Programação para Internet

Módulo 4

Páginas Interativas com JavaScript

Prof. Dr. Daniel A. Furtado - FACOM/UFU

Conteúdo da Módulo

- Introdução à Linguagem JavaScript
- Recursos Básicos da Linguagem
- Document Object Model – Árvore DOM
- Tratamento de Eventos
- Manipulação da Árvore DOM

O que é JavaScript?

- Linguagem de script de alto nível integrada nos navegadores
- Permite prover interatividade e dinamismo a websites
- Permite programar o comportamento da página Web na ocorrência de eventos
- Permite alterar o documento HTML dinamicamente
- Comumente executada no lado cliente (front-end), pelo navegador de Internet
  - Linguagem interpretada pelo navegador
  - Não é necessário compilar explicitamente o código JavaScript
- Também pode ser utilizada no lado servidor (back-end)
  - Utilizando ferramentas como o Node.js
- JavaScript é as vezes referenciada pela abreviação JS
- Não confundir com a linguagem de programação Java

O que posso fazer com JavaScript?

- Modificar o conteúdo dos elementos HTML da página
- Adicionar novos elementos HTML na página dinamicamente
- Remover elementos HTML da página dinamicamente
- Modificar os atributos dos elementos dinamicamente
- Modificar os estilos CSS dos elementos dinamicamente
- Fazer requisições HTTP assíncronas
- Validar formulários etc.

JavaScript e ECMAScript

- Ecma International – organização que desenvolve padrões
- **ECMAScript** é uma linguagem padronizada, uma especificação
  - ECMA-262 é o nome do padrão propriamente dito
  - Está em sua 16ª edição, a ECMAScript 2025
- JavaScript é uma implementação da linguagem ECMAScript
- Há também outras implementações como JScript e ActionScript
- JavaScript foi desenvolvida originalmente por Brendan Eich da Netscape

Conteúdo Complementar

Inserindo JavaScript de Forma Embutida no HTML

```
<html>

    <head>
        <script>
        // Código JavaScript
        </script>
    </head>

    <body>
        ...
    </body>

</html>
```

```
<html>

    <head>
        ...
    </head>

    <body>
        ...

        <script>
        // Código JavaScript
        </script>
    </body>

</html>
```

O código JavaScript pode ser inserido de forma embutida dentro do próprio arquivo HTML utilizando o elemento <script> da HTML. O código pode ser inserido na região de cabeçalho, no corpo da página ou até mesmo depois da tag </body>.

JavaScript em Arquivo Separado

Arquivo HTML

```
<html>

  <head>

    <script src="arquivoJavaScript.js"></script>

  </head>

  <body>

    ...

  </body>

</html>
```

Arquivo JavaScript (.js)

```
/* arquivoJavaScript.js */

alert('Hello World!');
```

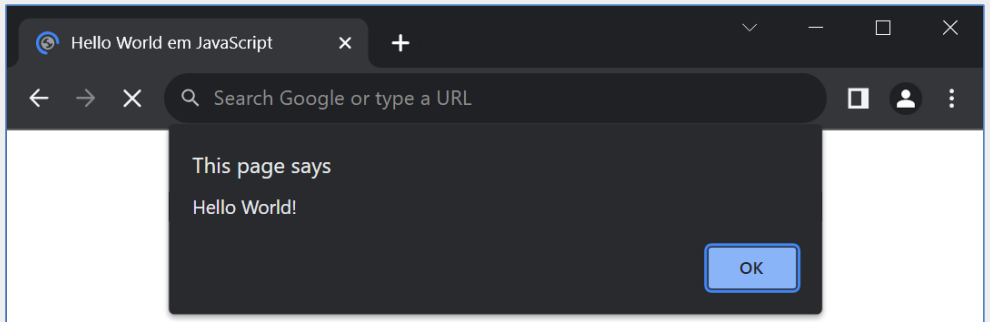
Uma forma melhor de inserir o código JavaScript é utilizar um arquivo externo e referenciá-lo na tag <script> com o atributo src. Também há a possibilidade de inserir JavaScript como **módulo**, mas esse assunto não será abordado neste material (módulos são inseridos com <script type="module">)

Vantagens do JavaScript em Arquivo Separado

- Melhor separação entre conteúdo (HTML) e comportamento (código JS)
- HTML e JavaScript conciso - código mais fácil de ler e manter
- Possibilidade de reutilizar o código JavaScript em vários arquivos HTML
- Arquivos JavaScript podem ser mantidos em cache pelo navegador
  - Maior agilidade no carregamento

Hello World em JavaScript

```html
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <title>Hello World em JavaScript</title>
</head>

<body>

  <h1>Minha primeira página com JavaScript</h1>
  <p>Lorem ipsum dolor sit amet.</p>

  <script>
    alert("Hello World!");
  </script>
</body>

</html>
```



Neste exemplo, a janela de alerta com a mensagem Hello World! é apresentada ao usuário durante o carregamento da página. O conteúdo propriamente dito (título e parágrafo) será mostrado depois que o usuário clicar no botão OK e a página terminar de ser carregada.

Execução do Código JavaScript no Navegador

- Fase 1 – Execução durante o carregamento da página
  - O código JavaScript fora de funções é executado durante a fase de **carregamento** do documento HTML, a medida em que o código é carregado pelo navegador (como no exemplo "hello world" do slide anterior)
  - Normalmente esse código faz inicializações ou registra funções tratadoras de eventos (event handlers), as quais serão chamadas posteriormente
  - Se há vários trechos de código inseridos com múltiplas tags <script>, então eles serão executados de forma síncrona, na ordem em que aparecem no documento HTML*
- Fase 2 – Execução em resposta a eventos
  - Na prática, a maior parte do código JavaScript é normalmente executada depois que a página é carregada, à medida em que o usuário interage e leva a ocorrência eventos como clique de botão, rolagem do conteúdo, novo dado de rede necessário etc.

***OBS**: esse comportamento pode ser alterado com os atributos async / defer

Execução do Código JavaScript no Navegador

```
<body>
  <h2>Clique na imagem a seguir</h2>
  <img src="images/logoUFU.png">

  <script>
    // Esta função será executada apenas na fase 2, quando o
    // usuário clicar na imagem
    function welcome() {
      alert("Seja bem vindo!");
    }

    // Esta linha é executada durante o carregamento da página (fase 1) e
    // tem como objetivo registrar a função welcome() para tratar
    // o evento 'click' na imagem. Porém a função welcome() em si não
    // é chamada durante o carregamento. Ela será chamada posteriormente,
    // quando o usuário clicar na imagem.
    document.images[0].onclick = welcome;
  </script>
</body>
```

JavaScript no Navegador – Threading Model

- De uma forma geral, o código JavaScript no navegador executa em modo de **thread única** (single-threaded), de forma síncrona
  - Por exemplo, enquanto uma função JavaScript estiver sendo executada, outras funções tratadoras de eventos precisam aguardar
  - Portanto, deve-se evitar operações que demoram para finalizar, sob pena de congelar a interface do usuário e causar uma experiência de navegação ruim
- Em alguns casos é possível executar o código JavaScript de forma assíncrona, como em requisições Ajax ou em funções definidas com **async** / **await**
- Também é possível executar o código JavaScript de forma assíncrona utilizando **web workers,** mas com acesso limitado ao contexto da thread principal

Recursos Básicos da Linguagem

Observações Gerais

- JavaScript é sensível a maiúsculas e minúsculas (case sensitive)
- Declarações podem ou não terminar com o ponto-e-vírgula
- Os tipos das variáveis são definidos automaticamente
- Comentários de linha: `// comentário`
- Comentários de bloco: `/* comentário */`

Estruturas Condicionais e de Repetição

```
if (expressão) {
   // operações se verdadeiro
}
else {
   // operações se falso
}
```

```
switch (expressao)      {
        case condicao1:
                // operações
                break;

        case condicaoN:
                // operações
                break;
        ...
        default:
                // operações
}
```

```
for (let i = 0; i < 10; i++)
{
   // operações
}
```

```
for (let item of array)
{
   // operações
}
```

```
while (expressao)
{
   // operações
}
```

```
do {

   // operações

} while (expressao)
```

Declaração de Variáveis

var nomeDaVariável = valorInicial

- Variável com escopo local se declarada dentro de uma função
- Variável com escopo global se declarada fora de funções
- Pode ser redeclarada. Pode ter valor atualizado
- Variáveis globais podem ser acessadas pelo objeto window (ex. window.idade)

let nomeDaVariável = valorInicial

- Variável tem escopo restrito ao bloco de código onde é declarada
- Pode ser acessada e atualizada apenas dentro do bloco
- Não pode ser redeclarada no mesmo bloco
- Se definida fora de funções, pode ser acessada de qualquer lugar, mas não com window.variavel

const nomeDaConstante = valor

- Semelhante a let, porém a variável não pode ser atualizada
- Deve ser inicializada no momento da declaração

Exemplos de Declarações de Variáveis

```
<script>

const pi = 3.14;
var soma = 0;      // soma é uma variável global
for (let i = 1; i <= 10; i++) {
    soma += i;  // i só pode ser acessada neste for
}

if (soma > 50) {
    let k = soma + pi;  // k só pode ser acessada neste bloco
    var m = k + 1;  // m poderá ser acessada fora do if
    console.log(k);
}

console.log(m); // mostrará o valor de m normalmente
console.log(k); // erro, pois k é restrita ao 'if' acima
</script>
```

Operadores Aritméticos, Relacionais e Lógicos

Operadores Aritméticos e de Atribuição

Operador	Significado
+	Adição (e concatenação)
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira
++	Incremento
--	Decremento
=	Atribuição
+=	Atribuição com soma ou concatenação
-=	Atribuição com sub.

