Manipulando o DOM com `getElementById` em JavaScript

1. INTRODUÇÃO À MANIPULAÇÃO DO DOM

O Document Object Model (DOM) é a representação estruturada de uma página web, funcionando como uma ponte que permite ao JavaScript interagir e modificar dinamicamente o conteúdo, a estrutura e o estilo de um documento HTML. Para que essa interatividade seja possível, primeiro precisamos acessar os elementos específicos da página. Este guia se concentrará no método fundamental para essa tarefa: **`getElementById`**.

Uma distinção crucial para qualquer desenvolvedor é a diferença entre a execução de código no lado do cliente (navegador) e no lado do servidor (como em ambientes Node.js). A manipulação do DOM é uma tarefa exclusiva do lado do cliente. Por essa razão, é fundamental entender que toda manipulação do DOM **só pode** ser testada no console do seu navegador, pois ambientes de servidor como o Node.js não têm acesso a essa estrutura. Para os exemplos a seguir, utilizaremos a estrutura HTML abaixo, que contém seis elementos `div` com identificadores únicos.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <title>Aula 30 - getElementById</title>
</head>
<body>
    <div id="c1">HTML</div>
    <div id="c2">CSS</div>
    <div id="c3">JavaScript</div>
    <div id="c4">PHP</div>
    <div id="c5">React</div>
    <div id="c6">MySQL</div>
    <script src="script.js"></script>
</body>
</html>
```

Para entender como o JavaScript "enxerga" e acessa esses elementos, é útil visualizar o DOM não como um bloco de código, mas como uma árvore hierárquica.

1.1. O DOM como uma Árvore de Elementos

Visualizar a estrutura do DOM como uma árvore hierárquica é fundamental para compreender as relações de parentesco (pais e filhos) entre os elementos. Essa perspectiva nos ajuda a navegar e a selecionar os componentes da página de forma lógica e eficiente. Nessa analogia, o elemento `<html>` é o **nó raiz** (`root`) de toda a estrutura. Diretamente ligados a ele, temos seus filhos imediatos, que são os elementos `<head>` e `<body>`. Por sua vez, o `<body>` possui seus próprios filhos, que em nosso exemplo são as seis `divs` e o elemento `<script>`. Podemos representar visualmente essa hierarquia da seguinte forma:

- `<html>` (Nó Raiz)
  - `<head>`
  - `<body>`
    - `<div>` (c1)
    - `<div>` (c2)
    - `<div>` (c3)
    - `<div>` (c4)
    - `<div>` (c5)
    - `<div>` (c6)
    - `<script>`

Para interagir com qualquer "galho" ou "folha" dessa árvore, precisamos de métodos específicos que nos permitam apontar para o elemento desejado. A ferramenta mais direta para selecionar um elemento único é o `getElementById`, que exploraremos a seguir.

1.2. Selecionando Elementos com `document.getElementById()`

O método **`getElementById()`** é uma das formas mais diretas e eficientes de obter um elemento específico da página. Como o nome sugere, sua operação depende de um atributo `id` que deve ser único em todo o documento HTML. A sintaxe para capturar um elemento é simples. Para selecionar a `div` com `id="c1"` e armazená-la em uma constante, utilizamos o seguinte código:

```
const dc1 = document.getElementById('c1');
```

Após a execução dessa linha, a constante `dc1` não contém apenas o texto do elemento, mas uma referência ao objeto completo do elemento `div`. Isso significa que temos acesso a todas as suas propriedades e métodos. Para inspecionar o que foi capturado, podemos usar `console.log(dc1)`. Uma dica prática: na maioria dos consoles de navegador, ao passar o mouse sobre o elemento logado, ele será destacado visualmente na própria página, reforçando a conexão entre o código e a interface. Uma vez que o elemento está capturado em uma variável, podemos ir além da simples leitura e começar a modificar suas propriedades e seu conteúdo de forma dinâmica.

1.3. Acessando e Modificando Propriedades do Elemento

Com um elemento do DOM armazenado em uma variável, um leque de propriedades se torna acessível. Duas das mais comuns para inspeção e modificação são o `id` (o identificador único) e o `innerHTML` (o conteúdo HTML interno do elemento).

1.3.1. Acessando Propriedades

Podemos ler facilmente as propriedades de um elemento capturado. O código abaixo imprime no console o objeto do elemento inteiro, seu `id` e seu conteúdo `innerHTML`.

```
// Exemplo para acessar e exibir propriedades
console.log(dc1);
console.log(dc1.id);
console.log(dc1.innerHTML);
```

Saída esperada no console:

1. O objeto do elemento `div` (`<div id="c1">HTML</div>`).
2. A string `"c1"`.
3. A string `"HTML"`.

1.3.2. Modificando Propriedades

O verdadeiro poder da manipulação do DOM reside na capacidade de alterar essas propriedades em tempo real. O exemplo a seguir modifica o conteúdo da `div c1`.

