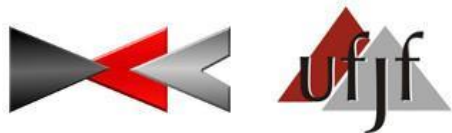

Manipulando o DOM

UFJF - DCC202 - Desenvolvimento Web

Prof. Igor Knop igor.knop@ufjf.br



<https://bit.ly/3nCF1Dz>

Manipulando o DOM

Objetivos:

- Representação do documento em objetos;
- Encontrar elementos na árvore;
- Alterar o conteúdo, propriedades e estilos;
- Criar novos elementos.

Manipulação do Modelo de Documento

Uma aplicação web é um programa que altera a representação de um documento HTML e seu estilo em resposta às ações do usuário.

Quando uma página é acessada pelo navegador, sua representação final pode ser manipulada através de uma interface de aplicações (API).

Essa API é acessada via uma série de objetos JavaScript, em particular o **Document**, e é conhecida como *Document Object Model* (DOM)

Objetos principais

Os principais objetos para manipulação das páginas em um navegador são:

- window
- navigator
- document

Objetos principais (2)

Os principais objetos para manipulação das páginas em um navegador são:

- window
 - A aba do navegador que a página está carregada;
 - Responder a eventos de controle do usuário;
 - Configurar timers e persistência de dados local.

Objetos principais (3)

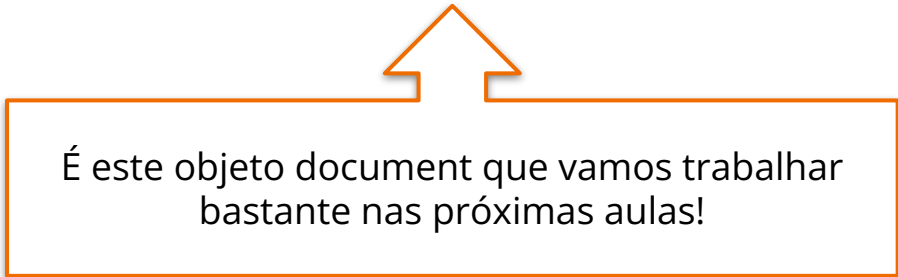
Os principais objetos para manipulação das páginas em um navegador são:

- navigator
 - Representa o navegador e sua versão;
 - Obtém a localização do usuário, sua linguagem e acesso a dispositivos como câmeras e microfone.

Objetos principais (4)

Os principais objetos para manipulação das páginas em um navegador são:

- document
 - É a representação da estrutura e estilos aplicados ao documento;
 - Permite adicionar e remover elementos;
 - Alterar classes e estilos individuais dos elementos.




É este objeto document que vamos trabalhar bastante nas próximas aulas!

DOM

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Página</title>
  </head>

  <body>
    <h1>Olá mundo!</h1>
  </body>
</html>
```

Olá mundo!



The screenshot shows a web browser window with the text "Olá mundo!" displayed. To the right of the text is a DOM tree view. The DOM tree shows the following structure:

```
<!DOCTYPE html>
<html>
  ▼ <head>
    <meta charset="UTF-8">
    <title>Página</title>
  </head>
  ▼ <body>
    ... <h1>Olá mundo!</h1> == $0
  </body>
</html>
```


Seleção de elementos

Para a manipulação de um determinado elemento, deve-se primeiro criar uma referência para ele no código. Existem vários métodos para se obter um ou mais objetos:

- `document.getElementsByTagName()`
- `document.getElementsByClassName()`
- `document.getElementById()`
- `document.querySelectorAll()`
- `document.querySelector()`

Seleção de elementos

Para a manipulação de um determinado elemento, deve-se primeiro criar uma referência para ele no código. Existem vários métodos para se obter um ou mais objetos:

- `document.getElementsByTagName()` **Array**
- `document.getElementsByClassName()` **Array**
- `document.getElementById()` **Element**
- `document.querySelectorAll()` **Array** (recomendado)
- `document.querySelector()` **Element** (recomendado)

Seleção de elementos: `querySelectorAll()`

O método `querySelectorAll()` vai retornar um vetor com todos elementos que correspondem à busca.

Ele exige um argumento com um seletor em CSS para encontrar os elementos. Por exemplo:

- `document.querySelectorAll("span")`
Todos os elementos `span`
- `document.querySelectorAll("div.ativo")`
Todos os `div` com a classe "ativo"
- `document.querySelectorAll("#dashboard > h1")`
Todos os elementos `h1` que são filhos imediatos de um elemento com id "dashboard"
- `document.querySelectorAll(".produto[data-estoque=0]")`
Todos os elementos da classe "produto" que possuam o atributo `data-estoque` com valor 0

Seleção de elementos: `querySelector()`

O método `querySelector()` vai retornar apenas o primeiro elemento que corresponde à busca.

Ele exige um argumento com um seletor em CSS para encontrar os elementos. Por exemplo:

- `document.querySelector("h1")`

O primeiro elemento com a etiqueta `h1`

- `document.querySelector("#dashboard")`

O primeiro elemento com id “dashboard”

- `document.querySelector("input[name=nome]")`

O primeiro elemento `input` com a propriedade “name” com o valor “nome”.

exemplo01.js:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>Página</title>
  </head>
  <body>
    <h1>Olá Mundo!</h1>
    <script>
      const p = document.querySelector('h1');
      p.textContent = 'Olá mundo JavaScript!';
    </script>
  </body>
</html>
```