Operadores Relacionais e Lógicos

Operador	Significado
==	Comparação por igualdade
===	Comparação por igualdade, incluindo valor e tipo
!=	Diferente
>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual a
&&	“E” lógico
||	“Ou” lógico
!	Negação lógica

Conteúdo Complementar

Operador de Adição e Concatenação

- O operador **+** deve ser utilizado com atenção
- Permite somar ou concatenar, dependendo dos operandos
- Se um dos operandos é uma **string** então será feita a **concatenação**
  - O outro operando é convertido para string, caso não seja
- Se os dois operandos são **numéricos** então é realizada a **soma**
- Exemplos

```
let x = 5 + 5;     // x terá o valor 10
let y = '5' + 5;   // y terá a string ‘55’
let z = true + '5';  // z terá a string ‘true5’
```

Conteúdo Complementar

Diferença dos Operadores == e ===

- Operador ==
  - Compara apenas valores
  - Operandos de tipos diferentes são convertidos e os valores resultantes são comparados
- Operador ===
  - Compara o valor e o tipo dos operandos
  - Comparação com operandos de tipos diferentes sempre resulta em false
- Exemplos
  - `1 == true;  // retorna true, pois true é convertido para 1`
  - `2 == true;  // retorna false`
  - `1 === true;  // retorna false (tipos diferentes);`
  - `10 == "10"  // retorna true depois de converter 10 para a string "10"`
  - `10 === "10"  // retorna false; (tipos diferentes);`

Objetos window, navigator, document e console

window

- Objeto global que representa a aba do navegador contendo a página
- Possibilita obter informações ou realizar ações a respeito da janela, como:
  - Obter dimensões: window.innerWidth e window.innerHeight
  - Executar uma ação quando a página for carregada, fechada etc.
  - Mostrar mensagens de alerta: window.alert("mensagem");

navigator (ou window.navigator)

- Representa o navegador de Internet em uso (browser, user-agent)
- Fornece dados como o idioma do navegador, geolocalização, memória etc.
- Exemplo: window.alert(navigator.language); // mostra “pt-BR”

document (ou window.document)

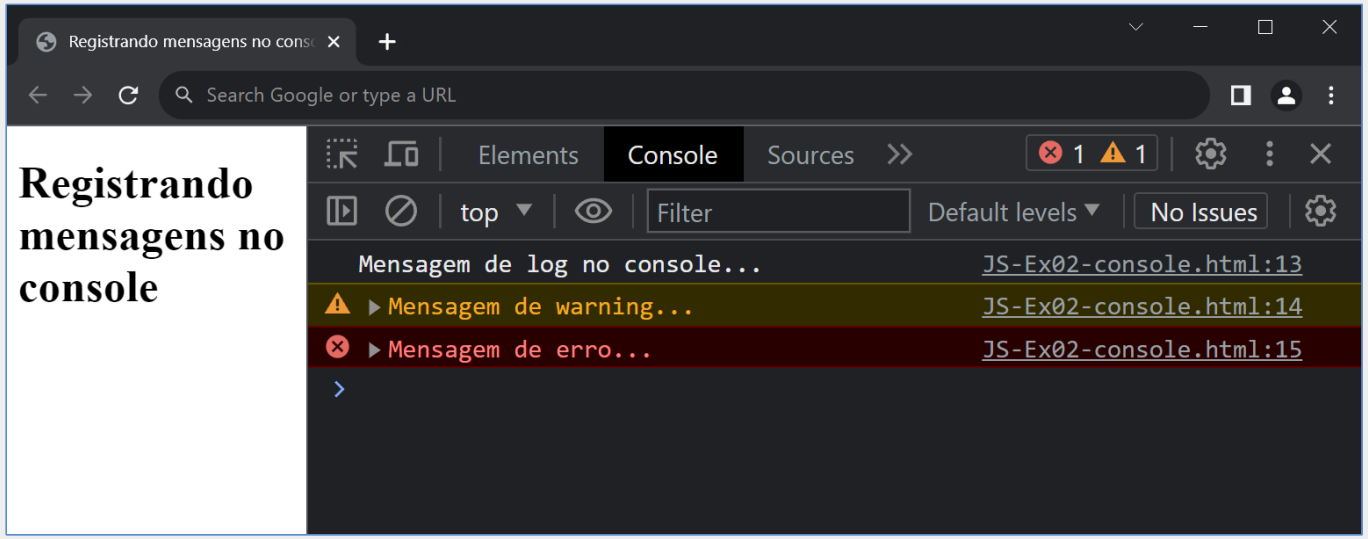
- Representa o documento HTML carregado na aba do navegador
- Possibilita a manipulação da árvore DOM

console (ou window.console)

- Dá acesso à janela de console de depuração do navegador

Registrando Mensagens na Janela de Console do Navegador

```html
<!DOCTYPE html>
<html lang="pt-BR">

<head>
  <meta charset="UTF-8">
  <title>Registrando mensagens no console</title>
</head>

<body>
  <h1>Registrando mensagens no console</h1>

  <script>
    console.log('Mensagem de log no console...');
    console.warn('Mensagem de warning...');
    console.error('Mensagem de erro...');
  </script>
</body>

</html>
```



Registrar mensagens na janela de console do navegador pode ser útil para fins de desenvolvimento, depuração e manutenção da página web.
Para visualizar a janela de console do navegador, tecle **F12** e clique na aba **Console**.

Strings

- Podem ser definidas com aspas simples ou duplas

```
let msg = "JavaScript";
```

- Strings são objetos com propriedades e métodos

```
msg.length, msg.indexOf('alright'), msg.substr(0,2), msg.split etc.
```

- Caracteres podem ser acessados por colchetes ou pelo método chatAt

```
let primLetra = msg[0];
let primLetra = msg.charAt(0);
```

- Strings com aspas duplas podem conter aspas simples e vice-versa

```
let msg = "It's alright";
```

- São imutáveis (caracteres da string não podem ser alterados)

Template Literals (ou Template Strings)

- São strings definidas com o caractere crase: `minha string`
- Suporta fácil interpolação de variáveis e expressões com ${ }
- Maior facilidade para definir strings de múltiplas linhas
- A string pode conter aspas simples ou duplas

```
let a = 1;
let b = 2;
let c = 3;

const delta = b*b - 4*a*c;

console.log(`o discriminante da equacao com
coeficientes ${a}, ${b} e ${c} é ${delta}`);
```

Mostrará na janela de console a mensagem formatada:
"o discriminante da equação com coeficientes 1, 2 e 3 é -8".

Arrays

- Em JavaScript, os arrays não são tipos de dados primitivos
- São tratados como objetos, com propriedades e métodos
- Podem armazenar valores de tipos diferentes
- Os elementos são acessados por índices numéricos
  - Não podem ser acessados por chaves
  - Não são do tipo associativo

Arrays

- Os elementos são colocados entre colchetes, separados por vírgula

```
let pares = [2, 4, 6, 8];
let primeiroPar = pares[0]; // 1º elemento
let nroElementos = pares.length; // tamanho do array
pares[3] = 10; // altera o 4ª elemento do array para 10
```

- É possível ter elementos de tipos diferentes

```
let arrayMisto = [2, 'A', true];
```

- O array pode ser iniciado com vazio

```
let pares = [];
```

Arrays – Outros Métodos

```
let vogais = ['E', 'I', 'O'];

vogais.push('U')    // adiciona um item no final do array

vogais.pop()        // remove e retorna o último item do array

vogais.unshift('A') // adiciona um item no início do array

vogais.shift()      // remove e retorna o primeiro item do array
```

Conteúdo Complementar

Percorrendo Array com Estrutura *for*

```
let pares = [2, 4, 6, 8];
for (let i = 0; i < pares.length; i++) {
    console.log(pares[i]);
};
```

```
let pares = [2, 4, 6, 8];
for (let item of pares) {
    console.log(item);
};
```

```
let pares = [2, 4, 6, 8];
for (let i in pares) {
    console.log(pares[i]);
};
```

Três formas diferentes de percorrer e mostrar os elementos de um array utilizando a estrutura for

Métodos forEach e map

```
// cria e mostra os elementos do array
let pares = [2, 4, 6, 8];
pares.forEach(function (elemento) {
     console.log(elemento);
});

// soma os elementos do array
let soma = 0;
pares.forEach(function (elemento) {
    soma += elemento;
});

// calcula o novo array paresDobro, que
// terá os valores [4, 8, 12, 16]
let paresDobro = pares.map(function (elemento) {
     return elemento * 2;
});
```

Arrays em JavaScript possuem o método **forEach**, que permite executar uma função, passada como parâmetro, para cada elemento do array. forEach sempre retorna undefined.

O método **map**, por outro lado, permite produzir um novo array após aplicação de uma função em cada elemento do array existente.

Conteúdo Complementar

Objeto Simples *(plain object, POJO)*

- Contém apenas dados
- Pode ser definido utilizando chaves { }
- Possui lista de pares do tipo propriedade : valor
- Criado como instância da classe Object

```
let carro = {
    modelo: "Fusca",
    ano: 1970,
    cor: "bege",
    "motor-hp": 65
}

console.log(carro.ano);          // 1970
console.log(carro["motor-hp"]); // 65
```

A propriedade "motor-hp", por ter um caractere especial, **não pode** ser acessada utilizando a notação com o ponto (carro.motor-hp). Para esses casos deve-se utilizar a notação com colchetes.