```
// Exemplo para modificar o conteúdo
dc1.innerHTML = 'mudando conteúdo';
```

Essa alteração é refletida imediatamente na página visível para o usuário. É importante entender que o JavaScript executa o código de forma síncrona, linha por linha. Se tivéssemos um `console.log(dc1.innerHTML)` antes da linha de modificação, ele imprimiria o valor original ("HTML"). A linha seguinte, `dc1.innerHTML = 'mudando conteúdo'`, executa imediatamente depois, atualizando o que o usuário vê na tela. Mas como podemos aplicar essas modificações a múltiplos elementos de forma eficiente, sem repetir o código para cada um?

1.4. Trabalhando com Múltiplos Elementos em um Array

Para manipular um grupo de elementos, a estratégia mais eficaz é selecioná-los individualmente com `getElementById` e, em seguida, organizá-los em uma estrutura de dados, como um array. Isso nos permite iterar sobre eles e aplicar modificações em massa. O bloco de código a seguir demonstra esse processo completo:

```
// 1. Capturar cada elemento individualmente
const dc1 = document.getElementById('c1');
const dc2 = document.getElementById('c2');
const dc3 = document.getElementById('c3');
const dc4 = document.getElementById('c4');
const dc5 = document.getElementById('c5');
const dc6 = document.getElementById('c6');

// 2. Criar um array contendo todos os elementos
const arrayElementos = [dc1, dc2, dc3, dc4, dc5, dc6];

// 3. Exibir o array no console para inspeção
console.log(arrayElementos);
```

Ao executar o `console.log`, o navegador exibirá um *array* contendo os seis objetos de elemento `div`. Agora, em vez de seis variáveis separadas, temos uma única coleção ordenada e iterável, pronta para ser manipulada em lote. Com nossos elementos organizados em um *array*, podemos utilizar diferentes abordagens de loop para percorrer essa coleção e aplicar a mesma operação a todos os seus itens.

1.5. Métodos de Iteração para Manipulação em Massa

Para evitar a repetição de código ao modificar vários elementos, utilizamos laços de repetição (iteração). Vamos analisar as abordagens mais comuns e eficientes para trabalhar com arrays de elementos do DOM.

1.5.1. Opção 1: O Laço `for...of`

O laço `for...of` é uma estrutura de controle de fluxo moderna e legível, ideal para percorrer coleções iteráveis como *arrays*. Sua sintaxe é clara e direta para aplicar uma operação a cada item da coleção.

```
const arrayElementos = [dc1, dc2, dc3, dc4, dc5, dc6];

for (const d of arrayElementos) {
    d.innerHTML = 'alterando elementos';
}
```

Após a execução deste código, o conteúdo de todas as seis `divs` na página será alterado para "`alterando elementos`".

1.5.2. Opção 2: Métodos de Array Modernos (`forEach`, `map`)

O JavaScript moderno oferece métodos de array que promovem um estilo de programação mais funcional e declarativo. Os dois mais relevantes para esta tarefa são `.forEach()` e `.map()`.

1.5.2.1. `forEach()`: A Escolha Correta para Efeitos Colaterais

O método `.forEach()` executa uma função para cada elemento do array. É a ferramenta semanticamente correta quando seu objetivo é realizar uma ação (um "efeito colateral"), como modificar o `innerHTML` de cada elemento, e você não precisa criar um novo array como resultado.

```
const arrayElementos = [dc1, dc2, dc3, dc4, dc5, dc6];

arrayElementos.forEach((e) => {
    e.innerHTML = 'alterando elementos';
});
```

Este código alcança o mesmo resultado do `for...of`, mas de uma forma que muitos desenvolvedores consideram mais limpa e expressiva.

1.5.2.2. `map()`: Para Transformação de Arrays

O método `.map()` também itera sobre cada elemento, mas seu propósito principal é **criar um novo array** a partir dos resultados da função aplicada a cada item. Usá-lo apenas para efeitos colaterais, como no exemplo abaixo, é um anti-padrão comum.

```
// Exemplo funcional, mas não idiomático
arrayElementos.map((e) => {
    e.innerHTML = 'alterando elementos';
});
```

Embora o código acima funcione, ele é um uso inadequado do `.map()`. Ele está modificando os elementos originais, mas também está criando e retornando um novo array (neste caso, `[undefined, undefined, ...]`) que não está sendo utilizado. Para a tarefa de simplesmente modificar elementos, `.forEach()` é a escolha superior e mais clara. Em resumo, ao trabalhar com arrays, prefira métodos modernos como `.forEach()` a laços tradicionais. Eles representam uma "estrutura mais simples e mais moderna", são menos propensos a erros e comunicam melhor a sua intenção: `forEach` para ações, `map` para transformações.

2. CONCLUSÃO E PRÓXIMOS PASSOS

Ao longo desta seção, aprendemos os conceitos fundamentais para a manipulação do DOM. Vimos o que é o DOM e por que ele é representado como uma árvore, como selecionar um elemento único com `document.getElementById`, como acessar e modificar suas propriedades (`id`, `innerHTML`) e, finalmente, como escalar essa manipulação para múltiplos elementos, organizando-os em um array e utilizando laços como `for...of` e métodos como `.forEach()` para aplicar alterações em massa.

Dominamos a seleção de elementos por IDs únicos. No entanto, o JavaScript oferece outros métodos poderosos, como `getElementsByClassName` ou `querySelectorAll`, que selecionam múltiplos elementos de uma vez. É crucial saber que esses métodos retornam uma `HTMLCollection` ou `NodeList`, que são objetos "análogos a um array" (*array-like*), mas ***não são arrays de verdade***. Consequentemente, eles não possuem métodos como `.forEach()` ou `.map()` diretamente. É por isso que técnicas como o operador *spread* (`[...colecao]`) são essenciais para primeiro converter essas coleções em arrays verdadeiros, desbloqueando todo o poder dos métodos de iteração modernos.