` remove o último elemento do array e o retorna. No exemplo, o método remove `"laranja"` do array `["maçã", "banana", "laranja"]`, deixando o array com `["maçã", "banana"]` e retornando `"laranja"`.

4. `.unshift()`

O método `.unshift()` **adiciona um ou mais elementos** ao início de um array e retorna o novo comprimento do array.

Exemplo:

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

O método `.unshift()` adiciona um ou mais elementos ao início do array, retornando o novo comprimento do array. No exemplo, `"maçã"` é adicionado ao início do array `["banana", "laranja"]`, resultando em `["maçã", "banana", "laranja"]`.

5. `.shift()`

O método `.shift()` **remove o primeiro elemento** de um array e o retorna. O array original é modificado.

Exemplo:

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

O método `.shift()` remove o primeiro elemento do array e o retorna. No exemplo, `"maçã"` é removido do array `["maçã", "banana", "laranja"]`, deixando o array com `["banana", "laranja"]` e retornando `"maçã"`.

6. `.reverse()`

O método `.reverse()` **inverte a ordem dos elementos** de um array, modificando-o diretamente.

Exemplo:

```
const numeros = [1, 2, 3, 4];
numeros.reverse();
console.log(numeros); // [4, 3, 2, 1]
```

O método `.reverse()` inverte a ordem dos elementos no array. No exemplo, o array original `[1, 2, 3, 4]` é alterado diretamente, ficando com a ordem inversa `[4, 3, 2, 1]`.

7. `.includes()`

O método `.includes()` **verifica se o array contém um valor específico**, retornando um **valor booleano** (`true` ou `false`).

Exemplo:

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

O método `.includes()` verifica se um valor específico existe no array, retornando `true` ou `false`. No exemplo, o método verifica a presença de `"banana"` e `"laranja"` no array, retornando `true` para `"banana"` e `false` para `"laranja"`.

Metódos de Array mais avançados

1. `.find()`

O método `.find()` retorna o **primeiro elemento** do array que satisfaz a condição passada em uma função de callback. Se nenhum elemento atender à condição, ele retorna `undefined`.

Exemplo:

```
const numeros = [10, 15, 20, 25];
const resultado = numeros.find(num => num > 15);
console.log(resultado); // 20
```

O método `.find()` retorna o primeiro elemento do array que atende a uma condição definida em uma função. No exemplo, a condição é encontrar o primeiro número maior que 15, e o método retorna `20`, já que é o primeiro valor a satisfazer essa condição.

2. `.findIndex()`

O método `.findIndex()` funciona como o `.find()`, mas em vez de retornar o elemento, ele retorna o **índice** do primeiro elemento que satisfaz a condição. Se nenhum elemento atender, retorna `-1`.

Exemplo:

```
const numeros = [10, 15, 20, 25];
const indice = numeros.findIndex(num => num > 15);
console.log(indice); // 2
```

O método `.findIndex()` funciona de forma semelhante ao `.find()`, mas retorna o índice do primeiro elemento que satisfaz a condição passada. No exemplo, o índice de `20`, que é o primeiro número maior que 15, é `2`.

3. `.map()`

O método `.map()` cria um **novo array**, aplicando uma função em cada elemento do array original. Ele não modifica o array original.

Exemplo:

```
const numeros = [1, 2, 3, 4];
const dobrados = numeros.map(num => num * 2);
console.log(dobrados); // [2, 4, 6, 8]
console.log(numeros); // [1, 2, 3, 4]
```

O método `.map()` cria um novo array aplicando uma transformação a cada elemento do array original. No exemplo, cada número é multiplicado por 2, resultando no novo array `[2, 4, 6, 8]`. O array original permanece inalterado.

4. `.filter()`

O método `.filter()` cria um **novo array** com todos os elementos que satisfazem a condição definida na função.

Exemplo:

```
const numeros = [10, 15, 20, 25];
const maioresQue15 = numeros.filter(num => num > 15);
console.log(maioresQue15); // [20, 25]
```

O método `.filter()` cria um novo array contendo apenas os elementos que atendem a uma condição definida em uma função. No exemplo, os números maiores que 15 são filtrados, resultando no array `[20, 25]`.

