DOM - Document Object Model e Eventos em JavaScript

Disciplina: Desenvolvimento Web

Prof. Dr. Rafael Will M. de Araujo

Conteúdo

Sumário

- O que é o DOM
- Como funciona a estrutura em árvore do DOM
- Como fazer buscas em elementos HTML
- Como manipular (criar e alterar) o DOM

DOM - *Document Object Model*

- DOM - "Modelo de Documento por Objetos"
- Quando uma página é carregada o navegador cria um modelo de objetos que representa a página (DOM).
- O DOM é construído como uma árvore de objetos:
  - ▷ Lembre-se que o HTML é basicamente um conjunto de marcadores aninhados.
  - ▷ Um marcador pode ter outros marcadores contidos nele, e cada um desses marcadores pode ter outros marcadores contidos neles...

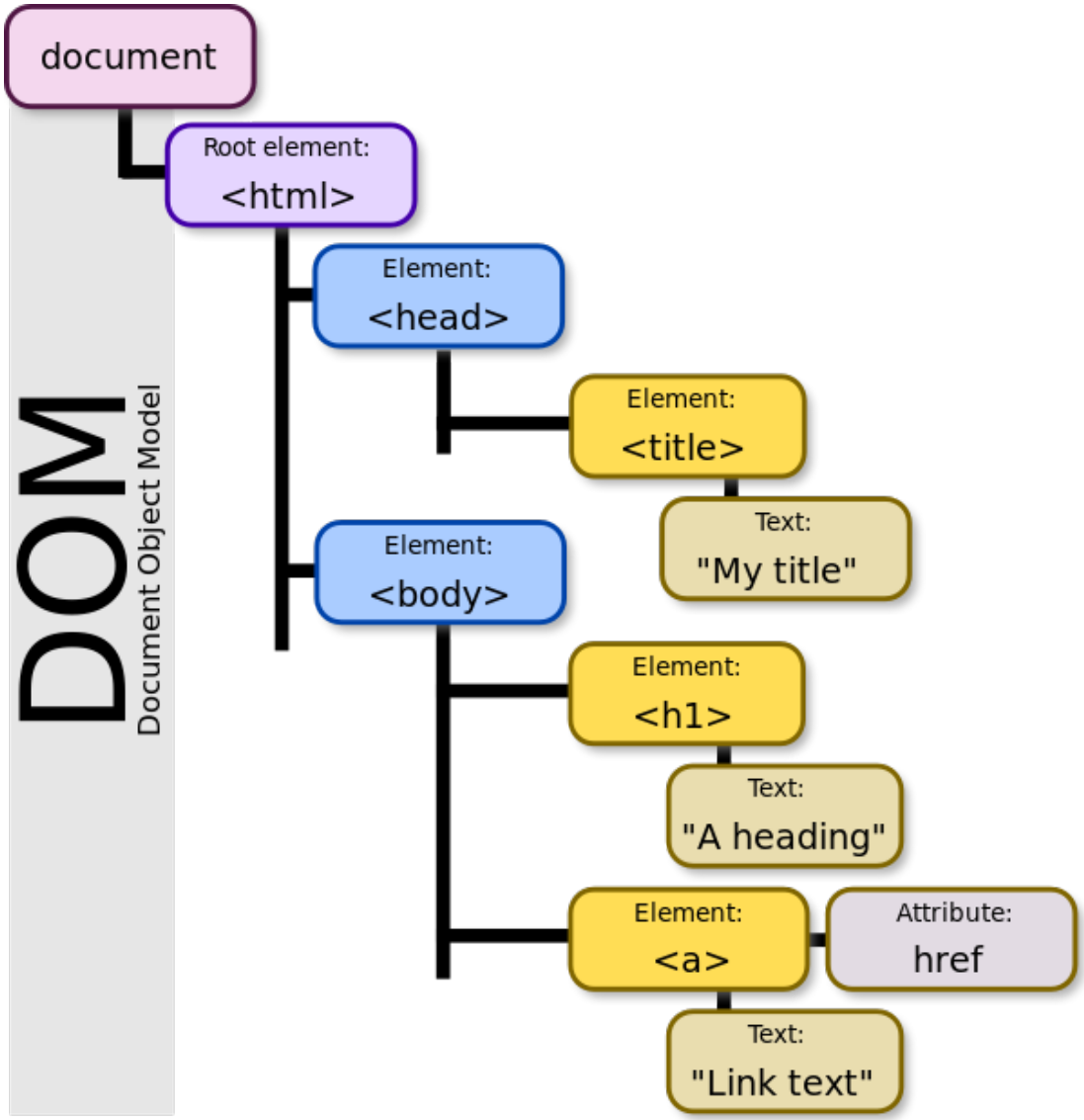
Exemplo 1

```
<html>
    <head></head>
    <body></body>
</html>
```