A estrutura será a entrada e saída

```
<!DOCTYPE html>
<html lang="pt">
  <head>
    <meta charset="utf-8">
    <title>Exemplo 01</title>
  </head>
  <body>
    <h1>Exemplo 01</h1>
    <p>Este é o exemplo 01.</p>
    <p>A soma de <span contenteditable="true">2</span> e
      <span contenteditable="true">3</span> é <span>?</span></p>
    <button>Calcular</button>
    <script></script>
  </body>
</html>
```

Referências para os elementos que vão servir de entrada e saída

...

```
<script>
// Captura as entradas e saídas de dados
const spans = document.querySelectorAll("span");

// Converte para números
const a = Number(spans[0].textContent);
const b = Number(spans[1].textContent);

// Calcula e escreve resultado na saída
spans[2].textContent = a + b;
</script>
```

...

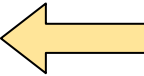
Referências para os elementos que vão servir de entrada e saída

...

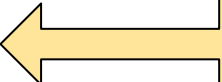
```
<script>
// Captura as entradas e saídas de dados
const spans = document.querySelectorAll("span");

// Converte para números
const a = Number(spans[0].textContent);
const b = Number(spans[1].textContent);

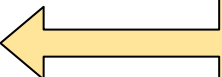
// Calcula e escreve resultado na saída
spans[2].textContent = a + b;
</script>
```



Captura todos os span da página em um vetor



Os dois primeiros span atuam como entrada de dados



O terceiro span vai servir de saída de dados!

...

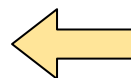
Captura de eventos

...

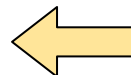
```
<script>
const botao = document.querySelector("button");
botao.addEventListener("click", calcular);

function calcular(){
  const spans = document.querySelectorAll("span");
  const a = Number(spans[0].textContent);
  const b = Number(spans[1].textContent);
  spans[2].textContent = a + b;
}
</script>
```

...



Capture o elemento
que vai gerar o evento
e registra-se um
ouvinte



O código que deve ser
executado é chamado
quando o evento é
disparado

Criando elementos e estilo

```
<!DOCTYPE html>
<html lang="pt">
<head>
  <meta charset="utf-8">
  <title>Exemplo 02</title>
  <style>
    .comprado { text-decoration: line-through; }
  </style>
</head>
```

...

A estrutura a ser modificada

...

```
<body>
```

```
  <h1>Exemplo 02</h1>
```

```
  <p>Lista de compras:</p>
```

```
  <input type="text" placeholder="nome do item" />
```

```
  <button>Adicionar</button>
```

```
  <ul></ul>
```

```
  <script></script>
```

```
</body>
```

```
</html>
```

Crie referências para os elementos e adicione os ouvintes para os eventos

```
<script>
```

```
  //Referenciar as entradas
```

```
  const entrada = document.querySelector("input");
```

```
  const botao = document.querySelector("button");
```

```
  //Referenciar as saídas
```

```
  const lista = document.querySelector("ul");
```

```
  //Registrar eventos para cálculos
```

```
  botao.addEventListener("click", adicionar);
```

```
...
```

Criando novos elementos

...