Declaração de Funções

```
function nomeDaFuncao(par1, par2, par3, ...) {
    // operações
    // operações
    // operações
}
```

```
function max(a, b) {
   if (a > b)
      return a;
   else
      return b;
}
```

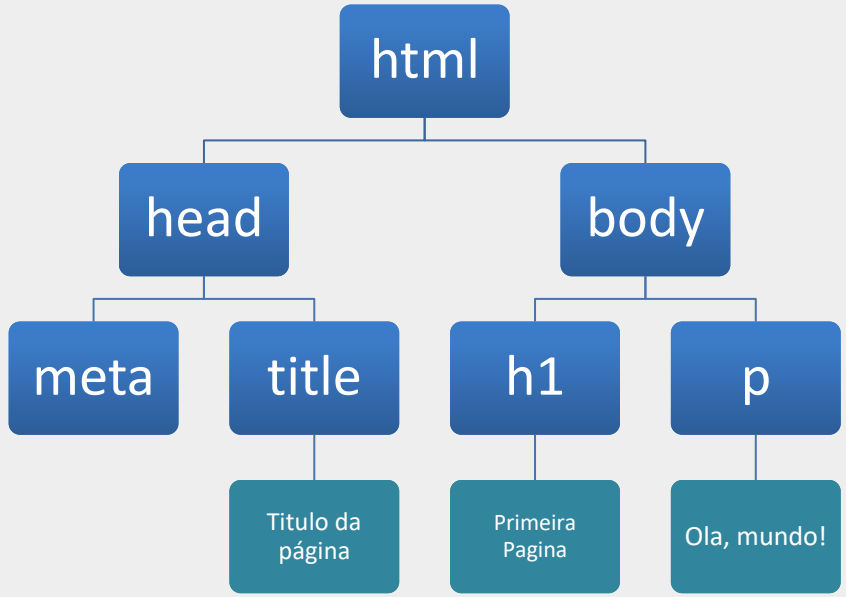
```
let maior = max(2, 5);
```

Quando **'return'** não é utilizada, o valor **undefined** é automaticamente retornado

Manipulação da Árvore DOM

Document Object Model - DOM

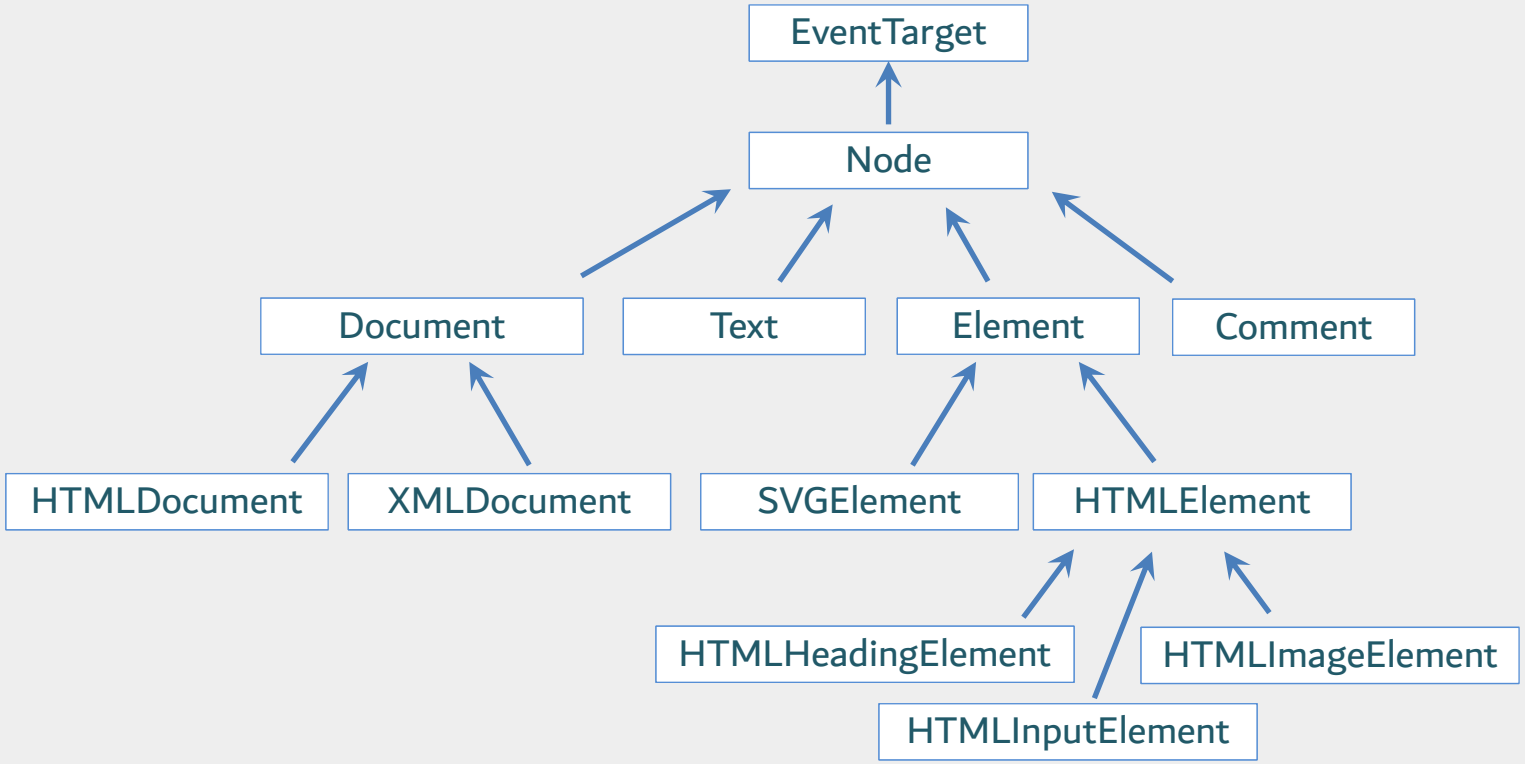
```
<!DOCTYPE html>
<html lang="pt-BR">

  <head>
    <meta charset="UTF-8">
    <title>Titulo da Pagina</title>
  </head>

  <body>
    <h1>Primeira Pagina</h1>
    <p>Ola, mundo!</p>
  </body>

</html>
```



Abstração da Árvore DOM correspondente

Nota: Ao carregar uma página, o navegador percorre o respectivo código HTML e monta uma estrutura de dados internamente denominada **árvore DOM**, que é uma representação em memória de toda a estrutura do documento HTML. Nessa estrutura, cada elemento, comentário ou texto do documento HTML é representado como um objeto, denominado **nó**. A árvore DOM permite a manipulação do documento HTML dinamicamente, utilizando programação, com a **DOM API** e a JavaScript.

Tipos de Objetos na Árvore DOM – Exemplo Simplificado



EventTarget: interface que permite aos nós da árvore DOM suportarem eventos. Inclui definições como addEventListener.

Node: classe abstrata com definições de propriedades e métodos comuns a todos os tipos de nós. Inclui propriedades como parentNode e childNodes. Herda as definições de EventTarget.

Element: interface com funcionalidades a nível de elemento. Inclui propriedades como nextElementSibling e children. Herda as definições de Node.

HTMLInputElement: fornece propriedade e métodos adicionais para elementos do tipo input, como value, pattern, required etc. Herda as definições de HTMLElement.

Document: interface que representa o documento HTML carregado na aba do navegador. Serve de ponto de entrada para a árvore DOM. Contém definições a nível de documento como getElementById e createElement.

Os objetos que compõem a árvore DOM não são todos do mesmo tipo. Há uma hierarquia de classes e interfaces que definem os tipos dos objetos. Por exemplo, um texto dentro de um elemento <h1> é representado na árvore como um nó (objeto) do tipo Text. O <h1> em si é representado como um nó do tipo Element (mais especificamente, HTMLHeadingElement). Um elemento <input> terá um nó correspondente na árvore do tipo HTMLInputElement. Esse nó, além de possuir propriedades e métodos comuns a todos os elementos (herdados de Element), terá também propriedades específicas do elemento input como value e required. Até mesmo os comentários no documento HTML são representados na árvore DOM como nós do tipo Comment.

Busca na Árvore DOM

```
document.querySelector("seletor CSS")
```

- Faz uma busca na árvore DOM utilizando uma string de seleção CSS e retorna o primeiro nó da árvore do tipo Element que atende à string de seleção
- Retorna `null` caso não haja correspondências
- Nenhum nó é retornado caso o seletor inclua pseudo-elementos

OBS: o método querySelector é definido no objeto document e também nos nós do tipo Element. Portanto, é possível fazer uma busca em toda a árvore DOM ou apenas em um ramo da árvore que inicia a partir de um nó de elemento.

document.querySelector – Exemplo

```
<body>
  <h1>Primeiro Título</h1>
  <h1>Segundo Título</h1>
  <script>
    const firstNodeH1 = document.querySelector("h1");
    alert(firstNodeH1.textContent); // mostra 'Primeiro Título'
    firstNodeH1.textContent = "Título alterado com JavaScript";
  </script>
</body>
```

Neste exemplo, o método querySelector retorna uma referência para o nó da árvore DOM correspondente ao primeiro título h1 da página. A propriedade textContent do nó permite acessar e alterar o conteúdo do respectivo elemento HTML.
Antes do conteúdo da página ser apresentado, será mostrada a mensagem "Primeiro título" devido à chamada da função alert. Quando o usuário clicar em Ok, o código JavaScript prosseguirá e fará a alteração do conteúdo do primeiro título h1 para "Título alterado com JavaScript". Portanto, a página será apresentada ao usuário já com o título alterado.

document.querySelector – Exemplos Adicionais

Retorna o nó correspondente ao elemento com id='imagemLogo'

```
const nodeImgLogo = document.querySelector("#imagemLogo");
```

Retorna o nó correspondente ao primeiro 'li' filho da primeira 'ul'

```
const nodeLi = document.querySelector("ul > li");
```

Retorna o nó correspondente ao primeiro 'input' com name="sexo" selecionado

```
const nodeRadio = document.querySelector('input[name="sexo"]:checked');
```

Tipos de Objetos na Árvore DOM - Exemplo

```html
<body>

  <h1>Tipos de nós na DOM Tree</h1>
  <input type="text" id="myInput">
  <!-- comentário -->
  <pre id="output"></pre>

  <script>
    const output = document.querySelector("#output");
    for (let node of document.body.childNodes)
      output.textContent += node.constructor.prototype + '\n';

    const input = document.querySelector("#myInput");
    output.textContent += (input instanceof HTMLInputElement) + '\n';
    output.textContent += (input instanceof HTMLElement) + '\n';
    output.textContent += (input instanceof Element) + '\n';
    output.textContent += (input instanceof Node) + '\n';
    output.textContent += (input instanceof EventTarget) + '\n';

    output.textContent += document.constructor.prototype + '\n';
  </script>

</body>
```

