Aula 15 – Manipulação do DOM

Imagine que você está navegando na internet, e de repente, um formulário de login aparece, ou um item é adicionado ao seu carrinho de compras sem que a página inteira recarregue. Ou talvez você clique em um botão e um novo parágrafo de texto surge magicamente. Por trás dessas interações dinâmicas, existe uma ferramenta poderosa que permite ao JavaScript "conversar" com a estrutura da página web: o Document Object Model, ou DOM.

Dominar a manipulação do DOM é como ganhar superpoderes para dar vida às suas páginas estáticas. É a habilidade de transformar um site de um mero documento informativo em uma experiência interativa e responsiva, que reage às ações do usuário em tempo real. Sem essa capacidade, a web seria um lugar muito menos interessante, com páginas que se comportam como livros digitais, sem qualquer dinamismo.

Nesta aula, vamos desvendar os segredos do DOM, explorando como ele funciona e, mais importante, como você pode usá-lo para criar interfaces ricas e envolventes. Ao final, você será capaz de selecionar elementos específicos na sua página, alterar seu conteúdo e estilo, e até mesmo criar e remover componentes dinamicamente, preparando o terreno para construir aplicações web verdadeiramente interativas. Prepare-se para dar os primeiros passos na construção de experiências digitais que respondem e se adaptam.

O Que é o DOM (Document Object Model)?

Quando você abre uma página web no seu navegador, ele não apenas exibe o código HTML que você escreveu. Na verdade, o navegador constrói uma representação interna desse código, uma espécie de mapa interativo da sua página. Essa representação é o Document Object Model, ou DOM. Pense no DOM como a "API" (Interface de Programação de Aplicações) para o conteúdo da web, permitindo que linguagens de script, como o JavaScript, acessem e modifiquem a estrutura, o estilo e o conteúdo de um documento HTML.

> **O DOM é uma representação viva da sua página.** Ao contrário do código HTML estático, o DOM pode ser alterado dinamicamente pelo JavaScript, permitindo interações em tempo real sem recarregar a página.

O DOM organiza todos os elementos da sua página – cada parágrafo, imagem, link, botão – em uma estrutura hierárquica, semelhante a uma árvore genealógica. Cada parte da sua página se torna um "nó" nessa árvore, e esses nós podem ser de diferentes tipos: elementos (como ou <p>), atributos (como src de uma imagem), ou texto. Essa estrutura em árvore é fundamental porque define as relações entre os elementos: quem é pai de quem, quem são irmãos, e assim por diante.

A grande sacada do DOM é que ele não é estático; ele é uma representação *viva* da sua página. Isso significa que, ao contrário do seu código HTML original que é fixo, o DOM pode ser alterado dinamicamente pelo JavaScript. Você pode adicionar novos elementos, remover outros, mudar o texto de um parágrafo, ou até mesmo alterar a cor de um botão, tudo isso sem precisar recarregar a página. Essa capacidade de manipulação em tempo real é o que torna as aplicações web modernas tão fluidas e interativas.

DOM: O Mapa Interativo da Sua Página

Analogia da Casa

Imagine sua página HTML como a planta baixa de uma casa. O HTML descreve onde as paredes, portas e janelas estão. O DOM, por sua vez, é como um modelo 3D interativo dessa casa, que você pode tocar, mover móveis, pintar paredes e até adicionar ou remover cômodos, tudo enquanto a casa está "em uso".

O Papel do JavaScript

O JavaScript é o construtor ou decorador que usa esse modelo 3D para fazer as mudanças. Ele navega pela estrutura, encontra exatamente o que precisa e realiza modificações precisas.

Nós de Elemento

Cada tag HTML (<body>, <h1>, <p>, <a>) se torna um nó de elemento no DOM.

Nós de Texto

O texto dentro das tags se torna um nó de texto, representando o conteúdo visível.

Nós de Atributo

Os atributos (id, class, href) se tornam nós de atributo, armazenando informações extras.

A beleza dessa estrutura reside na sua flexibilidade. Se você quer mudar o título de uma seção, o JavaScript pode encontrar o nó <h1> correspondente e atualizar seu nó de texto. Se quer adicionar um novo item a uma lista, ele pode criar um novo nó <li> e anexá-lo ao nó <ul> existente. Essa capacidade de interagir com a página em um nível tão granular é o que nos permite construir experiências de usuário ricas e dinâmicas, que respondem instantaneamente às ações do usuário.

Selecionando Elementos: Encontrando o Caminho

Antes de poder manipular qualquer coisa na sua página, você precisa ser capaz de identificar e "selecionar" os elementos que deseja modificar. É como ter um controle remoto: você precisa apontar para a TV certa antes de poder mudar o canal. O JavaScript oferece várias ferramentas para fazer essa seleção, cada uma útil em diferentes cenários. Vamos explorar as mais comuns e eficazes.

getElementById(): O Identificador Único

Performance Superior

getElementById é otimizado pelos navegadores para busca rápida, pois IDs são únicos.

