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

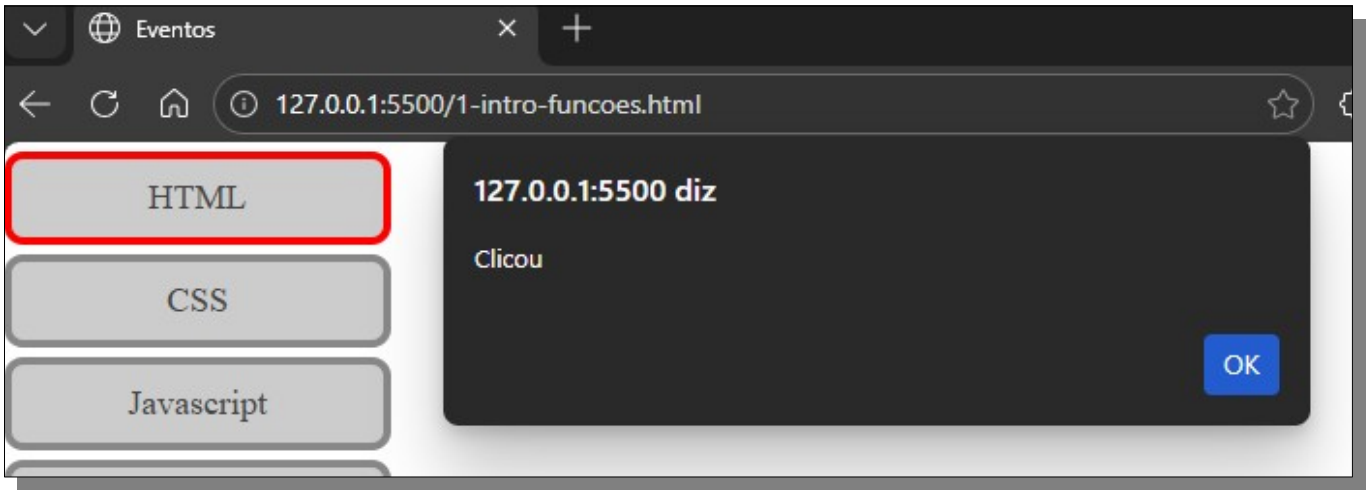
precisaria editar o arquivo HTML, em vez de manter toda a lógica de interação centralizada em seu arquivo JavaScript, o que torna a manutenção do projeto mais difícil e propensa a erros.
O atributo de evento `onclick` é o exemplo mais comum dessa abordagem.

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. EXERCÍCIO PRÁTICO

2.1. Preparação do Projeto: Estrutura e Estilo

Bem-vindo a este exercício prático! Nosso objetivo é construir, passo a passo, um componente interativo que permite transferir itens entre duas listas usando JavaScript puro. Ao longo desta atividade, vamos reforçar conceitos essenciais de manipulação do DOM (*Document Object Model*) e gerenciamento de eventos, habilidades fundamentais para qualquer desenvolvedor web. Vamos começar preparando a estrutura e o estilo do nosso componente.

2.1.1. O Código HTML Base

Primeiro, vamos definir a estrutura HTML do nosso projeto. O código abaixo cria dois contêineres principais (`divs`) e um botão de ação.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Aula 35</title>
    <link rel="stylesheet" type="text/css" href="estilo.css" />
</head>
<body>
    <main>
        <div id="caixa1" class="caixa">
            <div id="c1" class="curso c1">HTML</div>
            <div id="c2" class="curso c1">CSS</div>
            <div id="c3" class="curso c1">Javascript</div>
            <div id="c4" class="curso c1">PHP</div>
            <div id="c5" class="curso c1">React</div>
            <div id="c6" class="curso c1">MySQL</div>
        </div>
        <button id="btn_copiar">Copiar >></button>
        <div id="caixa2" class="caixa">
        </div>
    </main>
    <script src="script.js"></script>
</body>
</html>
```

Entendendo a Estrutura HTML:

- **`<div id="caixa1">`**: Este é o nosso contêiner de origem. Ele contém a lista inicial de cursos.
- **`<div class="curso">`**: Cada curso é representado por uma `div` com a classe `.curso`. Isso nos permitirá selecioná-los e estilizá-los de forma consistente.
- **`<button id="btn_copiar">`**: Este botão será o gatilho da nossa ação. Ao ser clicado, ele deverá mover os itens selecionados.
- **`<div id="caixa2">`**: Este é o contêiner de destino, que começa vazio e receberá os cursos transferidos.

1.3. A Estilização com CSS

Agora, vamos adicionar o estilo visual ao nosso componente. O CSS a seguir organizará os contêineres lado a lado e definirá a aparência dos cursos, incluindo um estilo especial para quando um item for selecionado.