//Escrever resultados na saída

```
function adicionar() {  
    const textoItem = entrada.value;  
    console.log(textoItem);  
    entrada.value = "";  
    entrada.focus();  
    const novoLi = document.createElement("li");  
    novoLi.textContent = textoItem;  
    console.log(novoLi);  
    lista.appendChild(novoLi);  
}
```

Novos elementos também podem disparar eventos

...

```
//Escrever resultados na saída
function adicionar() {
  const textoItem = entrada.value;
  console.log(textoItem);
  entrada.value = "";
  entrada.focus();
  const novoLi = document.createElement("li");
  novoLi.textContent = textoItem;
  console.log(novoLi);
  lista.appendChild(novoLi);
  novoLi.addEventListener("li", comprar);
}
```

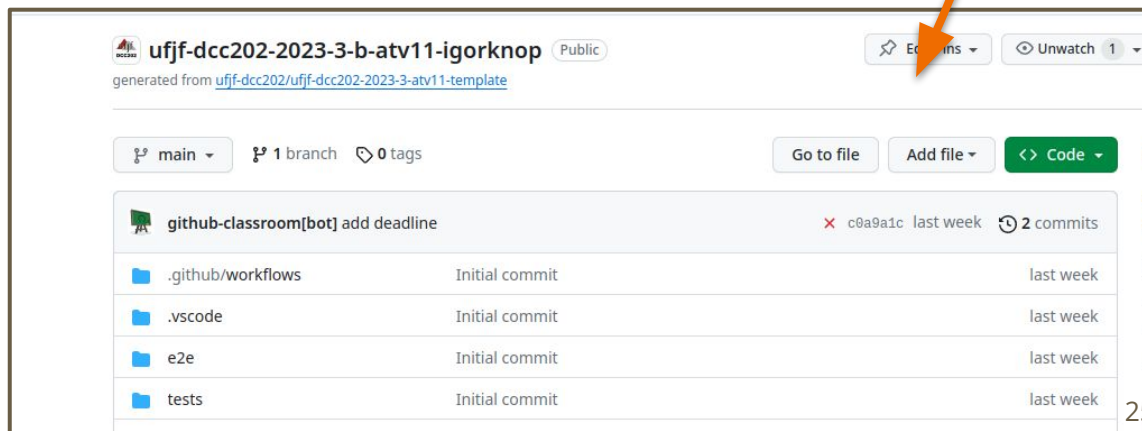
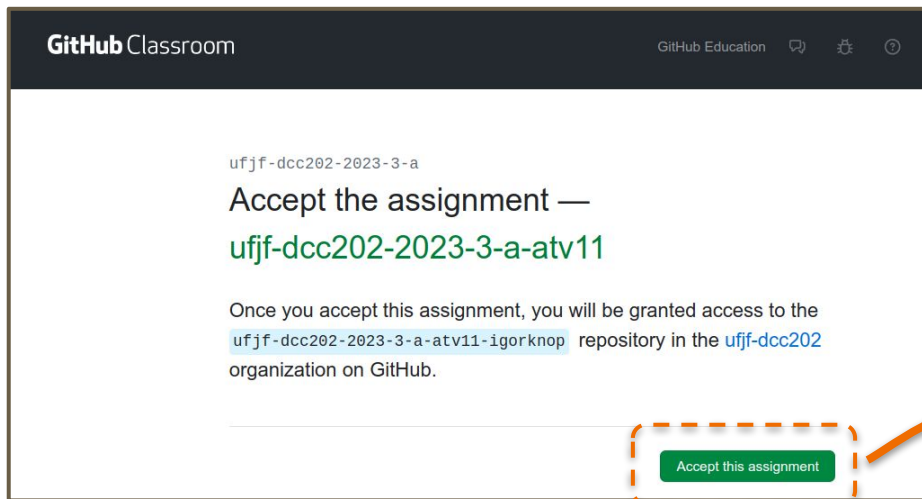
Altere as classes de estilo

```
...  
  
function comprar() {  
    if (this.classList.contains("comprado")) {  
        this.classList.remove("comprado");  
    } else {  
        this.classList.add("comprado");  
    }  
}  
  
</script>  
</body>  
</html>
```


Exercícios: aceite a atv11

Aceite a atividade 11 e copie o endereço do repositório criado por ela.