5. `.reduce()`

O método `.reduce()` aplica uma função em cada elemento do array, acumulando um único valor (como soma, produto, etc.). Ele recebe dois argumentos: o acumulador e o valor atual.

Exemplo:

```
const numeros = [1, 2, 3, 4];
const soma = numeros.reduce((acumulador, atual) => acumulador + atual, 0);
console.log(soma); // 10
```

O método `.reduce()` acumula valores do array em um único resultado, aplicando uma função de callback em cada elemento. No exemplo, o método soma todos os números do array, começando do acumulador `0`, e retorna o total `10`.

6. `.forEach()`

O método `.forEach()` executa uma função de callback para **cada elemento** do array. Ele **não** retorna um novo array e **não** altera o array original.

Exemplo:

```
const numeros = [1, 2, 3];
numeros.forEach(num => console.log(num * 2));
// Saída:
// 2
// 4
// 6
```

O método `.forEach()` executa uma função de callback para cada elemento do array. No exemplo, o método multiplica cada número por 2 e exibe o resultado no console, sem alterar o array original.

7. `.some()`

O método `.some()` verifica se **pelo menos um elemento** do array satisfaz a condição passada na função. Ele retorna um **valor booleano**.

Exemplo:

```
const numeros = [10, 15, 20];
const temNumeroMaiorQue15 = numeros.some(num => num > 15);
console.log(temNumeroMaiorQue15); // true
```

O método `.some()` verifica se pelo menos um elemento do array atende à condição definida na função. No exemplo, a condição é se há um número maior que 15, e o método retorna `true` porque o número `20` satisfaz a condição.

8. `.every()`

O método `.every()` verifica se **todos os elementos** do array satisfazem a condição passada na função. Ele também retorna um **valor booleano**.

Exemplo:

```
const numeros = [10, 15, 20];
const todosSaoMaioresQue5 = numeros.every(num => num > 5);
console.log(todosSaoMaioresQue5); // true
```

O método `.every()` verifica se todos os elementos do array satisfazem a condição definida na função. No exemplo, a condição é se todos os números são maiores que 5, e o método retorna `true`, pois todos os valores do array atendem à condição.

Objetivos de Aprendizagem da Seção

Dessa seção de Métodos de Array, mais uma vez, não é preciso saber decorado todos eles. O importante é conhecer a existência deles e como usá-los para resolver um problema. Alguns métodos que devem receber maior atenção são OS: `.length`, `.pop()`, `.push()`, `.map()`, `.filter()` e `.forEach()`

Interagindo com o HTML e o CSS

A **manipulação do DOM** (Document Object Model) em JavaScript permite que você interaja e altere o conteúdo, estrutura e estilo de uma página HTML dinamicamente. O DOM representa a página como uma árvore de nós, onde cada elemento, atributo e texto na página é um nó. Portanto, para acessar cada um desses nós em nosso código em JavaScript, precisamos aplicar alguns métodos e atribuir seu retorno a uma variável no código. A seguir, estão alguns dos métodos mais convencionais para acessar esses elementos no DOM.

⚠ IMPORTANTE: São vários métodos e muitos detalhes. Não há a necessidade de decorar todos, na verdade, não há necessidade de decorar praticamente nenhum. O que interessa é entender como funciona o uso de métodos, saber da existência deles e, quando for preciso, escolher e saber aprender como usá-los para resolver o seu problema.

1. `getElementById`

A função `getElementById()` permite acessar um elemento HTML específico através de seu **ID**. O ID de um elemento deve ser único dentro da página.

Exemplo:

```
<p id="mensagem">Olá, Mundo!</p>
```

```
const paragrafo = document.getElementById("mensagem");
console.log(paragrafo); // <p id="mensagem">Olá, Mundo!</p>
```

Aqui, o método encontra o elemento com o ID `mensagem` e o armazena na variável `paragrafo`. Com isso, você pode manipular esse elemento no código JavaScript.