A maneira mais direta e eficiente de selecionar um único elemento é usando seu atributo id. O id deve ser único em toda a página HTML, funcionando como um RG para cada elemento. Se você sabe o id exato do elemento que quer, getElementById() é a sua melhor aposta.

```
// Suponha que temos um HTML: <h1 id="tituloPrincipal">Bem-vindo!</h1>
const titulo = document.getElementById('tituloPrincipal');
console.log(titulo.textContent); // Saída: Bem-vindo!
```

Esta função retorna o elemento correspondente ao id especificado. Se nenhum elemento com aquele id for encontrado, ela retorna null. É como procurar um livro na biblioteca pelo seu ISBN único; se o ISBN não existe, você não encontra o livro. A performance de getElementById é geralmente superior, pois os navegadores otimizam a busca por IDs.

querySelector(): O Seletor CSS Universal

Para uma seleção mais flexível, o método querySelector() permite que você use qualquer seletor CSS válido para encontrar um elemento. Ele retorna o *primeiro* elemento que corresponde ao seletor especificado. Isso é incrivelmente poderoso, pois você pode selecionar elementos por tag, classe, ID, atributos, ou uma combinação complexa deles.

```
// HTML: <div class="container"><p class="texto">Primeiro parágrafo</p><p class="texto">Segundo parágrafo</p></div>
const primeiroParagrafo = document.querySelector('.texto');
console.log(primeiroParagrafo.textContent); // Saída: Primeiro parágrafo

const containerDiv = document.querySelector('div.container');
console.log(containerDiv); // Retorna o elemento <div> com a classe container
```

Flexibilidade Total

Use qualquer seletor CSS: classes, IDs, tags, atributos, pseudo-classes e combinações complexas.

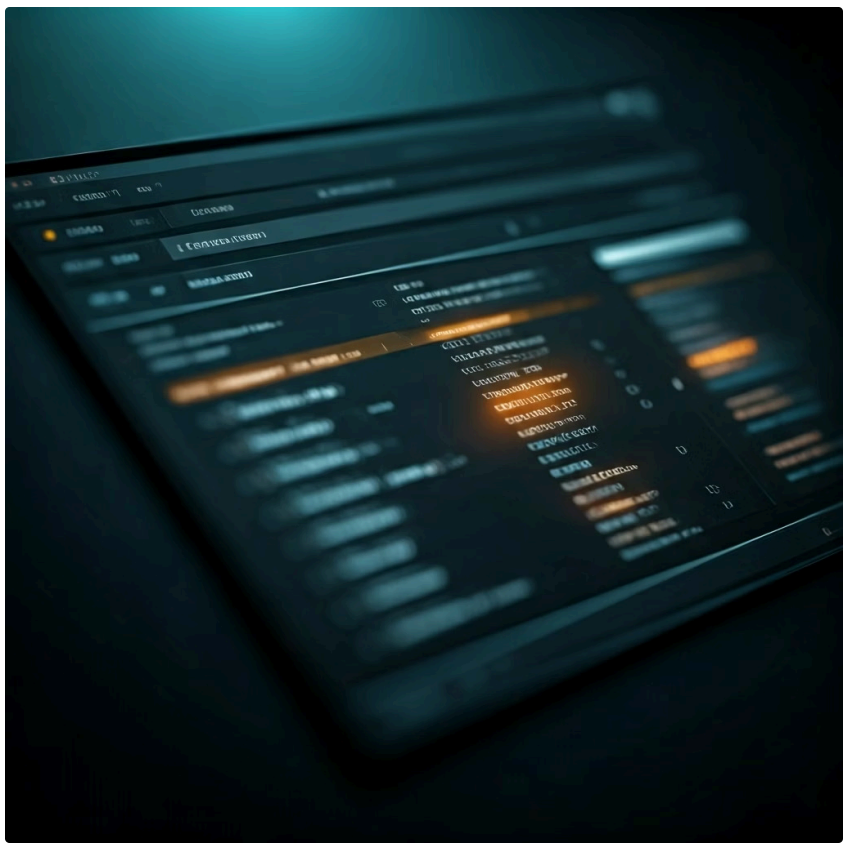
Primeira Correspondência

Retorna apenas o primeiro elemento que corresponde ao seletor. Para múltiplos elementos, use querySelectorAll().

Pense no querySelector() como um detetive que busca a primeira pista que se encaixa na descrição que você deu. Ele é versátil, mas lembre-se: ele para na primeira correspondência. Se você precisa de todos os elementos que correspondem a um seletor, precisará de outra ferramenta.

querySelectorAll(): Capturando Múltiplos Elementos

Quando você precisa interagir com vários elementos que compartilham uma característica, como todos os itens de uma lista ou todos os botões de uma determinada classe, querySelectorAll() é a ferramenta ideal. Assim como querySelector(), ele aceita seletores CSS, mas, em vez de retornar apenas o primeiro, ele retorna uma NodeList (uma coleção de nós) de *todos* os elementos que correspondem ao seletor.



```
// HTML: <ul><li class="item">Maçã</li><li class="item">Banana</li><li class="item">Laranja</li></ul>
const itensLista = document.querySelectorAll('.item');
itensLista.forEach(item => {
  console.log(item.textContent);
});
// Saída: Maçã, Banana, Laranja (cada um em uma linha)
```

NodeList vs Array

A NodeList retornada por querySelectorAll() é semelhante a um array, mas não é um array JavaScript puro. No entanto, você pode iterar sobre ela usando forEach() ou convertê-la em um array com Array.from().

É como pedir a um bibliotecário para listar todos os livros de um determinado gênero; ele não te dá apenas o primeiro, mas sim uma lista completa de todos eles. Essa função é essencial para aplicar ações em massa ou para construir componentes dinâmicos que gerenciam coleções de elementos.

Manipulando Conteúdo: Dando Voz aos Elementos

Uma vez que você selecionou um elemento, o próximo passo lógico é mudar o que ele exibe. A manipulação de conteúdo é uma das tarefas mais comuns e impactantes que você fará com o DOM. Existem duas propriedades principais para isso: textContent e innerHTML, e entender a diferença entre elas é crucial para evitar problemas de segurança e garantir o comportamento esperado.

textContent: O Texto Puro e Simples

Segurança em Primeiro Lugar

textContent trata todo conteúdo como texto puro, prevenindo ataques XSS (Cross-Site Scripting).