Faça o clone usando o seu VSCode e usando o protocolo HTTP.



Instale os pacotes e veja os testes falharem

Com o repositório clonado, confira se está com o usuário certo no git com:

```
git config user.name  
git config user.email
```

Instale os pacotes com:

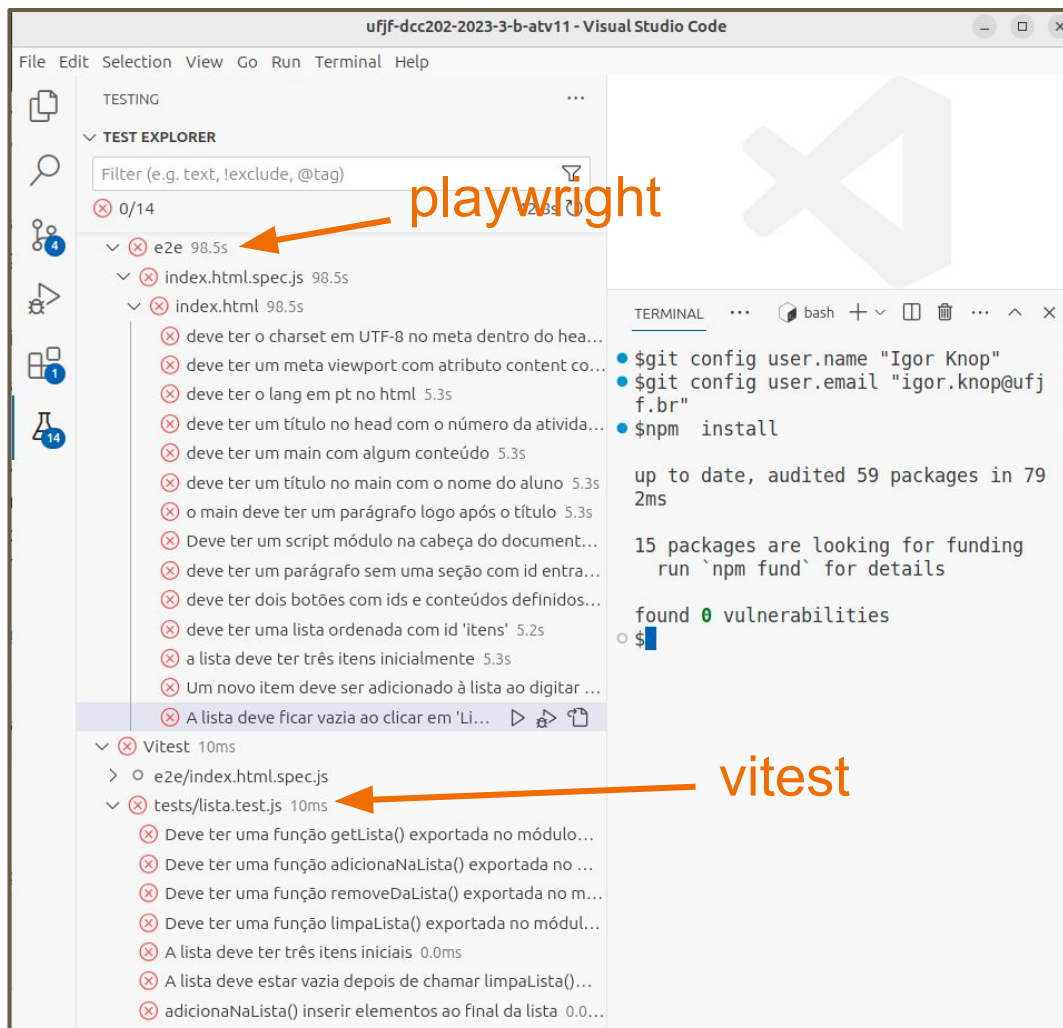
```
npm install
```

Execute o Live Server para expor a pasta do projeto e execute os testes, tanto no playwright quanto no vitest.

Lembre-se de não executar os testes e2e no vitest!

Ambos os testes podem ser feitos nas suas extensões ou pelo terminal com:

```
npm run e2e  
npm run test
```



Crie um arquivo index.html

Na pasta raiz do projeto, crie um arquivo `index.html`.