2. `getElementsByClassName`

O método `getElementsByClassName()` retorna todos os elementos de uma página HTML que possuem uma determinada **classe**. Esse método retorna uma coleção de elementos (HTMLCollection).

Exemplo:

```
<p class="texto">Texto 1</p>
<p class="texto">Texto 2</p>
```

```
const paragrafos = document.getElementsByClassName("texto");
console.log(paragrafos); // HTMLCollection [<p class="texto">Texto 1</p>, <p class="texto">Te
```

```
texto 2</p>]
```

Você pode acessar os itens dessa coleção usando índices:

```
console.log(paragrafos[0].textContent); // "Texto 1"
```

3. `getElementsByName`

O método `getElementsByName()` retorna todos os elementos de uma página HTML com um nome de **tag** específico.

Exemplo:

```
<p>Parágrafo 1</p>
<p>Parágrafo 2</p>
```

```
const paragrafos = document.getElementsByTagName("p");
console.log(paragrafos); // HTMLCollection [<p>Parágrafo 1</p>, <p>Parágrafo 2</p>]
```

4. `querySelector`

O `querySelector()` é um método mais flexível e poderoso, pois permite selecionar elementos usando **seletor CSS** (ID, classe, tag, etc.). Ele retorna o **primeiro** elemento que corresponde ao seletor.

Exemplo:

```
<p class="texto">Texto 1</p>
<p class="texto">Texto 2</p>
```

```
const primeiroParagrafo = document.querySelector(".texto");
console.log(primeiroParagrafo); // <p class="texto">Texto 1</p>
```

No exemplo, `querySelector(".texto")` retorna o primeiro elemento com a classe `.texto`.

Você também pode usar outros seletores CSS, como ID ou tag:

```
document.querySelector("#mensagem");
document.querySelector("p");
```

- `# exemplo`: Refere-se e retorna o primeiro elemento com ID "exemplo"
- `. exemplo`: Refere-se e retorna o primeiro elemento com classe "exemplo"
- `p`: Refere-se e retorna o primeiro elemento com a tag escolhida, no caso `<p></p>`. Poderia ser `h1` para puxar o primeiro `<h1></h1>`, `footer` para puxar o primeiro `<footer></footer>` e assim por diante.

5. `querySelectorAll`

O método `querySelectorAll()` retorna todos os elementos que correspondem ao seletor CSS fornecido, mas **em forma de NodeList**, que é semelhante a um array.

Exemplo:

```
<p class="texto">Texto 1</p>
<p class="texto">Texto 2</p>
```



```
const todosParagrafos = document.querySelectorAll(".texto");
console.log(todosParagrafos); // NodeList [<p class="texto">Texto 1</p>, <p class="texto">Texto 2</p>]

//Agora, você pode percorrer o NodeList como se fosse um array:
todosParagrafos.forEach(paragrafo => {
  console.log(paragrafo.textContent);
});
```

6. `textContent`

O `textContent` é uma propriedade usada para **obter ou alterar o texto** dentro de um elemento. Ele retorna ou define o conteúdo textual de um elemento (sem incluir HTML).

Exemplo:

```
<p id="mensagem">Olá!</p>
```

```
const paragrafo = document.getElementById("mensagem");
console.log(paragrafo.textContent); // "Olá!"

paragrafo.textContent = "Novo texto!";
console.log(paragrafo.textContent); // "Novo texto!"
```

7. `innerHTML`

O `innerHTML` é semelhante ao `textContent`, mas ele permite **obter ou alterar o conteúdo HTML** dentro de um elemento, ou seja, ele retorna o **HTML** que está dentro de um elemento (não apenas o texto, mas tags HTML também).