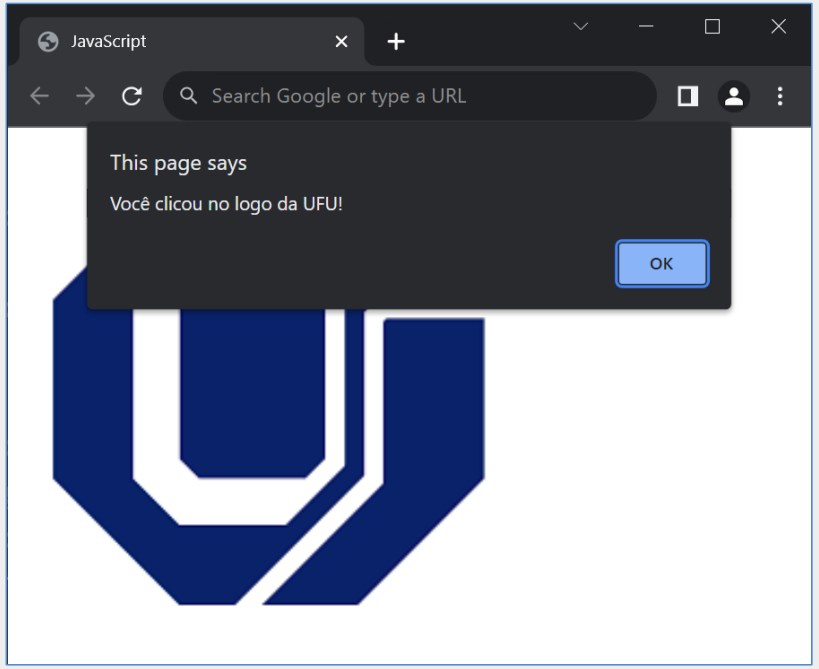
Tratamento de Eventos

- JavaScript é baseada em eventos
- É possível executar funções na ocorrência de eventos como “clique em botão”, “seleção de item”, “rolagem da página” etc.
- As funções para tratar os eventos podem ser registradas de duas formas:
  - Utilizando propriedades de eventos dos nós
  - Ou utilizando o método addEventListener

Tratando Eventos com Propriedades de Eventos

- Propriedades de eventos são propriedades dos objetos que permitem a indicação de uma função para tratar os eventos nos objetos
- Essas propriedades começam com “on” seguido do nome do evento, ou seja, onclick, onmouseover, onkeyup, onsubmit etc.
- Basta atribuir o nome da função à propriedade (sem parênteses). A função será chamada automaticamente quando ocorrer o evento no objeto

```
<body>
  <img src="images/logoUFU.png" id="imgUFU">
  <script>
    // Função para tratar o evento clique na 1ª imagem
    function trataClique() {
      alert("Você clicou no logo da UFU!");
    }

    // Registra a função trataClique para tratar o evento 'click' na imagem.
    // Registro feito com a propriedade de evento onclick do nó.
    const nodeImage = document.querySelector("#imgUFU");
    nodeImage.onclick = trataClique;
  </script>
</body>
```



Tratando Eventos com addEventListener

- Outra forma de registrar funções para tratar eventos é por meio do método addEventListener do objeto
- Esse método tem dois parâmetros principais:
  - O primeiro é o nome do evento (não tem o “on”)
  - O segundo é a função para tratar o evento
  - Ex.: object.addEventListener("click", trataClique);

```
<body>
  <img src="images/logoUFU.png" id="imgUFU">
  <script>
    // Função para tratar o evento clique na imagem
    function trataClique() {
      alert("Você clicou no logo da UFU!");
    }

    // Registra a função trataClique para tratar o evento 'click' na imagem.
    // Registro feito com o método addEventListener do nó.
    const nodeImage = document.querySelector("#imgUFU");
    nodeImage.addEventListener('click', trataClique);
  </script>
</body>
```

Código análogo ao do slide anterior, porém utilizando o método addEventListerner, ao invés da propriedade de evento do objeto.

OBS: repare que não há parênteses () depois de trataClique na linha do addEventListener, pois **não estamos chamando** a função, mas apenas **registrando** trataClique como a função a ser chamada quando o evento ocorrer.

addEventListener tem ainda um 3º parâmetro opcional não abordado neste material.

Tratando Eventos com addEventListener

- Com addEventListener é possível registrar múltiplas funções para tratar o evento no objeto
- Quando o evento ocorrer, as funções serão chamadas na ordem em que foram registradas

```
<body>
  <img src="images/logoUFU.png" id="imgUFU">
  <script>
    function funcao1() {
      alert("Você clicou no logo da UFU!");
    }
    function funcao2() {
      alert("Obrigado!");
    }
    const nodeImage = document.querySelector("#imgUFU");
    nodeImage.addEventListener('click', funcao1);
    nodeImage.addEventListener('click', funcao2);
  </script>
</body>
```

Quando o usuário clicar na imagem aparecerá primeiramente a mensagem "Você clicou...". Posteriormente, aparecerá "Obrigado!".

Funções Anônimas

- Funções tratadoras de eventos podem ser definidas no momento em que são indicadas para tratar o evento
- Isto é possível utilizando funções anônimas, como apresentado no exemplo a seguir

```
<body>
  <img src="images/logoUFU.png" id="imgUFU">
  <script>
    // Função anônima indicada para tratar evento
    // ao mesmo tempo em que é definida
    const nodeImage = document.querySelector("#imgUFU");
    nodeImage.addEventListener('click', function () {
      alert("Você clicou no logo da UFU!");
    });
  </script>
</body>
```

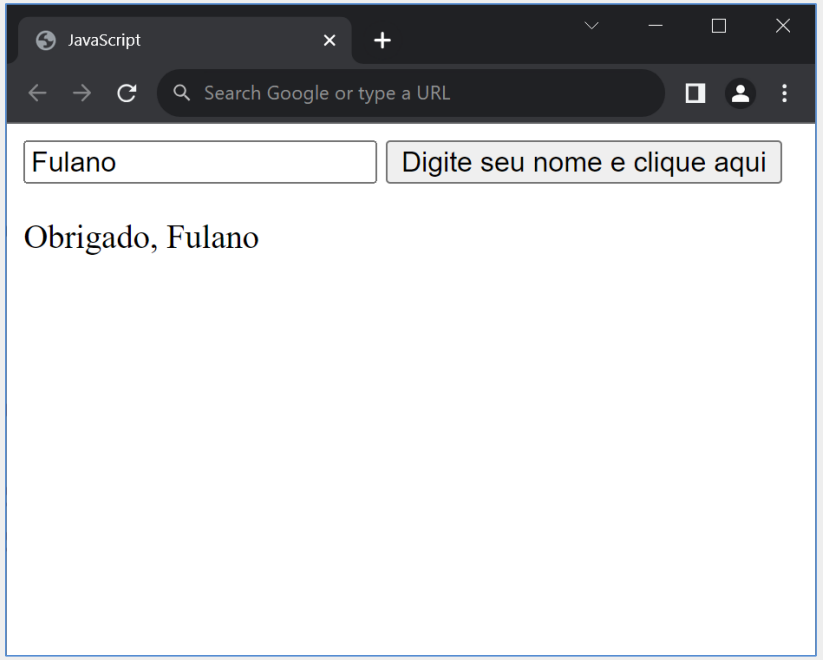
Funções Anônimas

```
<body>
  <img src="images/logoUFU.png" id="imgUFU">
  <script>
    // Função anônima indicada para tratar evento
    // ao mesmo tempo em que é definida
    const nodeImage = document.querySelector("#imgUFU");
    nodeImage.onclick = function () {
      alert("Você clicou no logo da UFU!");
    };
  </script>
</body>
```

Função anônima sendo utilizada em conjunto com propriedade de evento

Tratando Eventos – Exemplo Adicional

```
<body>
  <input type="text" name="nome">
  <button type="button">Digite seu nome e clique aqui</button>
  <p></p>

  <script>
    const botao = document.querySelector("button");
    botao.onclick = function () {
      const campoNome = document.querySelector("input");
      const parSaida =  document.querySelector("p");
      parSaida.textContent = 'Obrigado, ' + campoNome.value;
    };
  </script>
</body>
```

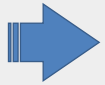


A propriedade value do objeto na árvore DOM correspondente ao elemento <input> permite acessar o valor do campo, ou seja, o texto preenchido pelo usuário.

Arrow Function =>

- Permite definir funções sem utilizar a palavra function
- A definição é feita de forma abreviada utilizando os caracteres '=>'

```
objeto.onclick = function () {
      // operações
}
```



```
objeto.onclick = () => {
      // operações
}
```

Função anônima definida com a palavra function

Declaração correspondente utilizando arrow function

OBS: arrow function não substitui a definição tradicional de funções em todas as situações.

Arrow Function – Exemplos Adicionais

- Função com uma única declaração dispensa as chaves

```
objeto.onclick = () => alert('Você clicou em...');
```

- Arrow function também pode ter parâmetros

```
objeto.onclick = (e) => alert('Objeto clicado: ' + e.target);
```

- Arrow function com um único parâmetro não precisa dos parênteses

```
objeto.onclick = e => alert('Objeto clicado: ' + e.target);
```

O parâmetro `e` utilizando nestes exemplos é um objeto contendo informações do evento disparado. Por exemplo, `e.target` contém uma referência para o objeto onde o evento foi disparado (no caso acima, o objeto na árvore DOM correspondente ao botão, imagem etc., que recebeu o click do usuário).