Exemplo 2

```
<table>
    <tr>
        <td></td>
    </tr>
</table>
```

DOM - *Document Object Model*

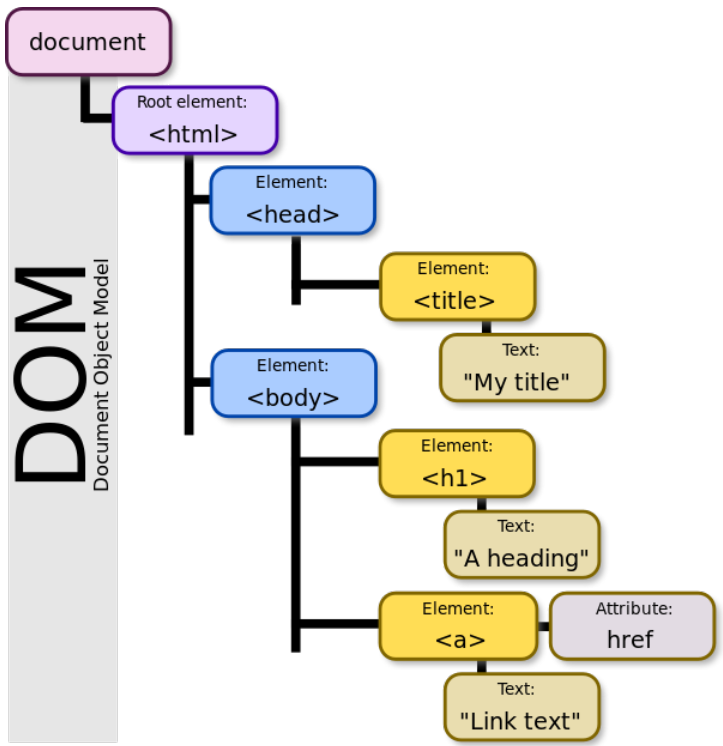


DOM - *Document Object Model*

- Como seria o documento HTML que gerou o DOM do slide anterior?

Exemplo

```
<!DOCTYPE html>
<html>
    <head>
        <title>My title</title>
    </head>
    <body>
        <a href='#'>Link text</a>
        <h1>A heading</h1>
    </body>
</html>
```



DOM - *Document Object Model*

- Porque estudar o DOM?
- O JS consegue recuperar a estrutura da página em um objeto:
  - Ou seja, temos acesso a toda estrutura da página em objetos no JS.
- Desta forma, podemos manipular páginas HTML utilizando JS. É possível, por exemplo:
  - Mudar o conteúdo de elementos HTML;
  - Mudar o estilo de elementos HTML;
  - Trabalhar com eventos;
  - Adicionar e remover elementos HTML.