Ignora Marcação HTML

Qualquer tag HTML inserida via textContent será exibida como texto, não como elementos renderizados.

```
// HTML: <p id="saudacao">Olá, <span>mundo</span>!</p>
const saudacao = document.getElementById('saudacao');
console.log(saudacao.textContent); // Saída: Olá, mundo!

saudacao.textContent = 'Bem-vindo ao DOM!';
// O HTML agora é: <p id="saudacao">Bem-vindo ao DOM!</p>
```

Usar textContent é a maneira mais segura de inserir texto em um elemento, especialmente se esse texto vier de uma fonte externa (como dados de um usuário ou de uma API). Ele previne ataques de Cross-Site Scripting (XSS), onde código malicioso HTML ou JavaScript pode ser injetado na sua página. É como escrever uma mensagem em um quadro branco: você só pode escrever texto, não pode colar imagens ou outros objetos.

innerHTML: O Poder do HTML Interno

Poder e Responsabilidade

A propriedade innerHTML é mais poderosa e, por isso, exige mais cuidado. Ela permite que você obtenha ou defina o conteúdo HTML completo dentro de um elemento, incluindo tags e texto. Quando você define innerHTML, o navegador interpreta a string fornecida como código HTML e a renderiza.

```
// HTML: <div id="areaConteudo"></div>
const areaConteudo = document.getElementById('areaConteudo');
areaConteudo.innerHTML = '<h2>Novo Título</h2><p>Este é um <strong>novo</strong> parágrafo.</p>';

/* O HTML agora é:
<div id="areaConteudo">
  <h2>Novo Título</h2>
  <p>Este é um <strong>novo</strong> parágrafo.</p>
</div>
*/
```

✔ Vantagem

Extremamente útil para construir blocos complexos de HTML dinamicamente.

⚠️ Cuidado

Risco de XSS se usado com conteúdo não confiável. Use apenas com dados que você controla.

innerHTML é extremamente útil para construir blocos complexos de HTML dinamicamente. No entanto, seu poder vem com um risco: se você inserir conteúdo não confiável (como entrada de usuário) usando innerHTML, você pode abrir uma brecha para ataques XSS. Um usuário mal-intencionado poderia injetar <script> tags que executariam código JavaScript no navegador de outros usuários. Use innerHTML com cautela e apenas com conteúdo que você confia plenamente. É como ter a capacidade de redecorar um cômodo inteiro, adicionando novas paredes, janelas e móveis, mas você precisa ter certeza de que os materiais que está usando são seguros.

Manipulando Atributos e Estilos: Personalizando a Experiência

Além de mudar o conteúdo textual ou HTML de um elemento, o DOM também permite que você altere seus atributos (como src de uma imagem, href de um link, class ou id) e seus estilos CSS. Essa capacidade é fundamental para criar interfaces que se adaptam e respondem visualmente às interações do usuário.

Manipulando Atributos: O DNA dos Elementos

Atributos fornecem informações adicionais sobre os elementos HTML. O JavaScript oferece métodos específicos para gerenciar esses atributos:

getAttribute(nome)

Retorna o valor de um atributo.

setAttribute(nome, valor)

Define ou atualiza o valor de um atributo.

removeAttribute(nome)

Remove um atributo.

```
// HTML: <img id="minhaImagem" src="imagem1.jpg" alt="Uma imagem">
const imagem = document.getElementById('minhaImagem');

console.log(imagem.getAttribute('src')); // Saída: imagem1.jpg

imagem.setAttribute('src', 'imagem2.png');
imagem.setAttribute('alt', 'Nova imagem');
// A imagem agora aponta para imagem2.png e tem um novo texto alternativo.

imagem.removeAttribute('alt');
// O atributo 'alt' foi removido.
```

Esses métodos são essenciais para tarefas como trocar a fonte de uma imagem, desabilitar um botão, ou adicionar um data-attribute para armazenar informações extras. É como mudar as especificações técnicas de um carro: você pode alterar a cor, o tipo de pneu ou adicionar um acessório, tudo isso sem mudar a estrutura básica do carro.

Manipulando Estilos: A Aparência da Página

Para alterar a aparência visual de um elemento, você pode acessar sua propriedade style. Esta propriedade permite definir estilos CSS diretamente no elemento, como se estivesse usando um estilo inline no HTML.

```
// HTML: <button id="meuBotao">Clique-me</button>
const botao = document.getElementById('meuBotao');
botao.style.backgroundColor = 'blue';
botao.style.color = 'white';
botao.style.padding = '10px 20px';
// O botão agora tem fundo azul, texto branco e padding.
```