Percorrendo Array com forEach e Arrow Function

```
let pares = [2, 4, 6, 8];
let soma = 0;
// soma os elementos
pares.forEach( elemento => soma += elemento );

// mostra os elementos na janela de console
pares.forEach( elemento => console.log(elemento) );
```

Outros Eventos Comuns

Evento	Propriedade	Quando Ocorre o Evento
mouseenter	onmouseenter	Quando o cursor do mouse entra na região do elemento
mouseleave	onmouseleave	Quando o cursor sai da região do elemento
keydown/keyup	onkeydown/onkeyup	Quando o usuário pressiona/libera alguma tecla
focus	onfocus	Quando o elemento recebe o foco (clique em campo do formulário)
blur	onblur	Quando o elemento perde o foco (clique fora do campo)
change	onchange	Quando um campo de formulário tem o valor alterado (em campos textuais é disparado apenas na perda do foco)
submit	onsubmit	Quando o formulário é submetido
load	onload	Quando a página termina de ser carregada por completo
DOMContentLoaded	--	Quando a árvore DOM termina de ser carregada

Eventos load vs DOMContentLoaded

load

- Ocorre quando a página termina de ser carregada por **completo**
- Só ocorre depois que o navegador termina de baixar as imagens, arquivos CSS etc., da página web

DOMContentLoaded

- Ocorre quando o documento é carregado e a árvore DOM termina de ser montada
- Não aguarda pelo carregamento de imagens, arquivos CSS etc.
- Geralmente ocorre antes do evento load

Onde está o erro?

Arquivo HTML

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Página com erro</title>
  <script src="meuscript.js"></script>
</head>
<body>
  <h1>Hello!</h1>
  <button>Clique aqui!</button>
</body>
</html>
```

Arquivo meuscript.js

```
1
2  const botao = document.querySelector("button");
3  botao.onclick = () => alert("Obrigado!");
4
```

Onde está o erro?

Arquivo HTML

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Página com erro</title>
  <script src="meuscript.js"></script>
</head>
<body>
  <h1>Hello!</h1>
  <button>Clique aqui!</button>
</body>
</html>
```

Arquivo meuscript.js

```
1
2  const botao = document.querySelector("button");
3  botao.onclick = () => alert("Obrigado!");
4
```

Quando o navegador executar o código JavaScript, a árvore DOM estará incompleta e não possuirá o nó correspondente ao elemento button. Dessa forma, o método querySelector retornará null. Isso acontece porque o arquivo JavaScript está sendo referenciado no início do arquivo HTML, dentro da região de cabeçalho, e será processado antes do HTML restante ser carregado.

Solução utilizando DOMContentLoaded

Arquivo HTML

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Página com erro</title>
  <script src="meuscript.js"></script>
</head>
<body>
  <h1>Hello!</h1>
  <button>Clique aqui!</button>
</body>
</html>
```

Arquivo meuscript.js

```js
document.addEventListener('DOMContentLoaded', () => {
  const botao = document.querySelector("button");
  botao.onclick = () => alert("Obrigado!");
});
```

Uma solução é utilizar o evento DOMContentLoaded para garantir que o código seja executado apenas depois que a árvore DOM for completamente carregada.

Outra possibilidade é utilizar o atributo defer na tag <script>, para dizer ao navegador que o script deve ser executado depois que o documento HTML terminar de ser carregado (<script src="meuscript.js" defer></script>)

Busca na Árvore DOM com querySelectorAll

`document.querySelectorAll`

- Aceita uma string de seleção CSS como parâmetro
- Retorna uma lista com **todos** os nós da árvore DOM que atendem à seleção
- Ou retorna `null` caso não haja correspondências

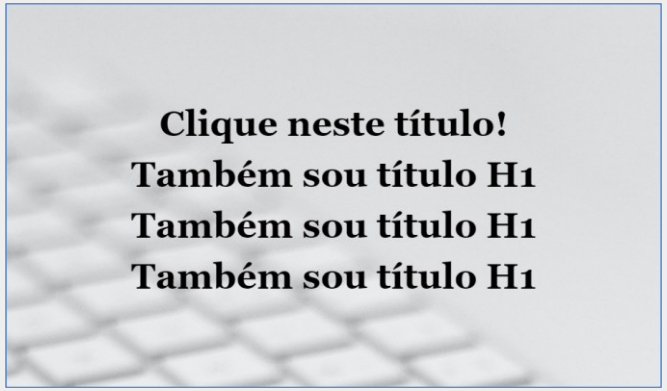
Busca na Árvore DOM com querySelectorAll

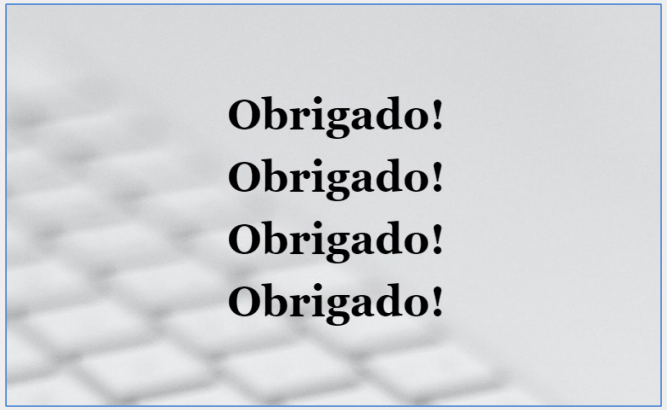
```
...
<body>
  <h1>Título 1</h1>
  <h1>Título 2</h1>
  <h1>Título 3</h1>
</body>
...
```

```
// retorna os nós correspondentes a todos os elementos h1 da página
const nodesH1 = document.querySelectorAll("h1");

// mostra "Título 1", "Título 2" e "Título 3"
for (let node of nodesH1) {
    console.log(node.textContent);
}
```

Manipulação da Árvore DOM – Exemplo

```
<main>
  <h1>Clique neste título!</h1>
  <h1>Também sou título H1</h1>
  <h1>Também sou título H1</h1>
  <h1>Também sou título H1</h1>
</main>
<script>
  const nodeH1 = document.querySelector("h1");
  nodeH1.addEventListener("click", function () {
    const nodesH1 = document.querySelectorAll("h1");
    for (let node of nodesH1)
      node.textContent = "Obrigado!";
  });
</script>
```





Neste exemplo, quando o usuário clicar no **primeiro** título <h1>, **todos** os títulos <h1> terão seu conteúdo alterado para "Obrigado!"

Manipulação da Árvore DOM – Exemplo

```
<main>
  <h1>Clique neste título!</h1>
  <h1>Clique neste título!</h1>
  <h1>Clique neste título!</h1>
  <h1>Clique neste título!</h1>
</main>
<script>
  const nodesH1 = document.querySelectorAll("h1");
  for (let node of nodesH1)
    node.onclick = () => node.textContent = "Obrigado!";
</script>
```

Neste exemplo, quando o usuário clicar em **qualquer** título <h1>, seu **respectivo** texto será alterado para "Obrigado!". Observe que uma função tratadora de evento é registrada para cada nó, que fará a modificação do texto do próprio nó.

Manipulação da Árvore DOM – Exemplo

```
<main>
  <h1>Clique neste título!</h1>
  <h1>Clique neste título!</h1>
  <h1>Clique neste título!</h1>
</main>
<script>
  const nodesH1 = document.querySelectorAll("h1");
  for (let node of nodesH1)
    node.onclick = alteraConteudo;

  function alteraConteudo(e) {
    e.target.textContent = "Obrigado!";
  }
</script>
```

Este exemplo é equivalente ao anterior, porém com a definição de uma função padrão no lugar da *arrow function*.

Repare que a função tem um parâmetro de nome e, que receberá o objeto representando o evento.

e.target permite acessar o objeto no qual o evento foi disparado (nó <h1> clicado).

Detalhes do Evento

- Além da propriedade target do objeto do evento, há também várias outras propriedades específicas para cada tipo de evento
- Por exemplo, para um evento de click, há também:
  - e.screenX - coordenada x (horizontal) do clique na **tela**
  - e.screenY - coordenada y (vertical) do clique na **tela**
  - e.clientX - coordenada x do clique na **viewport** (janela do navegador)
  - e.clientY - coordenada y do clique na **viewport** (janela do navegador)
  - e.ctrlKey - true ou false indicando se a tecla **ctrl** foi pressionada junto com o click
  - e.shiftKey - true ou false indicando se a tecla **shift** foi pressionada com o click
- Para um evento de teclado, há outras propriedades como:
  - e.key - string correspondente à tecla pressionada (ex.: "Enter", "a", "b" etc.)

Outras Formas de Realizar Buscas na Árvore DOM

document.getElementById

- busca um nó de elemento utilizando o seu id

document.getElementsByName

- busca nós de elementos pelo valor do atributo name do elemento

document.getElementsByTagName

- busca nós de elementos pelo nome da tag HTML, como img, h1, etc.

document.getElementsByClassName

- busca nós de elementos pelo valor do atributo class do elemento

Conteúdo Complementar

Exemplo de document.getElementsByName

```
...
<input type="radio" name="estadoCivil" value="solteiro">
<input type="radio" name="estadoCivil" value="casado">
<input type="radio" name="estadoCivil" value="divorciado">
...
<script>
   const radiosEstCiv = document.getElementsByName("estadoCivil");
   for (let radio of radiosEstCiv) {
      if (radio.checked)
         alert(radio.value);
   }
</script>
```

Busca correspondente utilizando *querySelectorAll*

```
document.querySelectorAll('input[name="estadoCivil"]');
```

Conteúdo Complementar

Exemplo de document.getElementsByTagName

```
...
<input type="radio" name="estadoCivil" value="solteiro">
<input type="radio" name="estadoCivil" value="casado">
<input type="text" name="bairro">
<input type="text" name="cidade">
...
<script>
   const listaDeInputs = document.getElementsByTagName("input");
   for (let input of listaDeInputs) {
      alert(input.name);
   }
</script>
```

Busca correspondente utilizando *querySelectorAll*

```
document.querySelectorAll("input");
```

Conteúdo Complementar

Exemplo de document.getElementsByClassName

```
...
<ul class="nav"> ... </ul>
...
<script>
   const primListaNav = document.getElementsByClassName("nav")[0];
</script>
```

Busca correspondente utilizando *querySelector*

