entrada.addEventListener('input', () => {

let texto = entrada.value;

demo.textContent = "Texto digitado: " + texto;

});

**1.**entrada

* **O que é?** entrada é uma variável que representa um elemento HTML da sua página. Provavelmente, é um campo de formulário, como <input type="text"> ou <textarea>.
* **Como funciona?** Em algum lugar antes deste código, esse elemento foi selecionado do DOM (Document Object Model - a estrutura da sua página) e armazenado nesta variável. Por exemplo:

const entrada = document.querySelector('#meu-input'); // Seleciona o elemento com id="meu-input"

**2. .**addEventListener(...)

* **O que é?** É um método (uma função) que "anexa" um "ouvinte de evento" a um elemento HTML.
* **O que faz?** Ele diz ao navegador: "Ei, preste atenção neste elemento (entrada). Quando um evento específico acontecer com ele, execute uma função que eu vou te passar".

**3.**'input'

* **O que é?** É o primeiro argumento do addEventListener. Ele especifica **qual evento** o navegador deve escutar.
* **O que faz?** O evento 'input' é disparado toda vez que o valor de um elemento <input>, <select>, ou <textarea> é alterado. Isso inclui digitar, colar, apagar ou qualquer outra ação que mude o conteúdo do campo. É a melhor escolha para capturar a digitação em tempo real.

**4.**() => { ... }

* **O que é?** Esta é uma **função de callback** (também conhecida como *Arrow Function*, uma sintaxe moderna do JavaScript). É o segundo argumento do addEventListener.
* **O que faz?** É o bloco de código que será **executado** sempre que o evento 'input' ocorrer no elemento entrada. Tudo o que está dentro das chaves {} acontece quando o usuário digita algo.

**5.**let texto = entrada.value;

* **O que é?** Esta linha está dentro da função de callback.
* **let texto**: Declara uma nova variável chamada texto.
* **entrada.value**: A propriedade .value de um elemento de entrada contém o **texto atual** que está dentro dele.
* **O que faz?** A linha inteira pega o texto que o usuário acabou de digitar no campo entrada e o armazena na variável texto.

**6.**demo.textContent = "Texto digitado: " + texto;

* **O que é?** Esta é a última linha da função, e é ela que atualiza a página.
* **demo**: Assim como entrada, demo é uma variável que representa outro elemento HTML na página, provavelmente um parágrafo <p>, uma <span> ou uma <div>. Por exemplo:

const demo = document.querySelector('#elemento-demo');

*// Seleciona o elemento com id="elemento-demo"*

* **.textContent**: É uma propriedade que define ou obtém o conteúdo de texto de um elemento. Ao atribuir um valor a ela, você substitui qualquer texto que estivesse lá antes.
* **"Texto digitado: " + texto**: Aqui, você está criando uma nova string (texto) que junta a frase "Texto digitado: " com o conteúdo da variável texto (que é o que o usuário digitou).
* **O que faz?** A linha inteira atualiza o conteúdo do elemento demo para mostrar a frase "Texto digitado: " seguida pelo que o usuário digitou no campo entrada.

**Resumo Visual**

Imagine que sua página tem estes dois elementos:

*<!-- O elemento "entrada" -->*

<input type="text" id="meu-input" placeholder="Digite algo aqui...">

*<!-- O elemento "demo" -->*

<p id="elemento-demo"></p>

O seu código JavaScript conecta os dois:

1. O usuário digita a letra "A" no campo <input>.
2. O evento 'input' é disparado.
3. A função () => { ... } é executada.
4. entrada.value agora é "A". A variável texto recebe "A".
5. O textContent do parágrafo <p> é atualizado para "Texto digitado: A".
6. O usuário digita a letra "B".
7. O evento 'input' é disparado novamente.
8. entrada.value agora é "AB". A variável texto recebe "AB".
9. O textContent do parágrafo <p> é atualizado para "Texto digitado: AB".

E assim por diante, criando um efeito de feedback instantâneo para o usuário.