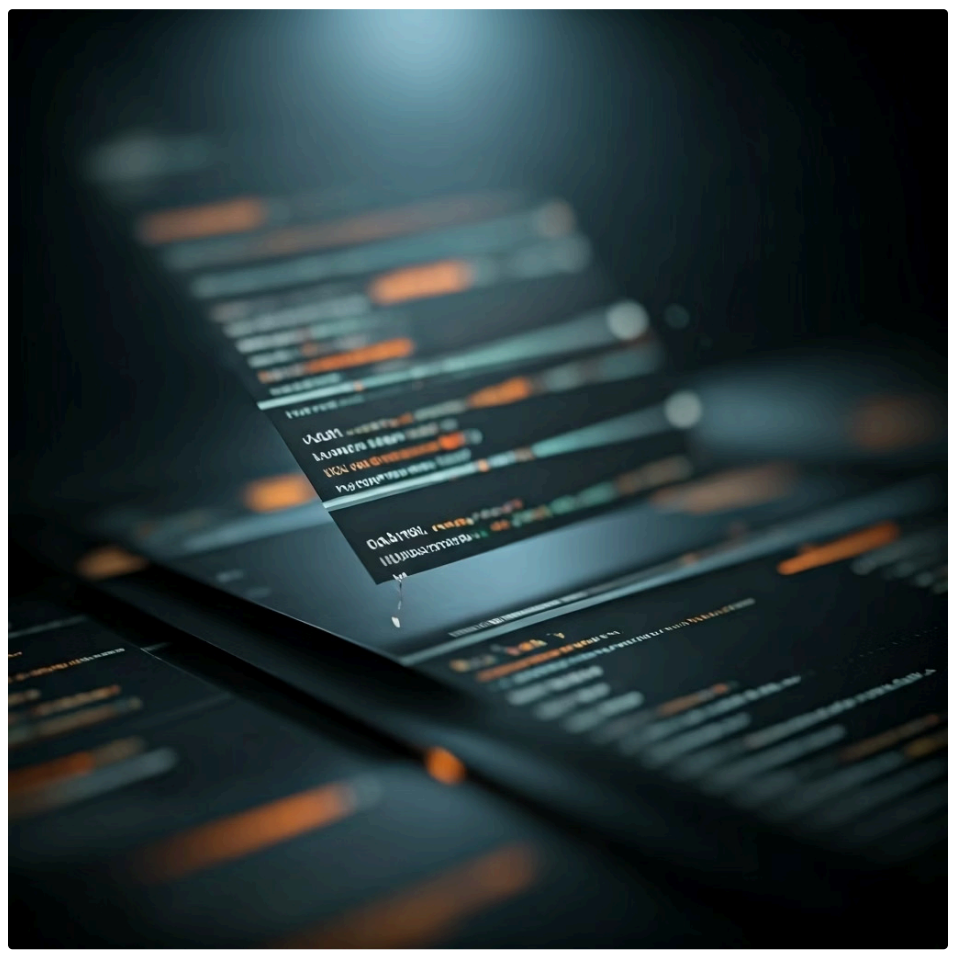
Melhor Prática

Embora seja possível manipular estilos diretamente, uma prática mais recomendada, especialmente para acessibilidade e manutenção, é manipular as classes CSS do elemento.

Manipulando Classes: A Maneira Inteligente de Estilizar

Em vez de alterar propriedades de estilo individuais, é geralmente melhor adicionar ou remover classes CSS de um elemento. Isso permite que você defina estilos complexos em seu arquivo CSS e simplesmente "ligue" ou "desligue" esses estilos via JavaScript.

A propriedade classList oferece métodos convenientes para isso:



add(classe)

Adiciona uma ou mais classes.

remove(classe)

Remove uma ou mais classes.

toggle(classe)

Adiciona a classe se ela não existe, remove se existe.

contains(classe)

Verifica se a classe existe.

```
// CSS: .destaque { background-color: yellow; font-weight: bold; }
// HTML: <p id="paragrafo">Este é um parágrafo.</p>

const paragrafo = document.getElementById('paragrafo');
paragrafo.classList.add('destaque');
// O parágrafo agora tem o estilo da classe 'destaque'.

if (paragrafo.classList.contains('destaque')) {
  console.log('Parágrafo está em destaque.');
}

paragrafo.classList.remove('destaque');
// O estilo de destaque foi removido.

paragrafo.classList.toggle('ativo');
// Adiciona a classe 'ativo'. Se fosse chamado novamente, removeria.
```

Manipular classes é como trocar a roupa de um personagem: você não precisa redesenhar cada peça de roupa, apenas troca o conjunto. Isso mantém seu código JavaScript mais limpo e seu CSS mais organizado, além de ser mais eficiente para o navegador. É uma prática fundamental para garantir que a acessibilidade (A11Y) seja mantida, pois as classes podem ser semanticamente significativas e os estilos definidos de forma consistente.

Criando e Removendo Elementos Dinamicamente

Construindo a Página em Tempo Real

A capacidade de adicionar e remover elementos da página em tempo de execução é o que realmente transforma um site estático em uma aplicação web dinâmica. Seja para adicionar um novo item a uma lista de tarefas, exibir mensagens de erro, ou carregar conteúdo sob demanda, essas operações são o coração da interatividade.

Criando Novos Elementos: createElement()

O primeiro passo para adicionar um novo elemento é criá-lo. O método document.createElement() faz exatamente isso: ele cria um novo nó de elemento HTML, mas ainda não o insere na página.

Na Memória

O elemento criado existe apenas na memória até ser anexado ao DOM.

```
// Criando um novo parágrafo
const novoParagrafo = document.createElement('p');
novoParagrafo.textContent = 'Este é um parágrafo criado dinamicamente!';
novoParagrafo.classList.add('mensagem-info');

// Neste ponto, 'novoParagrafo' existe na memória, mas não está visível na página.
```

Pense em createElement() como montar uma peça de Lego fora da estrutura principal. Você a constrói, dá suas características (texto, classes, atributos), mas ela ainda não faz parte do modelo final. Para que ela apareça na página, você precisa "anexá-la" a um elemento existente.

Inserindo Elementos na Página

Depois de criar um elemento, você precisa decidir onde ele será colocado na árvore DOM.

appendChild()

Anexa um elemento como o *último filho* de um elemento pai.

insertBefore()

Insere um elemento *antes* de um elemento de referência que já é filho do pai.

Exemplo com appendChild()

```
// HTML: <div id="container"></div>
const container = document.getElementById('container');

// Usando appendChild
const itemLista = document.createElement('li');
itemLista.textContent = 'Item 1';
container.appendChild(itemLista); // Adiciona Item 1 como último filho do container

const outroItem = document.createElement('li');
outroItem.textContent = 'Item 2';
container.appendChild(outroItem); // Adiciona Item 2 depois de Item 1
```

Exemplo com insertBefore()

```
// Supondo que 'container' já tem 'itemLista' e 'outroItem'
const primeiroItem = document.createElement('li');
primeiroItem.textContent = 'Item Zero';

// Insere 'primeiroItem' antes de 'itemLista' (que é o 'Item 1')
container.insertBefore(primeiroItem, itemLista);
```

appendChild()

É como colocar um novo livro no final de uma prateleira.

insertBefore()

É como inserir um livro em um lugar específico da prateleira, empurrando os outros para o lado.

