Propagação de Eventos em JavaScript com `stopPropagation`

1. O QUE É A PROPAGAÇÃO DE EVENTOS E POR QUE CONTROLÁ-LA?

No ecossistema do DOM (*Document Object Model*) em JavaScript, a propagação de eventos, mais conhecida como **event bubbling** (ou "borbulhamento"), é um comportamento fundamental e, por vezes, desafiador. Por padrão, quando um evento é acionado em um elemento aninhado, ele não termina ali; ele "borbulha" para cima na hierarquia de elementos, acionando sequencialmente os *event listeners* de cada um de seus elementos pais. Esse fluxo, embora poderoso, pode levar a comportamentos não intencionais, onde múltiplos *handlers* são executados por um único clique. Controlar esse fluxo é uma habilidade estratégica essencial para criar interfaces de usuário previsíveis, robustas e livres de conflitos.

A principal ferramenta para gerenciar esse desafio é o método `event.stopPropagation()`. Este guia prático demonstrará como a propagação de eventos funciona e como utilizar `stopPropagation()` para obter controle total sobre a interatividade dos seus componentes. Então, vamos ver o problema na prática para entender por que essa ferramenta é tão importante.

1.1. O Problema na Prática: O Efeito "Bolha"

Para entender como resolver a propagação indesejada, primeiro precisamos vê-la acontecer. A melhor metáfora para o *event bubbling* é a de uma "bolha". Quando você aciona um evento em um elemento filho, é como se uma bolha fosse criada e começasse a subir, passando por todos os elementos pais que a contêm até chegar ao topo do DOM. Se algum desses elementos pais tiver um *listener* para aquele mesmo tipo de evento, ele será acionado. Entender este comportamento é o primeiro passo para poder controlá-lo.

1.1.1. Cenário de Exemplo

Para nossa demonstração, vamos usar a mesma estrutura HTML e CSS das aulas anteriores: uma `div` contêiner principal com o ID `caixa1`, que por sua vez envolve várias `div`s filhas, cada uma com a classe `.curso` (como `C1`, `C2`, etc.).

1.1.2. Adicionando um Evento ao Contêiner Pai

Para vermos o problema em ação, vamos começar com um passo simples: adicionar um `event listener` de clique apenas à `div` pai, `caixa1`. O objetivo é registrar uma mensagem no console sempre que este contêiner for clicado.

```
const caixa1 = document.querySelector("#caixa1");

caixa1.addEventListener("click", () => {
    console.log("clicou");
});
```

Analisando o código, a intenção é clara: qualquer clique direto na área de `caixa1` deve resultar na mensagem "clicou" sendo exibida no console.

1.1.3. Demonstrando a Propagação Indesejada

Ao testar, o comportamento inicial parece correto: clicar diretamente na área de fundo da `div caixa1` exibe "clicou" no console. No entanto, o comportamento inesperado ocorre a seguir: ao clicar em **qualquer uma das `div`s filhas** dentro de `caixa1`, a mensagem "clicou" **também** aparece no console.

Isso acontece porque o **único evento** de clique gerado no filho não termina ali. Ele se propaga (ou "borbulha") para o elemento pai, que então executa seu próprio *listener* como se o clique tivesse ocorrido diretamente nele. A "bolha" do evento de clique no filho subiu e "estourou" no *listener* do pai. Essa propagação automática é a raiz do problema. Precisamos de uma forma de interromper esse fluxo quando não for desejado.

1.2. A Solução: Interrompendo a Bolha com `event.stopPropagation()`

O método `event.stopPropagation()` é a solução precisa para o problema da propagação. É uma solução cirúrgica porque nos permite intervir no ponto exato da hierarquia do DOM, parando o fluxo do evento sem afetar outros *listeners* no mesmo elemento ou em seus filhos. Para usá-lo, primeiro precisamos entender o objeto que nos dá acesso a ele.

1.2.1. Entendendo o Objeto `Event`

Quando um evento é disparado, a função de *callback* fornecida ao `addEventListener` recebe automaticamente um objeto como seu primeiro argumento. Este objeto, comumente nomeado como `evt` ou `event` por convenção, é uma mina de ouro de informações sobre o evento que acabou de ocorrer. Dentro deste objeto, propriedades como `evt.target` são extremamente úteis, pois nos dizem exatamente qual elemento filho originou o evento, mesmo que o *listener* esteja no pai. Podemos inspecionar este objeto facilmente com o seguinte código:

```
caixa1.addEventListener("click", (evt) => {
    console.log(evt);
});
```

Ao clicar em `caixa1`, o console exibirá um objeto `PointerEvent` com dezenas de propriedades. O método `stopPropagation()`, que é o nosso foco, é parte integrante deste objeto de evento.

