Eventos em JavaScript: `addEventListener`

1. INTRODUÇÃO AOS EVENTOS NO DOM

No contexto de uma página da web, "eventos" são as ações ou acontecimentos que ocorrem na página, geralmente iniciados pelo usuário. Eles são a espinha dorsal da interatividade, permitindo que nossas páginas reajam às interações do usuário em tempo real. Sem eventos, uma página seria estática; com eles, ela se torna uma aplicação dinâmica e responsiva. Qualquer elemento da página pode ser monitorado para uma vasta gama de eventos. Alguns dos mais comuns incluem:

- **Eventos de Clique:** `onclick` (clique simples), `ondblclick` (clique duplo).
- **Eventos de Mouse:** `onmouseenter` (mouse entra no elemento), `onmouseleave` (mouse sai do elemento), `onmousemove` (mouse se move sobre o elemento).
- **Eventos de Foco:** `onfocus` (elemento recebe foco), `onblur` (elemento perde o foco).
- **Eventos de Alteração:** `onchange` (o valor de um elemento de formulário muda), `oninput` (o valor de um elemento é alterado).
- **Eventos de Teclado:** `onkeydown` (tecla é pressionada), `onkeyup` (tecla é solta).
- **Outros Eventos:** `ondrag` (um elemento é arrastado), `onplay` (um vídeo ou áudio começa a tocar).

O conceito central é que podemos "escutar" esses eventos e, quando um deles ocorre, podemos "disparar uma função" em resposta. Isso conecta a ação do usuário (como um clique) a uma funcionalidade específica que programamos em JavaScript, como exibir uma mensagem, enviar dados de um formulário ou alterar o estilo de um elemento. A seguir, exploraremos as diferentes maneiras de se trabalhar com eventos, desde uma abordagem mais antiga diretamente no HTML até o método moderno e profissional em JavaScript.

1.1. Manipulando Eventos Diretamente no HTML

Uma das formas de manipular eventos é a abordagem "*inline*", que consiste em adicionar o código de resposta ao evento diretamente dentro da tag HTML do elemento. Embora essa técnica funcione, ela **não é considerada uma boa prática** no desenvolvimento web moderno. O principal motivo é que ela mistura a estrutura (HTML) com o comportamento (JavaScript), indo contra o princípio fundamental da "separação de responsabilidades". Isso significa que para alterar o comportamento de um clique, você

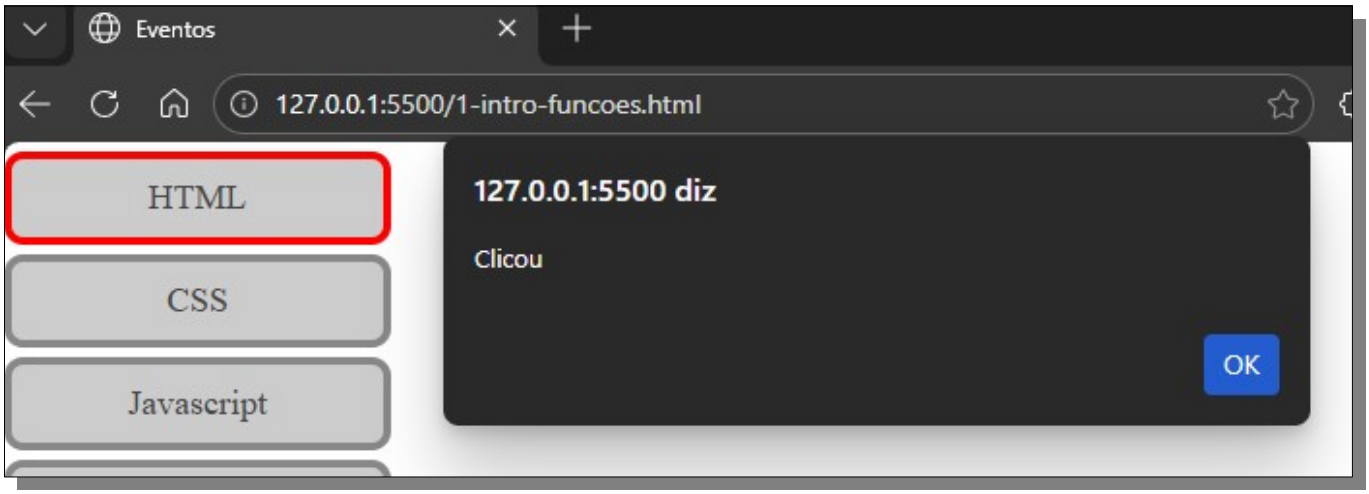
precisaria editar o arquivo HTML, em vez de manter toda a lógica de interação centralizada em seu arquivo JavaScript, o que torna a manutenção do projeto mais difícil e propensa a erros.
O atributo de evento `onclick` é o exemplo mais comum dessa abordagem.