A escolha entre eles depende da posição exata onde você quer que o novo elemento apareça.

Removendo Elementos: removeChild() e remove()

Assim como você pode adicionar elementos, também pode removê-los. Isso é útil para limpar a interface, remover itens de uma lista ou fechar mensagens.

removeChild()

Remove um elemento filho de seu elemento pai. Você precisa ter uma referência ao pai e ao filho.

remove()

Um método mais moderno e simples, chamado diretamente no elemento que você deseja remover.

```
// HTML: <ul id="minhaLista"><li>Item A</li><li id="itemB">Item B</li><li>Item C</li></ul>
const minhaLista = document.getElementById('minhaLista');
const itemB = document.getElementById('itemB');

// Usando removeChild (precisa do pai)
minhaLista.removeChild(itemB); // Item B foi removido da lista.

// Criando um novo item para demonstrar remove()
const itemTemporario = document.createElement('li');
itemTemporario.textContent = 'Item Temporário';
minhaLista.appendChild(itemTemporario);

// Usando remove() (chamado diretamente no elemento)
itemTemporario.remove(); // Item Temporário foi removido.
```

Método Preferido

remove() é geralmente preferível por ser mais conciso e não exigir uma referência ao elemento pai. É como simplesmente pegar um objeto e jogá-lo fora.

A capacidade de criar e remover elementos dinamicamente é a espinha dorsal de muitas aplicações web modernas, desde simples listas de tarefas até complexos painéis de controle.

Colocando em Prática: Um Exemplo Interativo Simples

Vamos consolidar o que aprendemos com um pequeno projeto. Imagine que queremos criar uma lista de tarefas simples onde podemos adicionar novos itens. Este exemplo integrará a seleção, criação e manipulação de conteúdo.

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Minha Lista de Tarefas</title>
  <style>
    body { font-family: sans-serif; margin: 20px; }
    #tarefaInput { padding: 8px; margin-right: 5px; }
    #adicionarTarefa { padding: 8px 15px; cursor: pointer; }
    ul { list-style: none; padding: 0; }
    li { background-color: #f4f4f4; margin-bottom: 5px; padding: 10px;
         border-radius: 4px; display: flex; justify-content: space-between;
         align-items: center; }
    .remover { background-color: #ff4d4d; color: white; border: none;
               padding: 5px 10px; border-radius: 3px; cursor: pointer; }
  </style>
</head>
<body>
  <h1>Minha Lista de Tarefas</h1>
  <div>
    <input type="text" id="tarefaInput" placeholder="Nova tarefa...">
    <button id="adicionarTarefa">Adicionar</button>
  </div>
  <ul id="listaDeTarefas">
    <!-- Tarefas serão adicionadas aqui -->
  </ul>

  <script>
    const tarefaInput = document.getElementById('tarefaInput');
    const adicionarBotao = document.getElementById('adicionarTarefa');
    const listaDeTarefas = document.getElementById('listaDeTarefas');

    adicionarBotao.addEventListener('click', () => {
      const textoTarefa = tarefaInput.value.trim();

      if (textoTarefa !== '') {
        // 1. Criar o novo item da lista (li)
        const novoItem = document.createElement('li');

        // 2. Adicionar o texto da tarefa ao item
        novoItem.textContent = textoTarefa;

        // 3. Criar um botão de remover para este item
        const botaoRemover = document.createElement('button');
        botaoRemover.textContent = 'Remover';
        botaoRemover.classList.add('remover');

        // 4. Adicionar um evento de clique ao botão de remover
        botaoRemover.addEventListener('click', () => {
          novoItem.remove();
        });

        // 5. Anexar o botão de remover ao item da lista
        novoItem.appendChild(botaoRemover);

        // 6. Anexar o novo item (com o botão) à lista de tarefas
        listaDeTarefas.appendChild(novoItem);

        // 7. Limpar o campo de input após adicionar a tarefa
        tarefaInput.value = '';
        tarefaInput.focus();
      }
    });
  </script>
</body>
</html>
```

01

Criar elemento

createElement('li') cria o novo item da lista.

02

Adicionar conteúdo

textContent define o texto da tarefa.

03

Criar botão

createElement('button') cria o botão de remover.

04

Adicionar evento

addEventListener conecta a ação de remoção ao botão.

05

Anexar ao DOM

appendChild insere o item completo na lista.

Este exemplo demonstra a criação de elementos (<li>, <button>), manipulação de texto (textContent), adição de classes (classList.add), e a remoção de elementos (remove()). Ele é um microcosmo de como a manipulação do DOM é usada para construir interfaces interativas.

Otimização e Boas Práticas: Construindo com Eficiência

Manipular o DOM é poderoso, mas também pode ser custoso em termos de performance se não for feito com cuidado. Cada alteração no DOM pode levar o navegador a recalcular o layout da página (reflow) e redesenhá-la (repaint), o que pode ser lento, especialmente em páginas complexas ou em dispositivos menos potentes.

Minimizando Reflows e Repaints

Sempre que possível, agrupe suas operações no DOM. Em vez de adicionar elementos um por um em um loop, crie todos os elementos necessários na memória e, em seguida, anexe-os ao DOM de uma vez.

❌ Ruim: Muitos reflows

```
for (let i = 0; i < 1000; i++) {
  const item = document.createElement('li');
  item.textContent = `Item ${i}`;
  listaDeTarefas.appendChild(item);
}
```

✔ Bom: Apenas um reflow

```
const fragmento =
document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  const item = document.createElement('li');
  item.textContent = `Item ${i}`;
  fragmento.appendChild(item);
}
listaDeTarefas.appendChild(fragmento);
```

DocumentFragment

O DocumentFragment é uma ferramenta excelente para isso. Ele atua como um contêiner temporário na memória, onde você pode construir e manipular uma subárvore do DOM sem que o navegador precise renderizá-la. Somente quando o fragmento é anexado ao DOM real, o navegador faz o trabalho de renderização.

Delegação de Eventos

Quando você tem muitos elementos semelhantes (como uma lista de itens) e precisa adicionar um evento a cada um, em vez de adicionar um addEventListener a cada item individualmente, você pode adicionar um único ouvinte de evento ao elemento pai. Isso é chamado de delegação de eventos.