**1. Eventos de Mouse**

São disparados pela interação do usuário com um mouse, trackpad ou tela sensível ao toque (que simula eventos de mouse).

| **Evento** | **Descrição** |
| --- | --- |
| click | Disparado quando o botão principal do mouse é pressionado e solto sobre um elemento. | |
| dblclick | Disparado com um clique duplo no elemento. | |
| mousedown | Disparado quando o botão do mouse é **pressionado** sobre o elemento. | |
| mouseup | Disparado quando o botão do mouse é **solto** sobre o elemento. | |
| mousemove | Disparado continuamente enquanto o ponteiro do mouse se move sobre o elemento. | |
| mouseover / mouseenter | Disparado quando o ponteiro do mouse **entra** na área do elemento. | |
| mouseout / mouseleave | Disparado quando o ponteiro do mouse **sai** da área do elemento. | |
| contextmenu | Disparado quando o botão direito do mouse é clicado (para abrir o menu de contexto). | |

**Diferença:** mouseover e mouseout "borbulham" (propagam) para elementos pais, enquanto mouseenter e mouseleave não, sendo geralmente mais previsíveis.

**2. Eventos de Teclado**

Relacionados ao uso do teclado.

| **Evento** | **Descrição** |
| --- | --- |
| keydown | Disparado quando uma tecla é **pressionada**. |
| keyup | Disparado quando uma tecla é **solta**. |

**3. Eventos de Formulário**

Essenciais para interagir com campos de formulário como <input>, <select>, <textarea> e <form>.

| **Evento** | **Descrição** |
| --- | --- |
| input | Disparado **imediatamente** quando o valor de um <input>, <select> ou <textarea> é alterado. Ideal para feedback em tempo real. |
| change | Disparado quando o valor de um elemento é alterado e o elemento **perde o foco**. Para caixas de texto, isso acontece quando você clica fora; para selects, acontece assim que uma nova opção é escolhida. |
| submit | Disparado no elemento <form> quando ele é enviado (seja por um clique em <button type="submit"> ou pressionando Enter). |
| focus | Disparado quando um elemento recebe foco (por exemplo, ao clicar em um campo de texto). |
| blur | Disparado quando um elemento perde o foco. |
| reset | Disparado no elemento <form> quando o formulário é resetado (por um <button type="reset">). |

**4. Eventos de Janela (Window) e Documento**

Relacionados ao estado geral da janela do navegador ou do documento HTML. Geralmente são "escutados" no objeto window ou document.

| **Evento** | **Descrição** |
| --- | --- |
| load | Disparado no objeto window quando a página inteira (incluindo imagens, scripts, CSS) terminou de carregar. |
| DOMContentLoaded | Disparado no document assim que o HTML da página foi completamente carregado e analisado, **sem esperar** por CSS, imagens e subframes. É o evento preferido para iniciar scripts que manipulam o DOM. |
| resize | Disparado no window sempre que a janela do navegador é redimensionada. |
| scroll | Disparado em um elemento (ou no window) quando seu conteúdo é rolado (scroll). |
| beforeunload | Disparado no window pouco antes do usuário sair da página (fechando a aba, navegando para outro site, etc.). Usado para exibir avisos como "Você tem certeza que quer sair?". |

**5. Eventos de Foco**

Já mencionados na categoria de formulários, mas aplicáveis a qualquer elemento que possa receber foco (como links e botões).

| **Evento** | **Descrição** |
| --- | --- |
| focus | O elemento recebeu foco. |
| blur | O elemento perdeu o foco. |
| focusin | Similar ao focus, mas "borbulha" (propaga). |
| focusout | Similar ao blur, mas "borbulha" (propaga). |

**6. Eventos de Mídia**

Para controlar elementos de áudio e vídeo (<audio>, <video>).

| **Evento** | **Descrição** |
| --- | --- |
| play | Disparado quando a mídia começa a tocar. |
| pause | Disparado quando a mídia é pausada. |
| ended | Disparado quando a mídia termina de tocar. |
| timeupdate | Disparado continuamente enquanto o tempo de reprodução da mídia muda. |
| volumechange | Disparado quando o volume é alterado. |

**Diferença Crucial: load vs. DOMContentLoaded**

Essa é uma das dúvidas mais comuns e importantes. A escolha entre um e outro pode impactar significativamente a performance e a responsividade da sua página.

A diferença está no **momento** em que são disparados e **o que eles esperam** para acontecer.

**DOMContentLoaded**

* **O que é?** É um evento disparado no objeto document.
* **Quando dispara?** Assim que o navegador termina de carregar e analisar **todo o código HTML** da página e constrói a árvore DOM (Document Object Model).
* **O que ele espera?** Apenas o HTML.
* **O que ele NÃO espera?** Ele **não espera** pelo carregamento de recursos externos como:
  + Folhas de estilo (CSS)
  + Imagens (<img>)
  + Iframes
  + Scripts assíncronos (<script async>)
* **Quando usar?** Na **maioria das vezes**. Se o seu script JavaScript precisa apenas interagir com os elementos da página (adicionar event listeners, mudar texto, esconder/mostrar divs), DOMContentLoaded é o ideal. Ele permite que seu código execute o mais rápido possível, sem esperar por imagens pesadas ou outros recursos, tornando a página interativa mais rapidamente.