```
* {
    padding: 0px;
    margin: 0px;
    border: none;
    box-sizing: border-box;
    font-size: large;
}

.curso {
    display: flex;
    justify-content: center;
    align-items: center;
    width: 200px;
    border: 4px solid #888;
    border-radius: 10px;
    padding: 10px;
    margin: 5px 0px;
    cursor: pointer;
}

.selecionado {
    background-color: #888 !important;
    color: #fcc !important;
    border-color: #f00 !important;
}

.c1 {
    background-color: #ccc;
    color: #444;
}

.caixa {
    border: 4px solid #000;
    background-color: #eee;
```

```
    padding: 10px;
    display: flex;
    flex-direction: column;
    justify-content: flex-start;
    align-items: center;
    width: 250px;
}

main {
    display: flex;
    justify-content: center;
    align-items: center;
    width: 100%;
    height: 100vh;
}

button {
    width: 150px;
    height: 40px;
    background-color: #008;
    color: #fff;
    cursor: pointer;
    border-radius: 10px;
    margin: 0px 5px;
}
```

Com a estrutura HTML e a estilização CSS prontas, temos a base visual do nosso projeto. O próximo passo é dar vida a este componente com a lógica de programação em JavaScript.

2.2. Parte 1: Implementando a Transferência Unidirecional

Nesta primeira fase, nosso objetivo é implementar a funcionalidade básica: permitir que o usuário selecione um ou mais cursos na primeira caixa e, ao clicar no botão "Copiar", mova esses cursos selecionados para a segunda caixa.

2.2.1. Selecionando os Elementos do DOM

O primeiro passo no nosso script é obter as referências para os elementos HTML com os quais vamos interagir. Armazenar essas referências em constantes facilita o acesso e a manipulação posterior.

```
const caixa1 = document.getElementById('caixa1');
const caixa2 = document.getElementById('caixa2');
const btn_copiar = document.getElementById('btn_copiar');
const todosCursos = document.querySelectorAll('.curso');
```

- **getElementById**: Usado para capturar elementos com um `id` único, como nossas caixas e o botão.
- **querySelectorAll**: Usado para capturar uma `NodeList` (uma coleção de nós) com todos os elementos que correspondem a um seletor CSS, neste caso, todos os `divs` com a classe `.curso`.

2.3. Adicionando Interatividade: A Lógica de Seleção

Para que o usuário possa selecionar os cursos, precisamos "escutar" o evento de clique em cada um deles. Uma boa prática de desenvolvimento é sempre verificar se estamos capturando o evento e o elemento corretos antes de implementar a lógica final. Vamos iterar sobre todos os cursos e adicionar um `addEventListener` a cada um, usando `console.log` para confirmar nossa seleção:

```
todosCursos.forEach(el => {
    el.addEventListener('click', (evt) => {
        console.log(evt.target);
    });
});
```

Abra o console do seu navegador e clique nos cursos. Você verá que cada clique exibe o elemento `div` correspondente. Isso confirma que nosso ouvinte de eventos está funcionando perfeitamente! Agora que validamos nosso `target`, podemos implementar a lógica de seleção. Substituímos o `console.log` pelo método `classList.toggle()`.