```
// HTML: <ul id="listaDeTarefas"><li>Tarefa 1 <button class="remover">Remover</button></li>...</ul>

listaDeTarefas.addEventListener('click', (event) => {
  // Verifica se o clique foi no botão de remover
  if (event.target.classList.contains('remover')) {
    event.target.closest('li').remove(); // Encontra o <li> pai e o remove
  }
});
```

Menos Ouvintes

Reduz o número de event listeners na memória, melhorando a performance.

Elementos Dinâmicos

Funciona automaticamente com elementos adicionados dinamicamente após o carregamento.

Código Mais Limpo

Simplifica o gerenciamento de eventos em coleções de elementos.

Foco em Ferramentas Modernas e Performance Web

Ferramentas como Vite, mencionadas na introdução do curso, otimizam o processo de desenvolvimento, mas as boas práticas de manipulação do DOM continuam sendo cruciais para o desempenho final da aplicação. O impacto da manipulação do DOM nas Core Web Vitals (métricas de performance do Google) é direto: operações lentas podem atrasar o First Input Delay (FID) e o Largest Contentful Paint (LCP). Ao agrupar operações e usar delegação de eventos, você contribui para uma experiência de usuário mais fluida e um melhor ranqueamento em performance.

Acessibilidade (A11Y) e DOM: Construindo para Todos

Inclusão Digital

A acessibilidade não é um extra, mas um pilar fundamental no desenvolvimento web. Ao manipular o DOM, é crucial garantir que as mudanças que você faz não prejudiquem a experiência de usuários com deficiência. O DOM é a representação que tecnologias assistivas, como leitores de tela, usam para entender a estrutura e o conteúdo da página.

Semântica é Chave

Sempre que possível, use elementos HTML semanticamente corretos (<button>, <nav>, <main>, <h1> a <h6>, etc.) em vez de ou genéricos. Quando você manipula o DOM, mantenha essa semântica. Se você está criando um botão, crie um elemento <button>, não um com um evento de clique.

❌ Ruim para acessibilidade

```
const divBotao = document.createElement('div');
divBotao.textContent = 'Clicar';
divBotao.addEventListener('click', () =>
alert('Clicou!'));
// Usuários de teclado e leitores de tela
// podem ter dificuldade em interagir com isso.
```

✔ Bom para acessibilidade

```
const botaoReal =
document.createElement('button');
botaoReal.textContent = 'Clicar';
botaoReal.addEventListener('click', () =>
alert('Clicou!'));
// O navegador já sabe que é um botão,
// e tecnologias assistivas também.
```

Benefícios da Semântica

Elementos semânticos vêm com comportamentos e papéis de acessibilidade embutidos, como ser focáveis via teclado e ter um papel reconhecido por leitores de tela.

Atributos ARIA (Accessible Rich Internet Applications)

Quando a semântica HTML nativa não é suficiente para descrever a função ou o estado de um elemento dinâmico, os atributos ARIA entram em cena. Eles fornecem informações adicionais para tecnologias assistivas.

Por exemplo, se você tem um painel que pode ser expandido ou recolhido:

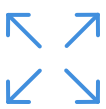
```
// HTML: <div id="painel" role="region" aria-expanded="false">...</div>
const painel = document.getElementById('painel');
const botaoToggle = document.getElementById('botaoToggle');

botaoToggle.addEventListener('click', () => {
  const isExpanded = painel.getAttribute('aria-expanded') === 'true';
  painel.setAttribute('aria-expanded', !isExpanded);