1.1.1. Exemplo 1: Código JavaScript Direto no Atributo

É possível escrever uma instrução JavaScript simples diretamente dentro do valor do atributo `onclick`.

```
<div id="C1" class="curso c1" onclick="alert('Clicou!')">HTML</div>
```

Neste caso, a função nativa `alert()` do JavaScript é chamada diretamente dentro das aspas do atributo `onclick`. Quando o usuário clica nesta `div`, o navegador executa esse código, exibindo a caixa de alerta.



1.1.2. Exemplo 2: Chamando uma Função Definida

Uma abordagem um pouco mais organizada é definir uma função em um script JavaScript e apenas chamar essa função no atributo `onclick`. Isso melhora a reutilização e a legibilidade.

Código HTML:

```
<div id="C1" class="curso c1" onclick="msg()">HTML</div>
```

Código JavaScript: A função `msg` pode ser declarada de várias maneiras em JavaScript. Todas as opções abaixo teriam o mesmo resultado:

Opção A: Função Nomeada

```
function msg() {
    alert("Clicou!");
}
```

Opção B: Função Anônima

```
const msg = function() {
    alert("Clicou!");
}
```

Opção C: Arrow Function

```
const msg = () => {
    alert("Clicou!");
}
```

Aqui, o atributo `onclick` contém apenas a chamada para a função `msg()`. Isso é preferível a escrever a lógica diretamente no HTML, pois a função `msg` pode ser reutilizada por outros elementos e sua lógica pode ser alterada em um único lugar (no arquivo `.js`), sem a necessidade de modificar o HTML. Embora funcional, essa técnica ainda mantém um vínculo direto entre o HTML e o JavaScript. A seguir, veremos como o método `addEventListener` oferece uma solução superior e mais profissional para gerenciar eventos.

1.2. A Abordagem Moderna: `addEventListener`

O método `addEventListener` é a forma recomendada e mais poderosa de gerenciar eventos em JavaScript. Ele permite um controle muito mais granular sobre os eventos e mantém o código JavaScript completamente separado da estrutura HTML. A partir deste ponto, considere `addEventListener` como a única forma profissional de se trabalhar com eventos em seus projetos. Embora os métodos *inline* existam, eles são considerados legados e devem ser evitados em código moderno. O processo para usar `addEventListener` envolve dois passos simples:

1.2.1. Passo 1: Selecionar o Elemento

Primeiro, precisamos obter uma referência ao elemento do *Document Object Model* (DOM) ao qual queremos anexar o evento. Podemos fazer isso usando métodos como `getElementById` ou, de forma mais versátil, com `querySelector`.

```
// Selecionando pelo ID
const C1 = document.getElementById('C1');

// Ou usando querySelector com a sintaxe de seletor CSS
const C1 = document.querySelector('#C1');
```

1.2.2. Passo 2: Adicionar o "Escutador" de Eventos

Com a referência ao elemento em mãos, usamos o método `addEventListener` nesse elemento. A sintaxe básica requer dois argumentos principais:

1. **O Evento:** O nome do evento que queremos "escutar", passado como uma string (ex: `'click'`, `'mouseover'`).
2. **A Função de *Callback***: A função que será executada quando o evento ocorrer.
   Existem diferentes formas de fornecer essa função de *callback.*

- **Usando uma função nomeada existente:** Se já temos uma função definida, podemos passá-la diretamente como segundo argumento.
- **Usando uma *Arrow Function* anônima:** É muito comum definir a lógica diretamente dentro do `addEventListener` usando uma função anônima. Isso é útil para ações que são específicas daquele evento.

Como veremos a seguir, a função de *callback* não serve apenas para executar uma ação. Ela também recebe automaticamente informações valiosas sobre o evento que acabou de ocorrer.