```
todosCursos.forEach(el => {
    el.addEventListener('click', (evt) => {
        const curso = evt.target;
        curso.classList.toggle('selecionado');
    });
});
```

Análise do Código:

- **forEach**: Iteramos sobre a `NodeList todosCursos` para aplicar a lógica a cada elemento individualmente.
- **classList.toggle('selecionado')**: Este método é a chave da nossa lógica de seleção. Quando um curso é clicado, ele verifica se a classe `selecionado` já existe no elemento. Se existir, ele a remove; se não existir, ele a adiciona. Isso cria um sistema de liga/desliga visualmente eficiente para marcar e desmarcar os itens.

Nota: *tente substituir o* ***.forEach*** *pelo* ***.map****, você verá que o método* ***.map*** *não funcionará, para que o mesmo funcione, é necessário "espalhar" utilizando o operador spread (…).*

2.4. Implementando a Ação de Cópia

Agora, vamos implementar a lógica que será executada quando o botão "Copiar" for clicado.

```
btn_copiar.addEventListener('click', () => {
    const cursosSelecionados = document.querySelectorAll('.selecionado');

    cursosSelecionados.forEach(el => {
        caixa2.appendChild(el);
    });
});
```

O fluxo de trabalho aqui é dividido em dois passos cruciais:

1. **Obter os selecionados:** Dentro do evento de clique, usamos `querySelectorAll('.selecionado')` para criar uma nova coleção contendo apenas os cursos que o usuário marcou.
2. **Mover os elementos:** Iteramos sobre essa coleção e, para cada elemento selecionado, usamos o método `caixa2.appendChild()`. É fundamental entender que **`appendChild` não copia o elemento, ele o move**. Quando um elemento é anexado a um novo "pai" (`caixa2`), ele é automaticamente removido de seu "pai" original (`caixa1`).

2.2.2. O Desafio a Ser Resolvido

Nossa implementação funciona, mas tem uma limitação importante. Se selecionarmos alguns cursos e os movermos para a `caixa2`, e depois desmarcarmos um deles, a lógica atual não prevê uma forma de retorná-lo para a `caixa1`. O sistema só funciona em uma direção. Este é o desafio que resolveremos na próxima etapa: criar uma transferência bidirecional.

2.3. Parte 2: A Solução Elegante com Transferência Bidirecional

Nesta seção, vamos abordar a solução completa para o desafio, implementando a transferência de itens nos dois sentidos. Você verá como uma pequena, mas poderosa, alteração no seletor CSS dentro do nosso JavaScript tornará a lógica robusta e completa, resolvendo o problema de forma elegante.

2.3.1. Pequenos Ajustes na Estrutura

Primeiro, vamos fazer uma pequena alteração semântica. Como a funcionalidade agora será de mover itens para ambos os lados, o termo "Transferir" é mais adequado do que "Copiar".

```
<button id="btn_transferir">Transferir >></button>
```

No JavaScript, vamos criar uma nova constante para o novo botão, garantindo que nosso código permaneça limpo e consistente.

```
const btn_transferir = document.getElementById('btn_transferir');
```

2.3.2. A Chave da Solução: O Processo de Descoberta do Seletor `:not()`

O segredo para a nossa solução bidirecional está em identificar não apenas os cursos selecionados, mas também os **não selecionados**. A pergunta é: como podemos fazer isso? A primeira ideia que pode surgir é usar a pseudoclasse `:not()`, que serve para excluir elementos de uma seleção. Vamos tentar uma abordagem ingênua:

```
// Tentativa inicial - Não faça isso!
const cursosNaoSelecionados = document.querySelectorAll(':not(.selecionado)');
console.log(cursosNaoSelecionados);
```

Se você executar esse código e clicar no botão, verá no console um resultado inesperado: ele retorna uma `NodeList` com quase todos os elementos da página (`<html>`, `<head>`, `<body>`, etc.). Isso acontece porque o seletor `:not(.selecionado)` é muito amplo; ele seleciona **qualquer elemento** no documento que não tenha a classe `.selecionado`.

Aqui está o "pulo do gato" de um desenvolvedor experiente: precisamos **limitar o escopo** da nossa busca antes de aplicar o filtro de exclusão. A solução correta é primeiro selecionar os elementos que nos interessam (aqueles com a classe `.curso`) e, *depois*, aplicar o `:not()` para filtrar os que não queremos.