Conteúdo

DOM - *Document Object Model*

- Uma das grandes funcionalidade de JS é manipular páginas HTML, para deixá-las mais dinâmicas.
- A primeira coisa que precisamos fazer é encontrar um elemento do HTML que desejamos manipular. Para isso, JS oferece um método chamado `document.getElementById()`.
- O `document.getElementById()` permite referenciar dinamicamente qualquer elemento do documento HTML através do seu ID.
- Lembrete: um ID é um atributo definido nos marcadores:

Exemplo

```
<p id="par">Parágrafo com ID</p>
```

DOM - *Document Object Model*

Acessando um objeto pelo DOM

```
<!DOCTYPE html>
<html>
<head></head>
<body>
    <p id="par">Parágrafo com ID</p>
    <script>
        const p = document.getElementById("par");
        alert(p);
    </script>
</body>
</html>
```

DOM - *Document Object Model*

Acessando um objeto pelo DOM

```
<!DOCTYPE html>
<html>
<head></head>
<body>
    <p id="par">Parágrafo com ID</p>
    <script>
        const p = document.getElementById("par");
        alert(p);
    </script>
</body>
</html>
```

- Teste o código acima. Em seguida, inverta a posição do script (coloque-o antes do parágrafo) e verifique a diferença na execução.

DOM - *Document Object Model*

- Note que no exemplo anterior o comando *alert()* não mostrou muita informação relevante:
  - `[object HTMLParagraphElement]`
- Conseguimos entender que é um objeto do tipo parágrafo em HTML. Mas e o que exatamente há "dentro" dele?
- O JavaScript oferece um método para saber o que há dentro dos objetos JS:
  - `console.log(obj);` - no Firefox
  - `console.dir(obj);` - no Chrome
- Lembrando que para visualizar o resultado do método acima, devemos utilizar o menu de ferramentas de desenvolvedor. Nos navegadores Chrome ou no Firefox esse menu é exibido ao pressionar a tecla F12.

Outras formas de recuperar elementos HTML

- Existem outras formas de recuperar elementos HTML, não só pelo ID.
- É possível recuperar elementos pelo **nome do marcador**:
  - `const objetos = document.getElementsByTagName("p");`
  - O argumento deverá ser sempre o nome do marcador (Exemplos: p, div, h1, table, ul, *etc*).
- Pelo **nome da classe**:
  - `const objetos = document.getElementsByClassName("classe");`
  - Retorna todos elemento que possuem o atributo: `class="classe"`;
- Utilizando **seletores CSS**:
  - `const objetos = document.querySelectorAll("p.importantes");`
  - No exemplo acima, serão selecionadas todas as *tags* `p` (parágrafos) que estejam na classe `importantes`.

Todos os métodos acima **retornam um array de objetos**. Os elementos serão acessados como: `objetos[0]`, `objetos[1]`, `objetos[2]`, ... e assim por diante.

- Existe um outro método que retorna o **primeiro objeto** que corresponde ao seletor:
  - `const objeto = document.querySelector("p.importantes");`

Acessando elementos dentro de outros elementos

- O JavaScript permite recuperar os elementos de um elemento específico. Por exemplo, suponha que desejamos recuperar todos os parágrafos dentro da *tag* `main`:

Acessando parágrafos dentro do *main*

```
const obj_main = document.getElementById("main");
const paragrafos = obj_main.getElementsByTagName("p");
```

Manipulando o HTML

- A manipulação de objetos HTML do JavaScript pode:

Escrever no documento HTML

```
document.write("<p>Mais um parágrafo</p>");
```

Mudar conteúdo HTML de um elemento

```
document.getElementById("id").innerText = "texto"

// ou então:

let variavel = document.getElementById("id")
variavel.innerText = "texto"
```

Mudar o valor de um atributo

```
document.getElementById("id").src = "imagem.jpg"
document.getElementById("tx-nome").value = "Rafael"

// ou então:

let variavel1 = document.getElementById("id")
let variavel2 = document.getElementById("tx-nome")
variavel1.src = "imagem.jpg"
variavel2.value = "Rafael"
```

Manipulando o HTML

- Também é possível modificar estilos CSS:
  - Usando a propriedade: `.style.propriedade`
  - Ou também: `.style["propriedade"]`
- Exemplos:

Modificando a cor de fundo

```
document.getElementById("id").style.backgroundColor = "#0F0";

// ou então:

document.getElementById("id").style["backgroundColor"] = "#0F0";

// ou então:

let variavel = document.getElementById("id")
variavel.style.backgroundColor = "#0F0";
```

Modificando margens

```
document.getElementById("id").style.margin = "10px";
```

Modificando o display

```
document.getElementById("id").style.display = "none";
```

- E várias outras propriedades...