```
document.querySelector(".nav");
```

Conteúdo Complementar

Acesso ao Conteúdo dos Elementos

Propriedade textContent

- Se o elemento possui conteúdo textual, retorna esse texto acrescido da **concatenação** do textContent de eventuais elementos filhos
- Uma alteração do valor removerá eventuais nós filhos e **substituirá** pelo novo texto

Propriedade innerText

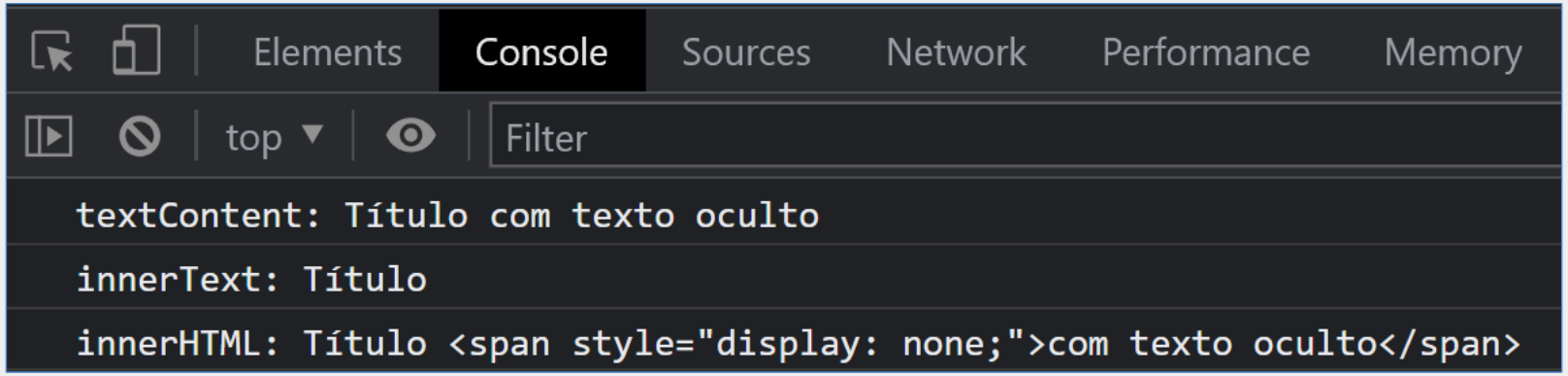
- Semelhante a textContent, porém não inclui conteúdos de elementos que "não podem ser lidos" pelo usuário, como conteúdo ocultado com CSS, conteúdo de tags como <script>, <style> etc.

Propriedade innerHTML

- Retorna **todo** o conteúdo do elemento, **incluindo as tags** HTML
- Quando alterada, o novo conteúdo é avaliado pelo navegador e caso tenha tags HTML haverá a criação de novos nós correspondentes na árvore DOM
- **OBS**: possibilidade de ataques XSS e desempenho inferior ao de textContent.

Acessando o Conteúdo com textContent, innerText e innerHTML

```
<h1>Título <span style="display: none;">com texto oculto</span></h1>
<script>
  const nodeH1 = document.querySelector("h1");
  console.log("textContent: " + nodeH1.textContent);
  console.log("innerText: " + nodeH1.innerText);
  console.log("innerHTML: " + nodeH1.innerHTML);
</script>
```

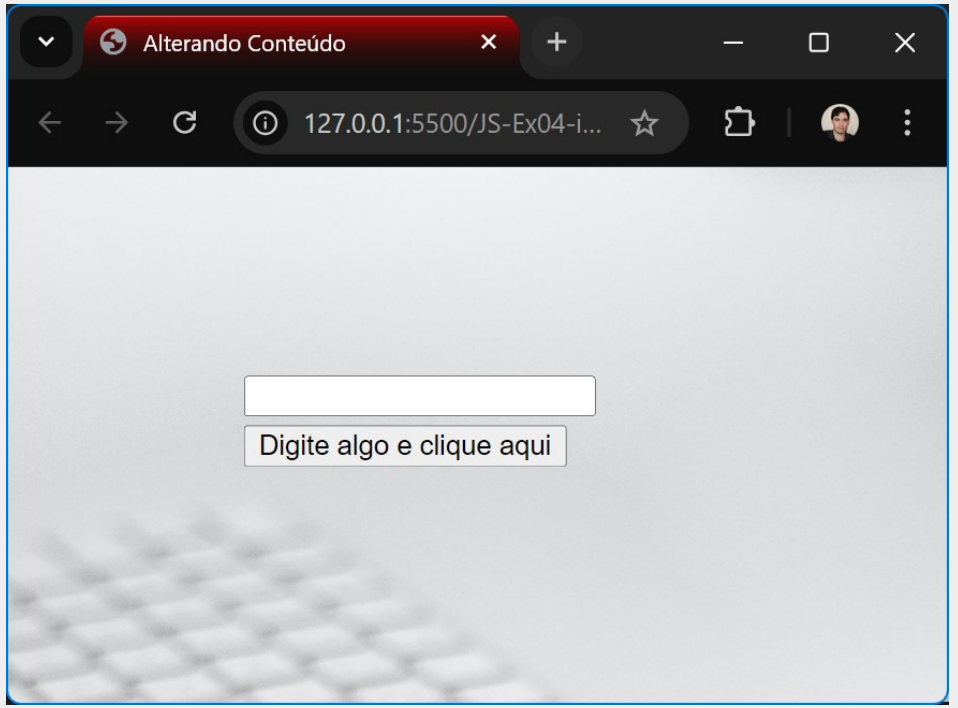


Modificando o Conteúdo com textContent, innerText e innerHTML

```
<main>
  <h2></h2>
  <h2></h2>
  <h2></h2>
</main>
<script>
  const nodes = document.querySelectorAll("h2");
  const conteudo = 'Titulo com <span style="color: red;">texto vermelho</span>';
  nodes[0].textContent = conteudo;
  nodes[1].innerText = conteudo;
  nodes[2].innerHTML = conteudo;
</script>
```

Aspectos de Segurança – Propriedade innerHTML e Ataques XSS

```
<main>
  <input type="text">
  <button>Digite algo e clique aqui</button>
  <p></p>
</main>
<script>
  const botao = document.querySelector("button");
  botao.onclick = function () {
    const valorUsuario = document.querySelector("input").value;
    const pSaida = document.querySelector("p");
    pSaida.innerHTML = valorUsuario;  // vulnerável a XSS!!
  }
</script>
```



Neste exemplo, o texto informado pelo usuário no campo textual será apresentado na página, como conteúdo de um parágrafo, quando o usuário clicar no botão. Porém o código é vulnerável a ataques XSS (*cross-site scripting*) devido ao uso inadequado da propriedade innerHTML, permitindo que um usuário malicioso execute o seu próprio código JavaScript na página. Por exemplo, se o usuário digitar no campo textual a string <img src="inexistente.jpg" onerror="alert('AHA')">, o código JavaScript inserido na propriedade onerror será executado, pois o navegador não encontrará a imagem "inexistente.jpg".

OBS: o código JavaScript alert('AHA') inserido neste exemplo com o atributo HTML onerror não é considerado uma boa prática (por ser do tipo "inline"). O uso aqui tem apenas o objetivo de ilustrar um ataque XSS.

Método Element.insertAdjacentHTML()

- Cria e insere nós na árvore DOM a partir de string contendo código HTML
- Permite indicar a posição de inserção
- Sintaxe: node.insertAdjacentHTML(posicao, stringHtml), onde posicao pode ser "beforebegin", "afterbegin", "beforeend" e "afterend"

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  Parágrafo de exemplo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

OBS: de maneira similar à propriedade **innerHTML**, o método **insertAdjacentHTML** deve ser utilizado com cuidado, pois pode tornar o website vulnerável a ataques XSS. Se o conteúdo HTML sendo inserido tem dados potencialmente produzidos pelo usuário, então o método não deve ser utilizado.

Método Element.insertAdjacentHTML()

```html
<body>
  <h1>Exemplo do Método insertAdjacentHTML</h1>
  <p>Lorem ipsum dolor sit amet.</p>
  <button>Carregar mais conteúdo</button>
</body>

<script>
  const button = document.querySelector("button");
  button.onclick = function () {
    let novoConteudo = carregaMaisConteudo();
    document.body.insertAdjacentHTML("beforeend", novoConteudo);
  }
</script>
```

Suponha que a função carregaMaisConteudo() tenha sido definida para buscar conteúdo HTML adicional no servidor. O conteúdo retornado pela função seria inserido dinamicamente na página, antes da tag </body>, assim que o usuário clicasse no botão.
OBS: código vulnerável a XSS, caso o novo conteúdo contenha dados produzidos pelo usuário.

Conteúdo Complementar

Manipulando Atributos

Para a maioria dos atributos HTML (como name, value, id, src, etc.) há uma propriedade de **mesmo nome** no objeto correspondente da árvore DOM

```
...
<input type="text" id="aabb" name="ccdd" value="rua abc">
...
<script>
    const campoRua = document.querySelector("input");
    console.log( campoRua.id );      // mostra 'aabb'
    console.log( campoRua.name );    // mostra 'ccdd'
    console.log( campoRua.value );   // mostra 'rua abc'
    campoRua.name = "novo valor";    // alt. o val. do atrib. 'name'
</script>
```

Manipulando Atributos

Porém alguns atributos são acessados de forma diferenciada

Atributo HTML	JavaScript
`for`	`node.htmlFor`
`class`	`node.className`
`data-matricula`	`node.dataset.matricula`
`data-num-matricula`	`node.dataset.numMatricula`
	`node.dataset["numMatricula"]`

Manipulando Atributos - Exemplo

```
<body>
  <main>
    <img src="images/logoUFU.png">
  </main>
  <script>
    const nodeImage = document.querySelector("img");
    nodeImage.onclick = function () {
      nodeImage.src = "images/logoFacom.png";
      nodeImage.alt = "Faculdade de Computação"
    }
  </script>
</body>
```

Quando o usuário clicar sobre a imagem ela será trocada por outra.

Manipulando Atributos para Alterar Estilos CSS

- Uma forma de alterar os estilos CSS de um elemento é por meio do atributo `style` do respectivo nó na árvore DOM
- Os nomes das propriedades segue o padrão **CamelCase**

CSS	JavaScript
`color`	`node.style.color`
`font-family`	`node.style.fontFamily`
`background-color`	`node.style.backgroundColor`

Ao utilizar a propriedade style do nó, a alteração do CSS ocorre de forma "inline".

Manipulando Atributos para Alterar Estilos CSS