```
// A forma correta e precisa
const NaoSelecionados = document.querySelectorAll('.curso:not(.selecionado)');
```

Análise do Seletor Vencedor:

- **`querySelectorAll('.curso:not(.selecionado)')`**: Esta linha é o coração da nossa solução. Ela instrui o navegador a selecionar todos os elementos que:
    1. Primeiro, possuem a classe `.curso`.
    2. E, dentre eles, **NÃO** possuem a classe `.selecionado`.

Esse processo de refinar seletores é uma habilidade crucial e demonstra o poder de usar CSS de forma inteligente dentro do JavaScript.

2.3.3. Finalizando a Lógica de Transferência Completa

Com as duas listas em mãos (selecionados e não selecionados), a lógica final se torna simples e direta. A cada clique no botão, vamos "re-classificar" todos os itens, movendo cada um para seu devido contêiner.

```
btn_transferir.addEventListener('click', () => {
const cursosSelecionados = document.querySelectorAll('.selecionado');
const NaoSelecionados = document.querySelectorAll('.curso:not(.selecionado)');

    cursosSelecionados.forEach(el => {
        caixa2.appendChild(el);
    });

    NaoSelecionados.forEach(el => {
        caixa1.appendChild(el);
    });
});
```

Explicação do Fluxo:

1. O primeiro loop (`forEach`) percorre todos os **cursos selecionados** e os anexa à `caixa2`. Se um item já estiver lá, ele permanecerá. Se estiver na `caixa1`, ele será movido.
2. O segundo loop percorre todos os **cursos não selecionados** e os anexa à `caixa1`. Da mesma forma, isso garante que qualquer item desmarcado (que não tenha a classe `.selecionado`) retorne ou permaneça na caixa da esquerda.

Com essa lógica, a cada clique, o estado da interface é completamente sincronizado com a seleção do usuário, resolvendo nosso desafio de forma completa e robusta.

2.4. Código Final e Conclusão

2.4.1. Resumo do Aprendizado

Neste exercício, percorremos uma jornada completa: começamos com a configuração de uma estrutura HTML e CSS, implementamos uma funcionalidade inicial com suas limitações e, finalmente, evoluímos para uma solução final elegante e robusta. Esse processo reflete o dia a dia do desenvolvimento de software, onde aprimoramos e refatoramos nosso código para atender a todos os requisitos.

2.4.2. Código JavaScript Consolidado

Aqui está o código JavaScript final e completo para sua referência.

```
const caixa1 = document.getElementById('caixa1');
const caixa2 = document.getElementById('caixa2');
const btn_transferir = document.getElementById('btn_transferir');
const todosCursos = document.querySelectorAll('.curso');

todosCursos.forEach(el => {
    el.addEventListener('click', (evt) => {
        const curso = evt.target;
        curso.classList.toggle('selecionado');
    });
});

btn_transferir.addEventListener('click', () => {
    const cursosSelecionados = document.querySelectorAll('.selecionado');
                                    const         cursosNaoSelecionados         =
document.querySelectorAll('.curso:not(.selecionado)');

    cursosSelecionados.forEach(el => {
        caixa2.appendChild(el);
    });

    cursosNaoSelecionados.forEach(el => {
        caixa1.appendChild(el);
    });
});
```

2.5. Principais Habilidades Desenvolvidas

Ao concluir este exercício, você praticou e fortaleceu diversas habilidades essenciais em JavaScript:

- **Seleção de Elementos do DOM:** Uso eficiente de `getElementById` e `querySelectorAll` para capturar os elementos necessários na página.
- **Manipulação de Eventos:** Aplicação de `addEventListener` para criar interatividade e responder às ações do usuário.

- **Gerenciamento de Classes CSS:** O poder do método `classList.toggle` para gerenciar estados visuais de forma dinâmica.
- **Movimentação de Elementos:** Compreensão do comportamento do `appendChild`, que move nós no DOM em vez de apenas copiá-los.
- **Seletores CSS Avançados:** A utilidade da pseudoclasse `:not()` para criar seleções complexas e otimizar a lógica de programação.

Continue praticando e explorando as vastas possibilidades que a manipulação do DOM com JavaScript oferece. Parabéns por concluir este desafio!