Exemplo:

```
<div id="caixa">
  <p>Texto dentro de um parágrafo.</p>
</div>
```

```
const caixa = document.getElementById("caixa");
console.log(caixa.innerHTML); // "<p>Texto dentro de um parágrafo.</p>"

caixa.innerHTML = "<p>Novo conteúdo com <strong>HTML</strong>!</p>";
console.log(caixa.innerHTML); // "<p>Novo conteúdo com <strong>HTML</strong>!</p>"
```

Diferença entre `textContent` e `innerHTML`

- `textContent`: Acessa ou altera apenas o **texto** dentro de um elemento. Muito mais recomendado por ser mais otimizado e performático.
- `innerHTML`: Acessa ou altera o **HTML** dentro de um elemento, ou seja, ele pode incluir tags HTML. Pouco recomendado.

8. `createElement`

O método `createElement()` permite criar novos **elementos HTML** no DOM. Ele não adiciona o elemento na página, apenas o cria em memória.

Exemplo:

```
const novoParagrafo = document.createElement("p");
novoParagrafo.textContent = "Este é um novo parágrafo!";
document.body.appendChild(novoParagrafo);
```

Aqui, um novo parágrafo (`<p>`) é criado, seu texto é definido com `textContent`, e depois ele é adicionado ao final do corpo da página com `appendChild()`.

9. `remove`

O método `remove()` é usado para **remover um elemento** do DOM. Esse método pode ser chamado diretamente no elemento que você deseja excluir.

Exemplo:

```
<p id="paragrafo">Este parágrafo será removido.</p>
```

```
const paragrafo = document.getElementById("paragrafo");
paragrafo.remove();
```

No exemplo acima, o parágrafo com o ID `paragrafo` é removido da página.

10. `removeChild`

O `removeChild()` permite **remover um filho** de um elemento. Para usar esse método, você precisa especificar qual filho (nó) deseja remover.

Exemplo:

```
<div id="caixa">
  <p id="paragrafo">Este parágrafo será removido da caixa.</p>
</div>
```

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

Neste caso, o parágrafo é removido da `div` com o ID `caixa`.

11. `appendChild`

O método `appendChild()` é usado para **adicionar um elemento** ao final de um elemento pai. Ele pode ser usado para adicionar novos nós ou elementos já existentes.

Exemplo:

```
const lista = document.createElement("ul");
const item1 = document.createElement("li");
item1.textContent = "Item 1";
const item2 = document.createElement("li");
item2.textContent = "Item 2";

lista.appendChild(item1);
lista.appendChild(item2);
```

```
document.body.appendChild(lista);
```

Aqui, uma lista (``) é criada, dois itens de lista (``) são adicionados a ela, e depois a lista é adicionada ao corpo da página.

12. `replaceChild`

O método `replaceChild()` substitui um filho de um elemento por outro. Ele recebe dois parâmetros: o novo nó e o nó que será substituído.

Exemplo:

```
<div id="caixa">
  <p id="paragrafo">Este parágrafo será substituído.</p>
</div>
```

```
const caixa = document.getElementById("caixa");
const novoParagrafo = document.createElement("p");
novoParagrafo.textContent = "Este é o novo parágrafo!";

const paragrafoAntigo = document.getElementById("paragrafo");
caixa.replaceChild(novoParagrafo, paragrafoAntigo);
```

Neste exemplo, o parágrafo antigo é substituído por um novo parágrafo na `div` com o ID `caixa`.

13. Adicionando CSS com JavaScript

Você pode adicionar estilos CSS diretamente a elementos no DOM usando a propriedade `style`. Isso permite definir estilos diretamente no código JavaScript.

Exemplo:

```
const paragrafo = document.createElement("p");
paragrafo.textContent = "Este parágrafo tem estilo!";
document.body.appendChild(paragrafo);

// Adicionando estilo
paragrafo.style.color = "blue";           // Cor do texto
paragrafo.style.fontSize = "20px";        // Tamanho da fonte
paragrafo.style.backgroundColor = "yellow"; // Cor de fundo
```

Aqui, o parágrafo criado é estilizado com uma cor de texto azul, fonte de tamanho 20px e fundo amarelo.

14. Usando `classList` para adicionar/remover classes CSS

Se você preferir trabalhar com classes CSS (em vez de definir os estilos diretamente com a propriedade `style`), pode usar a propriedade `classList` para adicionar, remover ou alternar classes.