1.2.2. Implementando a Solução em um Único Elemento

Agora, vamos aplicar a solução. Adicionaremos um novo `event listener` a um elemento filho específico (`c1`) com o único propósito de parar a propagação antes que ela atinja o pai.

```
const c1 = document.querySelector("#c1");

c1.addEventListener("click", (evt) => {
    evt.stopPropagation();
});
```

Quando a `div c1` é clicada, seu `event listener` é o primeiro a ser acionado na fase de borbulhamento. A chamada `evt.stopPropagation()` dentro deste *listener* atua como uma barreira imediata. Ela instrui o navegador a encerrar a propagação do evento naquele exato ponto, impedindo que ele continue sua subida na hierarquia do DOM e alcance o *listener* da `div caixa1`.

O resultado é imediato: clicar em `c1` agora não produz mais a mensagem "clicou" no console. No entanto, clicar nas outras `div`s filhas ainda aciona o *listener* do pai, pois elas não têm a instrução para

parar a propagação. Isso nos leva à próxima questão: como escalar essa solução para todos os elementos de uma vez?

1.3. Padrão Avançado: Aplicando `stopPropagation` a Múltiplos Elementos

Em aplicações reais, raramente aplicamos lógica a um único elemento. É muito mais comum precisar do mesmo comportamento em toda uma coleção de elementos. Aplicar soluções de forma eficiente e escalável é uma marca de um código robusto. Esta seção demonstra o padrão ideal para aplicar `stopPropagation` a múltiplos elementos em JavaScript moderno.

1.3.1. Selecionando uma Coleção de Elementos

Primeiro, vamos selecionar todos os elementos alvo de uma só vez. Usando `document.querySelectorAll`, podemos obter uma `NodeList` de todos os elementos que compartilham a classe `.curso`. Em seguida, usamos o operador *spread* (`...`) para converter essa `NodeList` em um *Array*, o que nos dá acesso a métodos de iteração poderosos.

```
const cursos = [...document.querySelectorAll(".curso")];
```

1.3.2. Adicionando o Listener em Massa

Com um *array* de elementos em mãos, podemos iterar sobre ele e adicionar o nosso `event listener` com `stopPropagation` a cada um dos elementos de forma concisa.

```
cursos.map((el) => {
    el.addEventListener("click", (evt) => {
        evt.stopPropagation();
    })
});
```

Este código utiliza um método de iteração para percorrer cada elemento (`el`) no *array* `cursos`. Para cada um, ele adiciona um `event listener` de clique. A função de *callback* para cada *listener*

executa a mesma lógica: chama `evt.stopPropagation()`, garantindo que o clique em qualquer uma das `div`s de curso interrompa a propagação do evento.

> **Nota do instrutor:** Embora `.map()` funcione, o método `.forEach()` é semanticamente mais correto aqui, pois nosso objetivo é executar uma ação (adicionar um *listener*) em cada elemento, e não criar um novo *array* a partir dos resultados. No entanto, ambos resolvem o problema de forma eficaz.

1.3.3. Verificando o Resultado Final

Com a implementação final, o comportamento da página agora está sob nosso controle total:

- Clicar diretamente na área do contêiner `caixa1` (fora dos elementos filhos) ainda registra "clicou" no console, como esperado.
- Clicar em **qualquer uma** das `div`s de curso (`.curso`) não aciona mais o evento do contêiner pai. A propagação é interrompida no momento em que o clique é detectado no filho.

O problema da propagação indesejada foi completamente resolvido de forma limpa e eficiente para todos os elementos relevantes.

2. CONCLUSÃO E RESUMO ESTRATÉGICO

O método `event.stopPropagation()` é uma ferramenta essencial no arsenal de um desenvolvedor JavaScript. Ele oferece a precisão necessária para construir componentes interativos complexos, garantindo que as ações do usuário tenham os efeitos exatos pretendidos, sem acionar comportamentos indesejados em elementos pais. Dominar seu uso é um passo fundamental para a criação de interfaces de usuário mais limpas e previsíveis. Para revisar, aqui estão os aprendizados-chave deste guia:

- **Propagação de Eventos (Bubbling):** Por padrão, eventos em elementos DOM filhos "borbulham" para cima, acionando *listeners* em seus elementos pais.
- **O Objeto `Event`:** É o primeiro parâmetro passado para um callback de `event listener`. Ele contém dados e métodos cruciais sobre o evento, como o `stopPropagation()`.
- **O Método `stopPropagation()`:** Quando chamado dentro de um `event listener`, impede que o evento continue a se propagar **para cima** na hierarquia do DOM, isolando o evento no elemento em que foi capturado.
- **Aplicação em Escala:** Para aplicar `stopPropagation` a múltiplos elementos, o padrão moderno é usar `querySelectorAll` para selecionar a coleção e, em seguida, usar métodos de iteração de array (como `.forEach()`) para adicionar o *listener* a cada elemento de forma limpa e eficiente.