**Exemplo de uso:**

document.addEventListener('DOMContentLoaded', () => {

// O HTML está pronto. Pode manipular os elementos com segurança.

const botao = document.getElementById('meu-botao');

if (botao) {

botao.addEventListener('click', () => {

console.log('Botão clicado! O DOM estava pronto.');

});

}

});

**load**

* **O que é?** É um evento disparado no objeto window.
* **Quando dispara?** Apenas depois que **TUDO** na página foi completamente carregado.
* **O que ele espera?**
  + O HTML (e a árvore DOM construída).
  + Todas as folhas de estilo (CSS).
  + Todos os scripts.
  + Todas as imagens.
  + Todos os iframes e outros recursos externos.
* **Quando usar?** Em situações específicas onde seu código depende das propriedades de recursos externos. Por exemplo:
  + Se você precisa saber as dimensões (largura/altura) de uma imagem para fazer um cálculo. Você só pode fazer isso depois que a imagem foi de fato carregada.
  + Para iniciar uma animação complexa que só deve começar quando a página estiver 100% pronta visualmente.

**Exemplo de uso:**

window.addEventListener('load', () => {

// A página inteira, incluindo imagens e CSS, está pronta.

const imagem = document.getElementById('minha-imagem');

if (imagem) {

// Agora podemos medir a imagem com segurança

console.log(`A imagem carregou e sua altura é: ${imagem.offsetHeight}px`);

}

});

**Tabela Comparativa**

| **Característica** | **DOMContentLoaded** | **load** |
| --- | --- | --- |
| **Objeto alvo** | document | window |
| **Gatilho** | HTML carregado e DOM pronto. | **Toda a página** e seus recursos (CSS, imagens, etc.) carregados. |
| **Velocidade** | **Mais rápido.** Dispara antes. | **Mais lento.** É o último evento do ciclo de carregamento. |
| **Uso principal** | Manipulação do DOM (adicionar eventos, alterar elementos). | Código que depende de recursos externos (ex: dimensões de imagem). |
| **Performance** | **Melhor** para a percepção do usuário (página fica interativa antes). | Pode fazer a página parecer lenta se um script importante esperar por ele. |

**Conclusão:** Para 95% dos casos, **use DOMContentLoaded**. Ele garante que a interatividade da sua página comece o mais cedo possível, melhorando a experiência do usuário. Reserve o evento load apenas para os casos em que você realmente precisa que todos os recursos secundários estejam disponíveis.

a propagação de eventos é um passo fundamental para se tornar um desenvolvedor JavaScript mais proficiente. Esse conceito resolve muitos comportamentos "estranhos" que iniciantes encontram e abre portas para técnicas mais avançadas.

**O Que é a Propagação de Eventos?**

Quando um evento ocorre em um elemento aninhado dentro de outros (por exemplo, um clique em um <span> dentro de um <p>, que está dentro de uma <div>), o navegador não dispara o evento *apenas* nesse elemento. Em vez disso, o evento viaja através de todos os seus ancestrais na árvore DOM.

Essa viagem do evento é chamada de **propagação de eventos** e acontece em duas fases principais:

1. **Fase de Captura (Capturing Phase):** O evento "desce" da raiz da árvore DOM (o window) até o elemento alvo.
2. **Fase de Borbulhamento (Bubbling Phase):** Depois de chegar ao alvo, o evento "sobe" de volta do elemento alvo até a raiz (window).

Vamos visualizar com um exemplo simples:

<div id="avo">

<div id="pai">

<span id="filho">Clique Aqui</span>

</div>

</div>

Se você clicar no <span> (filho), a viagem do evento click será:

1. **Captura:** window -> document -> <html> -> <body> -> div#avo -> div#pai -> span#filho (o alvo).
2. **Borbulhamento:** span#filho -> div#pai -> div#avo -> <body> -> <html> -> document -> window.

Por padrão, **todos os event listeners modernos operam na fase de borbulhamento (Bubbling)**.