Exemplo:

```
<style>
  .destaque {
    color: red;
    font-weight: bold;
  }
</style>
```

```
}  
</style>
```

```
const paragrafo = document.createElement("p");  
paragrafo.textContent = "Este parágrafo vai ganhar uma classe!";  
document.body.appendChild(paragrafo);  
  
// Adicionando a classe CSS  
paragrafo.classList.add("destaque");
```

Aqui, o parágrafo criado recebe a classe CSS `destaque`, que aplica um estilo de texto vermelho e em negrito.

15. Adicionando um evento de clique com `addEventListener`

O método `addEventListener` permite que você escute eventos em elementos HTML e execute uma função quando o evento ocorre. O evento `click` é usado para capturar cliques do usuário em um elemento específico. Detalhe: é declarada uma função para descrever o que acontecerá após o botão ser clicado utilizando uma Arrow Function.

Exemplo:

```
const botao = document.createElement("button");  
botao.textContent = "Clique aqui!";  
document.body.appendChild(botao);  
  
// Adicionando evento de clique  
botao.addEventListener("click", () => {  
  alert("Você clicou no botão!");  
});
```

Aqui, um botão é criado dinamicamente com o texto `"Clique aqui!"`. Usando `addEventListener`, é adicionado um evento de clique ao botão. Quando o botão é clicado, a função de callback exibe um alerta com a mensagem `"Você clicou no botão!"`.

Objetivos de Aprendizagem da Seção

Dessa seção de DOM, mais uma vez, não é preciso saber decorado todos eles. O importante é conhecer a existência deles e como usá-los para resolver um problema. Alguns métodos que devem receber maior atenção são os:

`.getElementById()`, `.querySelector()`, `.createElement`, `.addEventListener()`, `.remove()`, `classList` e `textContent`



DESAFIO DA CAPACITAÇÃO

PASSO A PASSO DO QUE DEVE SER FEITO:

1. Criar uma nova pasta e um novo repositório no Github vinculado a essa pasta. Ao criar o nome do repositório para esse novo projeto, é recomendado desta vez que se escreva o nome do que se trata o projeto (Calculadora, Countdown, To-do List, etc.).
2. Escolher um dos seguintes projetos que vêm acompanhados de um vídeo tutorial:
 - Calculadora IMC
 - Lâmpada
 - Semáforos
 - Calculadora Convencional
 - Countdown
 - To-do List
3. O desafio é fazer um desses projetos utilizando, claro, JavaScript, mas também HTML e CSS.
4. O tutorial explica passo-a-passo e vocês devem segui-lo obviamente, porém, ao sentirem-se confiantes, tentem escrever as linhas de código sozinhos, porque isso trará maior familiaridade com a linguagem e reforçará o que foi aprendido.
5. Para o HTML e o CSS, podem se basear no que é apresentado no vídeo, porém personalizem com outras cores e formatos que desejarem.
6. Se desafiem. Tentem fazer um projeto mais complicado se sentirem mais à vontade com o conteúdo. Deixem o seu projeto bonito, afinal isso é sempre importante em projetos reais.

▲ **IMPORTANTE:** Vocês podem escolher outro projeto que encontrarem na Internet, porém precisam avisar antecipadamente. Alguns projetos na Internet são de fato interessantes, mas as habilidades necessárias podem ser completamente diferentes dos que as que estão sendo almejadas de serem desenvolvidas aqui, nesta capacitação.



A entrega do desafio é apenas o Repositório + Vídeo explicativo

Basta enviar o link do repositório no Github onde todo esse processo ocorreu junto a um vídeo que não precisa ser muito longo (duração de 5 até 10 minutos), explicando os conceitos mais importantes de forma concisa e demonstrando onde esses conceitos foram aplicados no desafio. O vídeo deve ser colocado em sua pasta no Drive, dentro de uma pasta com nome "Capacitação em JavaScript".

Entregas:

- Link do Repositório.
- Vídeo explicativo (duração de 5 minutos até 10 minutos).