```
<main>
  <h1>Título</h1>
  <h1>Título</h1>
  <h1>Título</h1>
</main>
<script>
  const nodesH1 = document.querySelectorAll("h1");
  for (let node of nodesH1) {
    node.onclick = () => node.style.visibility = 'hidden';
  }
</script>
```

Neste exemplo, o clique em um título fará com que ele fique oculto.
OBS: algumas propriedades CSS cujos valores são calculados dinamicamente pelo navegador (ex. margin, padding, width etc.) não podem ter o valor corrente resgatado utilizando node.style.nomePropriedade. Nesses casos pode ser necessário utilizar a função getComputedStyle.

Manipulando Atributos para Alterar Estilos CSS

- Uma forma melhor de alterar os estilos dos elementos dinamicamente é adicionando ou removendo classes CSS
- Pode-se utilizar:
  - node.classList.add("classeCSS") para adicionar uma classe CSS ao elemento, ou
  - node.classList.remove("classeCSS") para remover uma classe CSS do elemento

OBS1: se a classe sendo adicionada já estiver presente, nada acontecerá.
OBS2: Se a classe sendo removida não constar na lista de classes do elemento, nada acontecerá (e nenhum erro será produzido).

Manipulando Atributos para Alterar Estilos CSS

```
<head>
  <style>
    .destaca {
      box-shadow: 0 0 20px red;
    }
  </style>
</head>
<body>
  <h1>Passe o cursor sobre a imagem</h1>
  <img src="images/logoUFU.png">
  <script>
    const img = document.querySelector("img");
    img.onmouseenter = () => img.classList.add("destaca");
    img.onmouseleave = () => img.classList.remove("destaca");
  </script>
</body>
```

Neste exemplo, aparecerá a sombra vermelha na imagem quando o usuário posicionar o ponteiro do mouse sobre ela. A sombra desaparecerá quando o ponteiro for afastado. Repare que a classe CSS .destaca é adicionada e removida dinamicamente com JavaScript.

Manipulando Atributos

node.getAttribute

- Permite acessar o valor do atributo conforme aparece na HTML (string)
- Em alguns casos retorna um valor igual à respectiva propriedade
- Há casos em que retorna um valor diferente da propriedade
- Atributos não padronizados **devem** ser acessados com `getAttribute`
  - Propriedades não são criadas para atributos não padronizados
  - Atributos criados pelo usuário são exemplos de atributos não padronizados

Conteúdo Complementar

Manipulando Atributos

node.getAttribute

```
...
<input type="radio" checked>
...
<script>
    const campo = document.querySelector("input");
    const a = campo.checked;                    // retorna true
    const b = campo.getAttribute("checked");    // retorna ""
</script>
```

Manipulando Atributos

node.getAttribute

```
...
<h1 style="color: blue">Título Qualquer</h1>
...
<script>
   const titulo = document.querySelector("h1");
   alert(titulo.style);        // Mostra [object CSSStyleDeclaration]
   alert(titulo.style.color);          // Mostra blue
   alert(titulo.getAttribute("style")); // Mostra 'color: blue'
</script>
```

Manipulando Atributos

`node.setAttribute`

- Define o valor de um atributo
- Se o atributo existe, atualiza o valor
- Caso contrário, cria um novo atributo com o respectivo valor

Conteúdo Complementar

Manipulando Atributos

node.setAttribute

```
...
<h1 id="tituloTeste1">Título Qualquer</h1>
...
<script>
    const titulo = document.querySelector("h1");
    titulo.setAttribute("id", "novoIdDoTitulo");
</script>
```

Acessando o Primeiro e o Último Nó Filho

node.firstChild

- retorna o primeiro nó filho do elemento
- abrange nós do tipo **texto**, **comentário** e **elemento**

node.firstElementChild

- retorna o primeiro nó filho do **tipo elemento**

node.lastChild

- retorna o último nó filho
- abrange nós do tipo **texto**, **comentário** e **elemento**

node.lastElementChild

- retorna o último nó filho do **tipo elemento**

Acessando o Primeiro e o Último Nó Filho

```
<body>

  <h2>Título h2</h2>
  <p>Parágrafo</p>
  <!-- comentário -->

  <script>
    alert(document.body.firstChild); // mostra [Object Text]
    alert(document.body.lastChild); // mostra [Object HTMLScriptElement]

    alert(document.body.firstElementChild); // [Object HTMLHeadingElement]
    alert(document.body.lastElementChild); // [Object HTMLScriptElement]

    alert(document.body.firstChild.textContent); // mostra string vazia
    alert(document.body.firstElementChild.textContent); // mostra "Título h2"
  </script>
</body>
```

Acessando a Lista de Nós Filhos de um Nó

node.childNodes

- retorna uma lista contendo todos os nós filhos do nó
- inclui nós de **texto**, nós de **comentário** e nós de **elemento**

node.children

- retorna lista contendo apenas os nós filhos do **tipo elemento**

Acessando a Lista de Nós Filhos de um Nó

```
<body>

  <h2>Tipos de nós na DOM Tree</h2>
  <p>Teste</p>
  <!-- comentário -->

  <script>
    for (let node of document.body.childNodes)
      alert(node);

    for (let node of document.body.children)
      alert(node);
  </script>
</body>
```

O primeiro for mostra:

- [Object Text]
- [Object HTMLHeadingElement]
- [Object Text]
- [Object HTMLParagraphElement]
- [Object Text]
- [Object Comment]
- [Object Text]
- [Object HTMLScriptElement]

O segundo for mostra apenas:

- [Object HTMLHeadingElement]
- [Object HTMLParagraphElement]
- [Object HTMLScriptElement]

Observe que há um nó do tipo Text na árvore DOM para cada texto "em branco" (contendo quebra de linha e espaços) que antecede os elementos no documento HTML.

Criando e Adicionando Novos Nós na Árvore DOM

document.createElement("nomeDoElementoASerCriado")

- cria um novo nó do tipo **Element**
- Ex.: let novoSpan = document.createElement("span");

node.appendChild(novoNo)

- acrescenta um nó filho no final da lista de filhos

node.removeChild(noFilhoASerRemovido)

- remove um nó filho (parâmetro) da lista de filhos

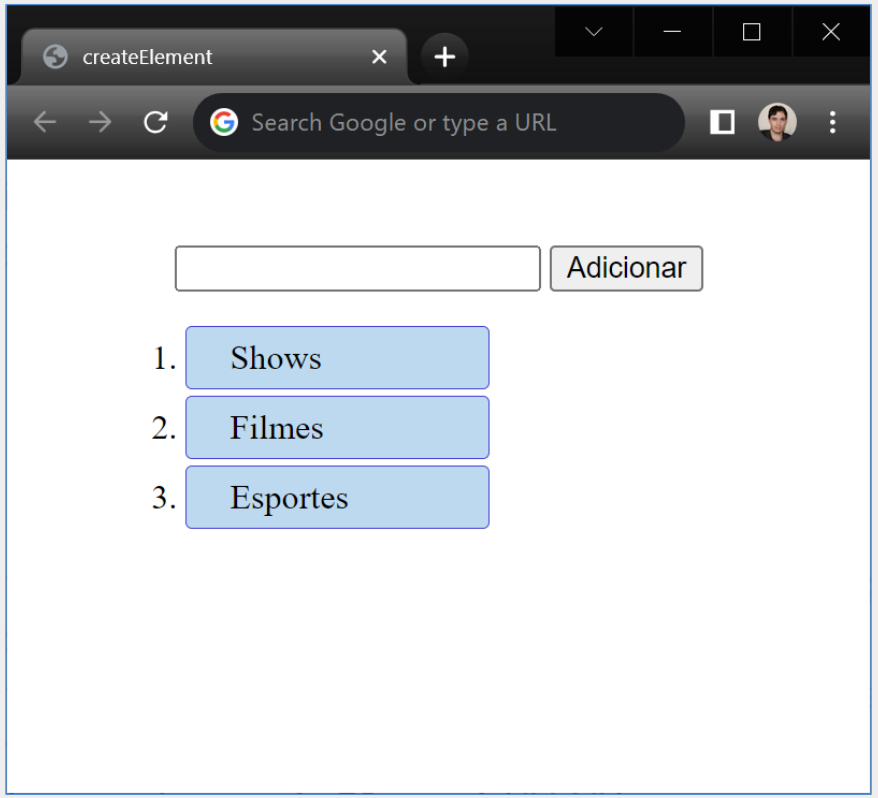
node.remove()

- remove o próprio nó da lista de filhos do nó pai

OBS: para inserir o novo nó em uma posição específica dentro da lista de filhos há o método insertBefore (ex.: *node.insertBefore(novoNo, node.childNodes[2]);*)

Manipulação da Árvore DOM – Exemplo

```
<input type="text">
<button>Adicionar</button>
<ol>
  <li>Shows</li>
  <li>Filmes</li>
</ol>
<script>
  const botaoAdicionar = document.querySelector("button");
  botaoAdicionar.addEventListener("click", function () {
    const campoInteresse = document.querySelector("input");
    const listaInteresses = document.querySelector("ol");

    const novoLi = document.createElement("li");
    novoLi.textContent = campoInteresse.value;

    listaInteresses.appendChild(novoLi);
    campoInteresse.value = '';
  })
</script>
```



Quando o usuário preencher o campo e clicar no botão **Adicionar,** será inserido um novo item na lista com o texto informado pelo usuário.

Manipulação da Árvore DOM – Exemplo

```
<input type="text">
<ol>
  <li>Shows</li>
  <li>Filmes</li>
</ol>
<script>
  const campoInteresse = document.querySelector("input");
  campoInteresse.addEventListener("keyup", e => {
    if (e.key === "Enter") {
      const listaInteresses = document.querySelector("ol");
      const novoLi = document.createElement("li");
      novoLi.textContent = campoInteresse.value;
      listaInteresses.appendChild(novoLi);
      campoInteresse.value = '';
    }
  });
</script>
```

Este exemplo é apenas uma alteração do exemplo anterior para permitir a inserção do novo item na lista quando o usuário pressionar a tecla Enter depois de digitar o novo interesse (não há o botão Adicionar).

Manipulação da Árvore DOM – Exemplo