  // Adicione ou remova classes para mostrar/esconder o conteúdo visualmente
  // painel.classList.toggle('expanded');
});
```

role="region"

Informa que o div é uma seção importante da página.



aria-expanded

Informa ao leitor de tela se o painel está aberto (true) ou fechado (false).

Atualização Dinâmica

Lembre-se de atualizar esses atributos ARIA para refletir o estado atual da interface.

Isso garante que todos os usuários, independentemente de suas capacidades, possam compreender e interagir com sua aplicação de forma eficaz. A acessibilidade é um compromisso contínuo, e a manipulação consciente do DOM é um passo crucial para cumpri-lo.

Quadro Comparativo: Métodos de Seleção de Elementos

Para solidificar o entendimento sobre os diferentes métodos de seleção, vejamos um quadro comparativo que destaca suas características principais.

getElementById()	Seleciona um único elemento por seu ID único.	DOM API	document.getElementById('meuID')
querySelector()	Seleciona o *primeiro* elemento que corresponde a um seletor CSS.	DOM API	document.querySelector('.minhaClasse')
querySelectorAll()	Seleciona *todos* os elementos que correspondem a um seletor CSS, retornando uma NodeList.	DOM API	document.querySelectorAll('li.item')

Quadro Comparativo: Manipulação de Conteúdo

Entender a diferença entre textContent e innerHTML é vital para segurança e funcionalidade.

textContent	Define ou obtém o conteúdo de texto puro de um elemento. Seguro contra XSS.	DOM API	elemento.textContent = 'Novo texto'
innerHTML	Define ou obtém o conteúdo HTML de um elemento. Permite inserir tags HTML, mas exige cautela com segurança (XSS).	DOM API	elemento.innerHTML = 'Texto'

Quadro Comparativo: Manipulação de Atributos e Classes

A forma como você altera atributos e classes impacta a flexibilidade e manutenção do seu código.

setAttribute()	Define o valor de um atributo HTML.	DOM API	img.setAttribute('src', 'nova.jpg')
removeAttribute()	Remove um atributo HTML.	DOM API	img.removeAttribute('alt')
style	Acessa e modifica estilos CSS inline.	DOM API	div.style.color = 'red'
classList.add()	Adiciona uma classe CSS a um elemento.	DOM API	p.classList.add('destaque')
classList.remove()	Remove uma classe CSS de um elemento.	DOM API	p.classList.remove('destaque')
classList.toggle()	Alterna a presença de uma classe CSS.	DOM API	p.classList.toggle('ativo')

Quadro Comparativo: Criação e Remoção de Elementos

As operações de criação e remoção são a base da interatividade dinâmica.

createElement()	Cria um novo nó de elemento HTML na memória.	DOM API	document.createElement('div')
appendChild()	Anexa um elemento como o último filho de um pai.	DOM API	pai.appendChild(filho)
insertBefore()	Insere um elemento antes de um filho de referência.	DOM API	pai.insertBefore(novo, referencia)
removeChild()	Remove um filho de um elemento pai.	DOM API	pai.removeChild(filho)
remove()	Remove o próprio elemento do DOM.	DOM API	elemento.remove()

Síntese e Próximos Passos

Dominando o DOM

Chegamos ao fim da nossa jornada pela Manipulação do DOM. Vimos que o DOM é a representação viva da sua página web, uma estrutura em árvore que o JavaScript pode acessar e modificar. Aprendemos a selecionar elementos com precisão usando getElementById(), querySelector() e querySelectorAll(), cada um com sua utilidade específica. Exploramos como alterar o conteúdo de elementos com textContent (seguro para texto puro) e innerHTML (poderoso para HTML, mas exige cautela).

Selecionar

Encontrar elementos específicos na página

Otimizar

Garantir performance e acessibilidade

Manipular

Alterar conteúdo, atributos e estilos

Criar

Adicionar novos elementos dinamicamente