1.3. O Objeto de Evento (`event`) e a Propriedade `target`

Quando um evento é disparado e a função de *callback* é executada, o navegador passa automaticamente um argumento para essa função: o **objeto de evento**. Este objeto é uma fonte rica de informações contextuais sobre a interação do usuário, como as coordenadas do mouse, a tecla pressionada e, mais importante, qual elemento originou o evento. Para inspecionar esse objeto, podemos recebê-lo

como um parâmetro (comumente chamado de `evt` ou `event`) em nossa função de *callback* e exibi-lo no console.

```
const C1 = document.querySelector('#C1');

C1.addEventListener('click', (evt) => {
    console.log(evt);
});
```

Ao clicar no elemento, você verá um objeto `PointerEvent` no console, cheio de propriedades. Dentre todas elas, a mais utilizada e importante é `evt.target`. Ela contém uma referência direta ao elemento do DOM que disparou o evento.

```
const C1 = document.querySelector('#C1');

C1.addEventListener('click', (evt) => {
    console.log(evt.target); // Exibirá a tag <div id="C1" ...> no console
});
```

Podemos usar essa referência para manipular diretamente o elemento que foi clicado. No exemplo abaixo, adicionamos uma classe CSS chamada `destaque` ao elemento que originou o evento, alterando sua aparência visual.

```
const C1 = document.querySelector('#C1');

C1.addEventListener('click', (evt) => {
    // Armazenamos a referência ao elemento clicado em uma constante
    const elemento = evt.target;

    // Adicionamos a classe 'destaque' a este elemento
    elemento.classList.add('destaque');
});
```

Quando o usuário clica na `div`, a classe `destaque` é adicionada a ela, e seu estilo é alterado conforme definido no CSS. Este conceito se torna ainda mais poderoso quando o aplicamos a múltiplos elementos de uma só vez.

1.4. Adicionando Eventos em Múltiplos Elementos

Um desafio comum é aplicar o mesmo comportamento de evento a um grupo de elementos semelhantes, como itens de uma lista ou, no nosso caso, vários *cards* de cursos. A combinação de `addEventListener`, `querySelectorAll` e `evt.target` oferece uma solução elegante para isso.

1.4.1. Passo 1: Selecionar Todos os Elementos

Primeiro, usamos `document.querySelectorAll()` para obter uma coleção de todos os elementos que compartilham um seletor CSS comum.

```
const todosCursos = [...document.querySelectorAll('.curso')];
```

O método `querySelectorAll` não retorna um `Array` padrão, mas sim uma `NodeList`. Embora parecida, uma `NodeList` não possui todos os métodos de um `Array`, como o `.map()`. Usamos o *spread syntax* (`...`) dentro de colchetes (`[]`) para converter essa `NodeList` em um `Array` de verdade, nos dando acesso a todo o poder dos métodos de array.

1.4.2. Passo 2: Iterar e Adicionar o Listener

Com um `Array` de elementos em mãos, percorremos cada um deles usando `.map()` e adicionamos o mesmo "escutador" de eventos a cada um.

```
todosCursos.map((el) => {
    el.addEventListener('click', (evt) => {
        const elementoClicado = evt.target;
        elementoClicado.classList.add('destaque');
    });
});
```

1.4.3. Análise do Poder da Técnica

A beleza dessa abordagem está no fato de que, embora o mesmo *listener* seja adicionado a todos os elementos, o uso de `evt.target` dentro da função de *callback* nos permite identificar e interagir **especificamente com o elemento que foi clicado**. Não importa se o usuário clicou no curso de HTML, JavaScript ou C++; `evt.target` sempre se referirá à `div` correta.

Com essa referência, podemos extrair qualquer informação do elemento.

- **Exemplo: Obter o `id` do elemento clicado**
    - *Resultado ao clicar no primeiro curso:* `"C1 foi clicado"`
    - *Resultado ao clicar no curso de C++:* `"C7 foi clicado"`
- **Exemplo: Obter o conteúdo HTML do elemento clicado**
    - *Resultado ao clicar no primeiro curso:* `"HTML foi clicado"`
    - *Resultado ao clicar no curso de C++:* `"C++ foi clicado"`
    - *Resultado ao clicar no curso de Raspberry:* `"Raspberry foi clicado"`

Essa abordagem é extremamente eficiente e escalável, permitindo construir interfaces complexas e interativas com um código limpo e de fácil manutenção.

2. CONCLUSÃO E PRÓXIMOS PASSOS

Nesta apostila, exploramos o conceito fundamental de eventos em JavaScript, contrastando a antiga abordagem "inline" no HTML com o método moderno e superior `addEventListener`. Vimos como selecionar elementos do DOM, anexar "escutadores" de eventos a eles e, mais importante, como utilizar o objeto de evento e a propriedade `evt.target` para identificar e manipular precisamente o elemento com o qual o usuário interagiu. A combinação de `querySelectorAll` com `addEventListener` provou ser uma técnica poderosa para aplicar comportamentos a múltiplos elementos de forma escalável.

Não se intimide se os conceitos parecerem complexos; a maestria em manipulação de eventos vem com a prática contínua, que é a chave para se tornar um desenvolvedor proficiente. Continue explorando e aplicando esses conhecimentos, pois o estudo de eventos é um caminho contínuo e fundamental para a sua jornada em JavaScript.