Adicionando novos elementos

- Podemos criar e adicionar novos elementos à elementos já existentes:

Código HTML

```
<div id="div1">
</div>
```

Código JS

```
let div = document.getElementById("div1");
let paragrafo = document.createElement("p");   // cria uma tag p (parágrafo)
div.appendChild(paragrafo);    // adiciona o parágrafo criado como filho da div
```

- Rode o código acima e veja o HTML gerado pelo navegador.

Adicionando novos elementos

- Podemos criar e adicionar novos elementos à elementos já existentes:

Código HTML

```
<div id="div1">
</div>
```

Código JS

```
let div = document.getElementById("div1");
let paragrafo = document.createElement("p");
paragrafo.innerText = "Texto do parágrafo";
paragrafo.style.color = "magenta";
div.appendChild(paragrafo);
```

- Rode o código acima e veja o HTML gerado pelo navegador.

Conteúdo

O que são eventos?

- Quando trabalhamos com interface gráfica é necessário programar a resposta do sistema para cada interação do usuário.
- Essas interações podem ser, por exemplo:
  - O clique do mouse;
  - A passagem do mouse por cima de um componente;
  - A mudança de conteúdo de algum componente (especialmente *inputs* - campos de formulários);
  - O apertar de uma tecla;

O que são eventos?

- Quando trabalhamos com interface gráfica é necessário programar a resposta do sistema para cada interação do usuário.
- Essas interações podem ser, por exemplo:
  - O clique do mouse;
  - A passagem do mouse por cima de um componente;
  - A mudança de conteúdo de algum componente (especialmente *inputs* - campos de formulários);
  - O apertar de uma tecla;
- Cada uma dessas interações (além de várias outras existentes) gera o que chamamos de **evento**. E um evento pode ser respondido por uma **função**.
  - Uma função em **JavaScript**!

Eventos em JavaScript

- O JavaScript possui uma programação que possibilita "escutarmos" a manifestação desses eventos em cada elemento HTML, adicionando "escutadores" (**listeners**) nos elementos que queremos "escutar".
- Esses *listeners* são funções que criamos para serem executadas toda vez que um determinado evento acontecer no nosso elemento HTML especificado.

Exemplos

- O que acontece ao clicar nos componentes gerados pelos exemplos a seguir?

Exemplo de um botão do tipo *button*

```
<input type="button" value="Clique aqui" id="bt-botao">
```

Exemplo de uma div qualquer

```
<div id="div1">
    Clique aqui
</div>
```

Exemplos

- O que acontece ao clicar nos componentes gerados pelos exemplos a seguir?

Exemplo de um botão do tipo *button*

```
<input type="button" value="Clique aqui" id="bt-botao">
```

Exemplo de uma div qualquer

```
<div id="div1">
    Clique aqui
</div>
```

- Nada acontece, pois não programamos os eventos que respondem por essas ações.