Remover

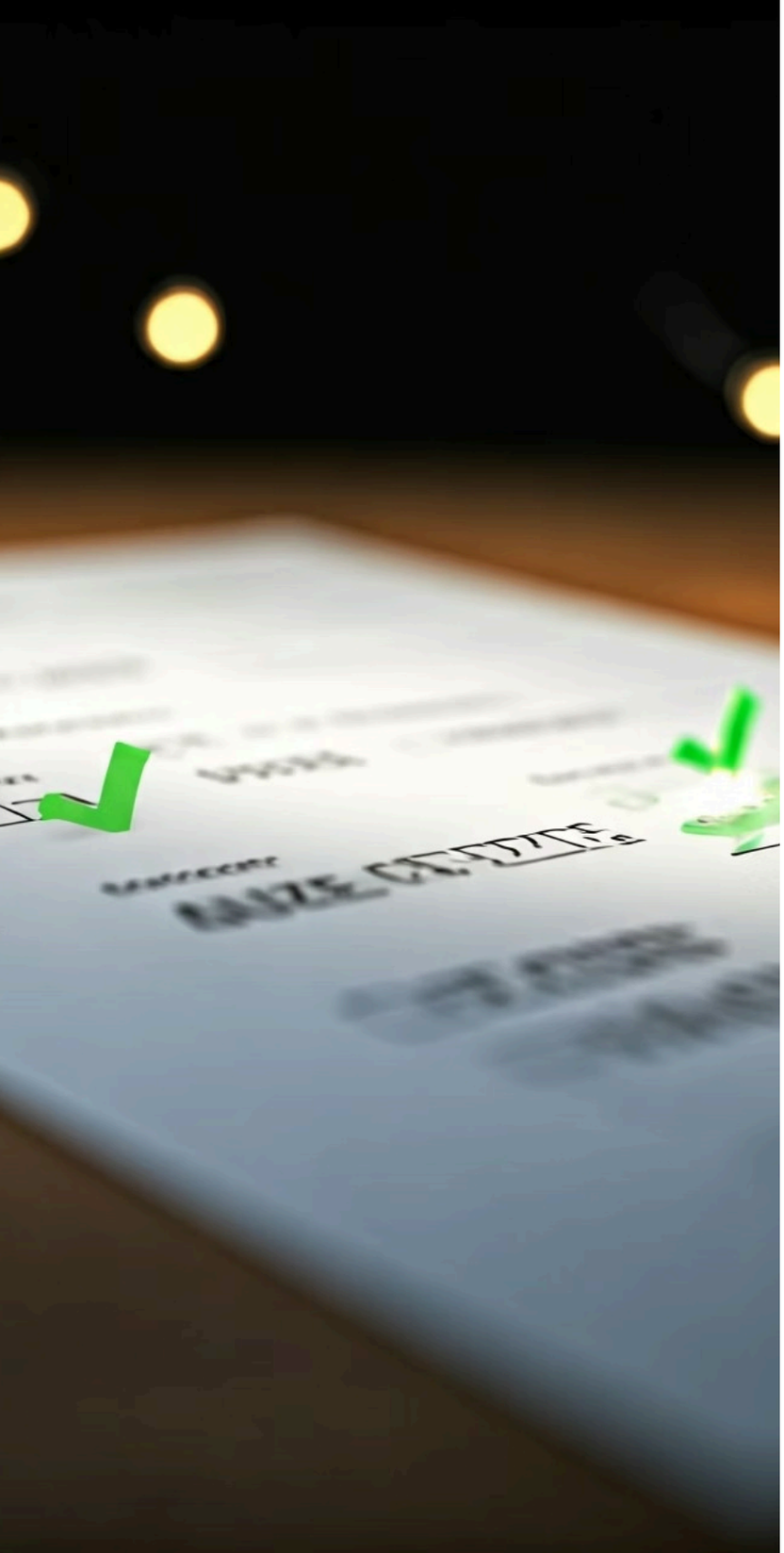
Limpar elementos desnecessários

Dominamos a arte de personalizar a aparência e o comportamento dos elementos, manipulando atributos com setAttribute() e removeAttribute(), e controlando estilos diretamente ou, de forma mais elegante e eficiente, através da adição e remoção de classes CSS com classList. Por fim, desvendamos como criar novos elementos do zero com createElement(), inseri-los na página com appendChild() e insertBefore(), e removê-los quando não são mais necessários com removeChild() ou remove().

Em prática: A manipulação do DOM é a base para qualquer interface web dinâmica. Use-a para validar formulários, criar galerias de imagens interativas, construir menus responsivos, ou adicionar e remover itens de uma lista em tempo real. Lembre-se sempre das boas práticas de performance e acessibilidade para construir aplicações robustas e inclusivas.

Autoavaliação

1. **Qual método é mais adequado para selecionar um único elemento HTML de forma eficiente, sabendo que ele possui um identificador exclusivo?** a) document.querySelectorAll('.minhaClasse')
   b) document.querySelector('div')
   c) document.getElementById('meuID')
   d) document.getElementsByTagName('p')
2. **Você precisa inserir um bloco de HTML complexo (contendo tags <b> e <i>) dentro de um div. Qual propriedade é a mais indicada para essa tarefa?** a) element.textContent
   b) element.innerText
   c) element.innerHTML
   d) element.value
3. **Para adicionar uma classe CSS chamada "ativo" a um elemento button e, posteriormente, removê-la, qual sequência de métodos da propriedade classList seria a mais apropriada?** a) button.classList.set('ativo'); button.classList.unset('ativo');
   b) button.classList.add('ativo'); button.classList.remove('ativo');
   c) button.classList.toggle('ativo'); button.classList.toggle('ativo');
   d) button.classList.change('ativo', true); button.classList.change('ativo', false);
4. **Qual das seguintes práticas é considerada uma boa prática para otimizar a performance ao adicionar múltiplos elementos ao DOM?** a) Adicionar cada elemento individualmente ao DOM dentro de um loop.
   b) Utilizar document.write() para inserir o HTML de uma vez.
   c) Criar os elementos em um DocumentFragment e anexá-lo ao DOM uma única vez.
   d) Usar innerHTML para construir uma string HTML gigante e atribuí-la ao elemento pai.
5. Explique a importância da semântica HTML e dos atributos ARIA ao manipular o DOM para garantir a acessibilidade de uma aplicação web.



Gabarito

Questão 1

Resposta: c)

getElementById() é o método mais eficiente para selecionar um elemento por ID único.

Questão 2

Resposta: c)

innerHTML permite inserir HTML complexo com tags formatadas.

Questão 3

Resposta: b)

classList.add() e classList.remove() são os métodos corretos para gerenciar classes.

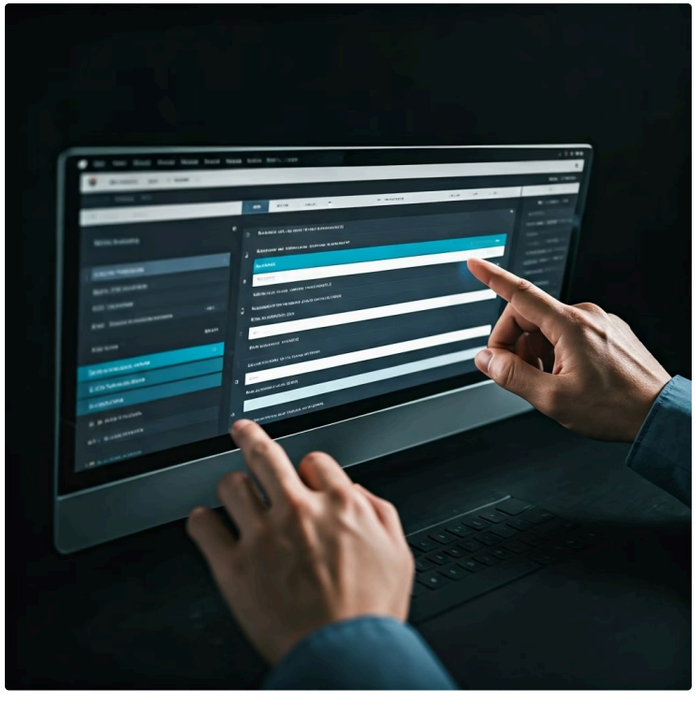
Questão 4

Resposta: c)

DocumentFragment minimiza reflows ao agrupar operações antes de anexar ao DOM.

Próxima Aula

Aula 16 – Eventos e Interatividade



Na próxima aula, vamos aprofundar a interação com o usuário, aprendendo a responder a cliques, digitações e outros eventos, transformando a manipulação do DOM em uma experiência verdadeiramente dinâmica.

Recursos Adicionais:

- **MDN Web Docs - Introdução ao DOM:** Para uma referência completa e aprofundada.
- **Artigos sobre Core Web Vitals:** Para entender o impacto da performance do DOM na experiência do usuário.
- **Guia de Acessibilidade Web (W3C):** Para aprofundar os conceitos de A11Y e ARIA.

NOTA IMPORTANTE: As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.