```
const campoInteresse = document.querySelector("input");
campoInteresse.addEventListener("keyup", e => {
  if (e.key === "Enter") {

    const novoLi = document.createElement("li");
    const novoSpan = document.createElement("span");
    const novoBotao = document.createElement("button");

    novoSpan.textContent = campoInteresse.value;
    novoBotao.textContent = '✕';

    novoLi.appendChild(novoSpan);
    novoLi.appendChild(novoBotao);
    const listaInteresses = document.querySelector("ol");
    listaInteresses.appendChild(novoLi);

    novoBotao.onclick = function () {
      listaInteresses.removeChild(novoLi);
      // Outra opção: novoLi.remove();
      // Outra opção: novoLi.parentNode.removeChild(novoLi);
    }
    campoInteresse.value = '';
  }
})
```

Este exemplo é uma alteração do exemplo anterior que adiciona um botão "x" dentro do <li> para permitir ao usuário excluir o item da lista posteriormente. Observe que cada item de lista passa a conter dois elementos, um para o texto e um <button> para o botão "x". Repare também que foi adicionada uma função para tratar o evento click no botão "x".

Outras Propriedades e Métodos para Manipulação da Árvore DOM

node.parentNode

- retorna o nó pai do nó em questão

node.nextSibling

- retorna o próximo nó irmão (nó de **qualquer tipo**)

node.nextElementSibling

- retorna o próximo nó irmão do **tipo elemento**

node.previousSibling

- retorna o nó irmão anterior (nó de **qualquer tipo**)

node.previousElementSibling

- retorna o nó irmão anterior do **tipo elemento**

Outras Propriedades e Métodos para Manipulação da Árvore DOM

node.cloneNode(deep)

- duplica o objeto correspondente ao nó
- se o parâmetro deep for **true**, clona também os nós filhos
- pode ser usado para duplicar um ramo do documento HTML (o clone não é inserido automaticamente na árvore DOM)

Outras Propriedades do Objeto *document*

- `document.head` - acesso direto ao nó corresp. ao elemento <head>
- `document.body` - acesso direto ao nó corresp. ao elemento <body>
- `document.title` - acesso direto ao nó corresp. ao elemento <title>
- `document.location` - objeto com URL da página. Pode ser modificado.
- `document.forms` - retorna coleção de todos os formulários (<form>)
- `document.images` - retorna coleção de todas as imagens (<img>)
- `document.anchors` - retorna coleção de todos os links (<a>)

Exemplo de Uso de document.forms

HTML

```
<form name="cadastro" action="cadastra.php" method="post">
   <p><label>Produto: <input name="produto"></label></p>
   <p><label>Último Nome: <input name="ultimo-nome"></label></p>
   <label>Estado:
      <select name="estado">
         <option value="MG">Minas Gerais</option>
         <option value="SP">São Paulo</option>
      </select>
   </label>
</form>
```

JavaScript

```
const valorDoCampoProduto = document.forms.cadastro.produto.value;
const valorDoCampoUltNome = document.forms.cadastro["ultimo-nome"].value;
const valorDoCampoEstado = document.forms.cadastro.estado.value;
```

Observe que neste exemplo os valores dos campos são resgatados no código JavaScript de forma direta e prática, utilizando document.forms e o nome do formulário em questão, seguido pelo nome do campo. Repare que nem foi necessário utilizar o método querySelector.

Formas Alternativas de Uso de document.forms

HTML

```
<form name="cadastro" action="cadastra.php" method="post">
   <p><label>Produto: <input name="produto"></label></p>
   <p><label>Último Nome: <input name="ultimo-nome"></label></p>
</form>
```

Outras formas de acessar o nó correspondente ao campo "produto"

- `const campoProduto = document.forms.cadastro.elements.produto;`
- `const campoProduto = document.forms["cadastro"].elements.produto;`
- `const campoProduto = document.forms[0]["ultimo-nome"];`
- `const campoProduto = document.forms["cadastro"]["produto"];`
- `const campoProduto = document.forms.item(0)["produto"];`
- `const campoProduto = document.forms.namedItem("cadastro")["produto"];`

Conteúdo Complementar

Exemplo de Formulário a ser Validado com JavaScript

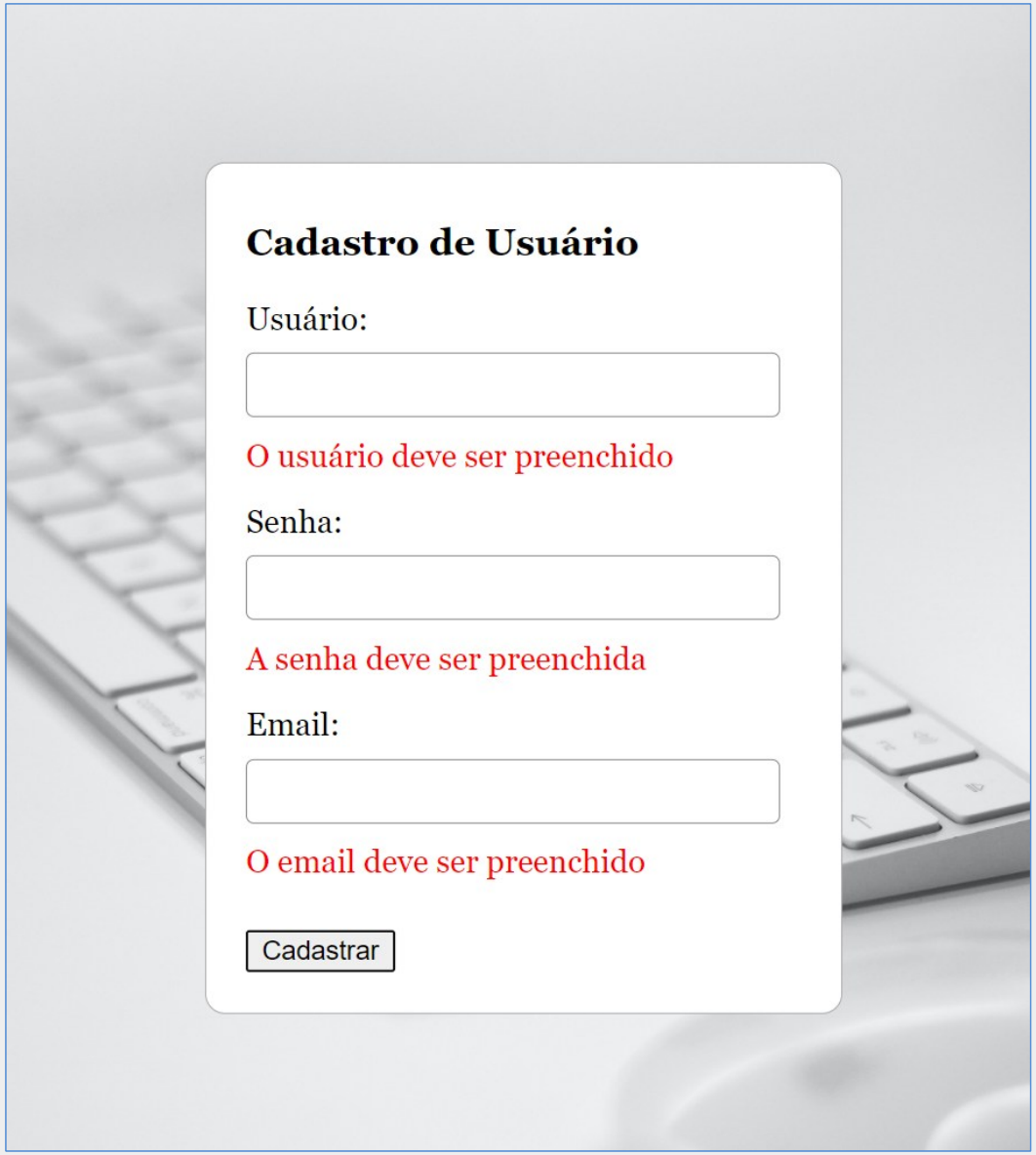
```
<form name="cadastro" action="login.php" method="post">
  <div>
    <label for="usuario">Usuário:</label>
    <input type="text" id="usuario" name="usuario">
    <span></span>
  </div>
  <div>
    <label for="senha">Senha:</label>
    <input type="password" id="senha" name="senha">
    <span></span>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email">
    <span></span>
  </div>
  <button>Cadastrar</button>
</form>
```

Observe que depois de cada campo textual há um elemento vazio, que será utilizado no código JavaScript (próximo slide) como container para apresentação de mensagem informativa caso o usuário tente enviar o formulário sem preenchimento.

Exemplo de Código JavaScript para Validar o Formulário Anterior

```
document.forms.cadastro.onsubmit = validaForm;
function validaForm(e) {
  let form = e.target;
  let formValido = true;

  const spanUsuario = form.usuario.nextElementSibling;
  const spanSenha = form.senha.nextElementSibling;
  const spanEmail = form.email.nextElementSibling;

  spanUsuario.textContent = "";
  spanSenha.textContent = "";
  spanEmail.textContent = "";

  if (form.usuario.value === "") {
    spanUsuario.textContent = 'Usuário deve ser preenchido';
    formValido = false;
  }
  if (form.senha.value === "") {
    spanSenha.textContent = 'A senha deve ser preenchida';
    formValido = false;
  }
  if (form.email.value === "") {
    spanEmail.textContent = 'O email deve ser preenchido';
    formValido = false;
  }

  if (!formValido)
    e.preventDefault();
}
```

O método preventDefault impede a execução da ação padrão associada ao evento. Neste caso, a chamada impede que o formulário seja submetido.



Referências

- https://developer.mozilla.org/en-US/docs/Web/JavaScript
- https://www.ecma-international.org/ecma-262/
- David Flanagan. **JavaScript: The Definitive Guide**. 7ª ed., 2020.
- Jon Duckett. **JavaScript and JQuery: Interactive Front-End Web Development**.