**Fase 1: Captura (Capturing)**

* **Direção:** De fora para dentro (do ancestral mais distante para o alvo).
* **Comportamento:** É como se o navegador estivesse perguntando: "Alguém no caminho para o alvo quer lidar com este evento *antes* que ele chegue lá?".
* **Uso:** É raramente usado no dia a dia. Sua principal utilidade é para lógicas muito específicas, como monitorar eventos em um nível superior antes que os elementos internos possam cancelá-los.
* **Como usar:** Para fazer um event listener operar na fase de captura, você precisa passar true como o terceiro argumento do método addEventListener.

*// Este listener será acionado na fase de CAPTURA*

elemento.addEventListener('click', minhaFuncao, true);

**Fase 2: Borbulhamento (Bubbling)**

* **Direção:** De dentro para fora (do alvo para o ancestral mais distante).
* **Comportamento:** Este é o comportamento padrão e mais intuitivo. O elemento mais específico lida com o evento primeiro, e depois seus pais têm a chance de reagir. É como uma bolha subindo na água.
* **Uso:** É o padrão para quase tudo. Permite uma técnica poderosa chamada **delegação de eventos** (vamos ver a seguir).
* **Como usar:** Você não precisa fazer nada de especial. Este é o comportamento padrão.

*// Este listener será acionado na fase de BORBULHAMENTO (padrão)*

elemento.addEventListener('click', minhaFuncao);

*// ou explicitamente:*

elemento.addEventListener('click', minhaFuncao, false);

**Exemplo Prático**

Vamos adicionar event listeners à nossa estrutura HTML para ver a propagação em ação:

<style>

div, span { padding: 20px; border: 1px solid; }

#avo { border-color: red; }

#pai { border-color: green; }

#filho { border-color: blue; }

</style>

<div id="avo">AVÔ

<div id="pai">PAI

<span id="filho">FILHO</span>

</div>

</div>

<script>

const avo = document.getElementById('avo');

const pai = document.getElementById('pai');

const filho = document.getElementById('filho');

// Listeners na fase de CAPTURA (descendo)

avo.addEventListener('click', () => console.log('Captura: AVÔ'), true);

pai.addEventListener('click', () => console.log('Captura: PAI'), true);

filho.addEventListener('click', () => console.log('Captura: FILHO (alvo)'), true);

// Listeners na fase de BORBULHAMENTO (subindo)

avo.addEventListener('click', () => console.log('Borbulhamento: AVÔ'), false);

pai.addEventListener('click', () => console.log('Borbulhamento: PAI'), false);

filho.addEventListener('click', () => console.log('Borbulhamento: FILHO (alvo)'), false);

</script>

Se você clicar no **FILHO**, o console mostrará:

Captura: AVÔ

Captura: PAI

Captura: FILHO (alvo)

Borbulhamento: FILHO (alvo)

Borbulhamento: PAI

Borbulhamento: AVÔ

Isso demonstra perfeitamente o fluxo: o evento desce (captura) e depois sobe (borbulhamento).

**Como Parar a Propagação?**

Às vezes, você não quer que o evento continue sua viagem. Por exemplo, ao clicar em um botão dentro de uma div que também tem um evento de clique, você pode querer que apenas a ação do botão seja executada.

Para isso, usamos o método event.stopPropagation().

pai.addEventListener('click', () => {

console.log('Clique no PAI');

});

filho.addEventListener('click', (*event*) => {

event.stopPropagation(); *// Impede que o evento "borbulhe" para o PAI*

console.log('Clique no FILHO. A propagação foi interrompida.');

});

Agora, se você clicar no **FILHO**, o console mostrará apenas "Clique no FILHO. A propagação foi interrompida.". O evento de clique do **PAI** nunca será acionado.

**Por que isso é tão útil? Delegação de Eventos**

O borbulhamento permite uma técnica de otimização chamada **delegação de eventos**. Em vez de adicionar um event listener a 100 botões em uma lista, você pode adicionar **apenas um** listener ao elemento pai (por exemplo, a <ul>).

Quando um botão é clicado, o evento "borbulha" até a <ul>, que o captura. Dentro do listener do pai, você pode verificar qual filho (event.target) originou o evento e agir de acordo.

**Vantagens:**

1. **Performance:** Menos event listeners na página.
2. **Dinâmica:** Funciona para elementos adicionados à lista *depois* que o listener foi criado, pois eles também estarão dentro do pai.

Este é um tópico um pouco mais avançado, mas é a aplicação prática mais importante do borbulhamento.