Além dos elementos básicos, adicione na cabeça do documento o carregamento de um módulo `main.js`.

Esse módulo `main.js` vai ser o responsável por ligar a nossa estrutura do HTML com o modelo que vamos criar.

```
<!DOCTYPE html>
<html lang="pt">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>DCC202 - Atividade 11: Igor Knop</title>
    <script src="main.js" type="module"></script>
  </head>

  <body>
    <main>
      <h1>DCC202 - Atividade 11: Igor Knop</h1>
      <p>Esta é a página principal tem código JavaScript nela.
Você deve interagir com os elementos e ver o resultado na
página.</p>
      <section>
        <h2>Lista</h2>
        <p id="entrada" contenteditable>Item</p>
        <button id="adicionar">Adicionar</button>
        <button id="limpar">Limpar Lista</button>
        <ol id="itens"></ol>
      </section>
    </main>
  </body>
</html>
```

Estrutura da aplicação

Além dos elementos básicos, adicione uma seção com:

- um título;
- um parágrafo com atributos **id** e **contenteditable**;
- um elemento **button** com **id** adicionar;
- um elemento **button** com um **id** limpar;
- uma lista ordenada.

Os ids vão nos permitir encontrar os elementos e o **contenteditable** libera a edição do conteúdo do elemento pelo usuário direto no navegador.

```
<!DOCTYPE html>
<html lang="pt">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>DCC202 - Atividade 11: Igor Knop</title>
    <script src="main.js" type="module"></script>
  </head>

  <body>
    <main>
      <h1>DCC202 - Atividade 11: Igor Knop</h1>
      <p>Esta é a página principal tem código JavaScript nela.
Você deve interagir com os elementos e ver o resultado na
página.</p>
      <section>
        <h2>Lista</h2>
        <p id="entrada" contenteditable>Item</p>
        <button id="adicionar">Adicionar</button>
        <button id="limpar">Limpar Lista</button>
        <ol id="itens"></ol>
      </section>
    </main>
  </body>
</html>
```

Crie um main.js

Ainda na pasta raiz, crie um módulo main.js que terá a responsabilidade de ligar a nossa estrutura em nosso modelo.

Como ainda não temos o modelo, por enquanto ele vai apenas selecionar os elementos que temos interesse, criando referências para eles.

```
//main.js
```

```
const olItens = document.querySelector("#itens");  
const pEntrada = document.querySelector('#entrada');  
const btnAdicionar = document.querySelector("#adicionar");  
const btnLimpar = document.querySelector("#limpar");
```

Crie um módulo lista.js

Nós vamos simplesmente modelar uma lista de palavras.

Para servir de estado, vamos usar um vetor **lista** com três textos iniciais.

Além disso, vamos criar uma função de acesso **getLista()** e já exportá-la com a palavra reservada **export**.

Essa função não vai retornar a nossa lista diretamente pois não é um tipo primitivo. Isso faria expor a mesma referência externamente.

Por isso vamos usar a função **structuredClone()** para fazer uma cópia profunda, garantindo que nada fora do módulo tenha acesso a esse estado.

```
// lista.js  
const lista = ["Um", "Dois", "Três"];
```

```
export function getLista() {  
  return structuredClone(lista);  
}
```

```
export function limpaLista() {  
  lista.splice(0);  
}
```

Crie um módulo lista.js (2)

Já a função **limpaLista()** vai permitir excluir todos os elementos da lista.

O método **splice()** do vetor permite remover um pedaço, modificando o vetor original.

Normalmente precisamos passar dois parâmetros: onde começar a cortar e quantos itens cortar. Se o segundo parâmetro estiver ausente, ele pega até o final do vetor.

Então nesse formato, ele vai pegar um pedaço do primeiro item do vetor (posição 0) até o final.

Se não quiséssemos alterar o vetor original, usamos o método **toSpliced()**.

```
// lista.js  
const lista = ["Um", "Dois", "Três"];
```

```
export function getLista() {  
  return structuredClone(lista);  
}
```

```
export function limpaLista() {  
  lista.splice(0);  
}
```

Atualizando a lista no main.js

Vamos voltar no main.js e agora criar uma função `atualizarLista()` que vai ser responsável por:

1. apagar todo o conteúdo do elemento de lista na nossa estrutura;
2. cria um elemento item de lista para cada item do nosso estado;

A propriedade `innerHTML` permite alterar toda a estrutura interna de um elemento. Ao colocar um string vazia, estamos esvaziando a lista. Alterar essa propriedade causa o redesenho da árvore de elementos.

Para criar um novo elemento de lista, usamos o método `document.createElement()`. Esse novo elemento tem que ser adicionado ao documento em algum para que apareça no navegador do usuário!

```
//main.js
```

```
import { getLista } from "../lista.js";
```



```
const olItens = document.querySelector("#itens");  
const pEntrada = document.querySelector("#entrada");  
const btnAdicionar = document.querySelector("#adicionar");  
const btnLimpar = document.querySelector("#limpar");
```

```
function atualizarLista() {  
  olItens.innerHTML = "";  
  let lista = getLista();  
  for (let i = 0; i < lista.length; i++) {  
    const li = document.createElement('li');  
    li.textContent = lista[i];  
    olItens.appendChild(li);  
  }  
}
```


Atualizando a lista no main.js

Vamos voltar no main.js e agora criar uma função `atualizarLista()` que vai ser responsável por:

1. apagar todo o conteúdo do elemento de lista na nossa estrutura;
2. cria um elemento item de lista para cada item do nosso estado;

A propriedade `innerHTML` permite alterar toda a estrutura interna de um elemento. Ao colocar um string vazia, estamos esvaziando a lista. Alterar essa propriedade causa o redesenho da árvore de elementos.

Para criar um novo elemento de lista, usamos o método `document.createElement()`. Esse novo elemento tem que ser adicionado ao documento em algum para que apareça no navegador do usuário!

```
//main.js
```

```
import { getLista } from "../lista.js";
```

```
const olItens = document.querySelector("#itens");  
const pEntrada = document.querySelector("#entrada");  
const btnAdicionar = document.querySelector("#adicionar");  
const btnLimpar = document.querySelector("#limpar");
```

```
atualizarLista();
```

```
function atualizarLista()  
{  
  olItens.innerHTML = "";  
  let lista = getLista();  
  for (let i = 0; i < lista.length; i++) {  
    const li = document.createElement('li');  
    li.textContent = lista[i];  
    olItens.appendChild(li);  
  }  
}
```

Para ter efeito logo quando o script é carregado, é preciso chamar o `atualizarLista()`

Lista com itens iniciais

A lista deve, neste ponto, estar preenchida com itens de lista que equivalem ao nosso estado.

Essa atualização só é realizada no carregamento da página. Toda operação que altera a lista, deve, direta ou indiretamente, chamar **atualizarLista()**; para que a lista seja redesenhada no documento.

Vamos fazer isso com o botão de Limpar Lista...

← → ↻ ⓘ localhost:5500

DCC202 - Atividade 11: Igor Knop

Esta é a página principal tem código JavaScript nela. Você deve interagir com os elementos e página.

Lista

Item

Adicionar Limpar Lista

1. Um
2. Dois
3. Três

JS lista.js > adicionaNaLista

```
1 // lista.js
2 const lista = ["Um", "Dois", "Três"];
3
4 export function getLista() {
5   return structuredClone(lista);
6 }
7
```

Adicionando um ouvinte ao botão

Crie uma nova função `limparItensDeLista()`. Essa função vai usar a `limpaLista()` que agora deve ser importado do módulo e também a `atualizarLista()`, que está omitida no código.

Precisamos agora cadastrar um ouvinte ao evento `'click'` do botão. Para isso usamos o método `addEventListener()`.

Sempre que o usuário interagir com a tela, ele vai chamar a função passada como argumento, a `limparItensDeLista`. Essa função também é conhecida como função de *callback*.



```
//main.js
import { getLista, limpaLista } from "../lista.js";

const olItens = document.querySelector("#itens");
const pEntrada = document.querySelector("#entrada");
const btnAdicionar = document.querySelector("#adicionar");
const btnLimpar = document.querySelector("#limpar");
```

```
btnLimpar.addEventListener('click', limparItensDeLista);
```



```
function limparItensDeLista() {
  limpaLista();
  atualizarLista();
}
```

...