Conteúdo

Associando um listener a um evento

- A associação entre um listener e o seu evento é chamado de **registro de listener**, e pode ser feito de duas maneiras:
  - usando um atributo específico **on***
  - usando o método `addEventListener()`
- Todo evento definido pelo navegador e passível de ser escutado em um elemento HTML possui um atributo nos objetos equivalente ao seu nome, prefixado por **on**.
- Alguns exemplos:
  - `onclick` - quando o elemento é clicado com o mouse;
  - `onmouseover` - quando o mouse passa por cima do elemento;
  - `onmouseout` - quando o usuário afasta o mouse de um elemento;
  - `onkeydown` - quando uma tecla é pressionada;
  - `onchange` - quando o valor do elemento é alterado;
  - `onfocusout` - quando o elemento perde o foco (ou seja, o foco foi passado para outro elemento);

Associando listeners a eventos: opção 1

- Podemos associar uma função já existente a um evento, através de atribuição simples ao atributo **on***:

Código JS: eventos.js

```
function responde_botao() {
  alert("Clicou no botão!");
}

function responde_div() {
  alert("Clicou na div!");
}

const btn = document.getElementById("bt-botao");
const div = document.getElementById("div1");

btn.onclick = responde_botao;   // associa a função responde_botao() ao evento onclick do botão
div.onclick = responde_div;   // associa a função responde_div() ao evento onclick da div
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">

<br><br>

<div id="div1">
    Clique aqui
</div>
```

Associando listeners a eventos: opção 2

- Também podemos associar uma função anônima ao atributo **on***:

Código JS: eventos.js

```
const btn = document.getElementById("bt-botao");
const div = document.getElementById("div1");


btn.onclick = function () {
  alert("Clicou no botão!");
}

div.onclick = function () {
  alert("Clicou na div!");
}
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">

<br><br>

<div id="div1">
    Clique aqui
</div>
```

Associando listeners a eventos: opção 3

- Podemos passar a função como parâmetro no método `addEventListener()` do elemento. Neste caso, deve ser informado também a qual evento aquela função está sendo associada.
  - O nome do evento deve ser passado como string, sem o prefixo **on**, e deve ser o primeiro parâmetro.

Código JS: eventos.js

```
function responde_botao() {
  alert("Clicou no botão!");
}

function responde_div() {
  alert("Clicou na div!");
}

const btn = document.getElementById("bt-botao");
const div = document.getElementById("div1");

btn.addEventListener("click", responde_botao);
div.addEventListener("click", responde_div);
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">

<br><br>

<div id="div1">
    Clique aqui
</div>
```

Associando listeners a eventos: opção 4

- Também podemos passar uma função anônima como parâmetro no método `addEventListener()` do elemento.
  - ▷ Note que apesar de estranha, essa associação é válida no JavaScript. Na prática, o endereço de memória da função está sendo passado por parâmetro para o método `addEventListener()`.

Código JS: eventos.js

```
const div = document.getElementById("div1");
const btn = document.getElementById("bt-botao");

btn.addEventListener("click", function (){
  alert("Clicou no botão!");
});

div.addEventListener("click", function (){
  alert("Clicou na div!");
});
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">

<br><br>

<div id="div1">
    Clique aqui
</div>
```

Conteúdo

mouseover e *mouseout*: exemplo 1

- Seguindo os exemplos anteriores, vamos implementar mais dois eventos para mudar a cor de fundo da div ao passar o mouse em cima dela (sem clicar).

Código JS: eventos.js

```
const div = document.getElementById("div1");
const btn = document.getElementById("bt-botao");

btn.addEventListener("click", function (){
  alert("Clicou no botão!");
});

div.addEventListener("click", function (){
  alert("Clicou na div!");
});

div.addEventListener("mouseover", function(){
  div.style.backgroundColor = "red";
});

div.addEventListener("mouseout", function(){
  div.style.backgroundColor = "white";
});
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">

<br><br>

<div id="div1">
    Clique aqui
</div>
```

Usando o objeto *event*: exemplo 2

- O objeto **event**: toda função de evento recebe como parâmetro de entrada um objeto de evento contendo alguns métodos e propriedades associados ao evento em si.
  - ▷ No evento *click* podemos consultar as posições X e Y do mouse no momento do clique.

Código JS: eventos.js

```
const btn = document.getElementById("bt-botao");

btn.addEventListener("click", function (event){
  alert("Clicou no botão! Coordenadas do mouse: (" + event.clientX + ", " + event.clientY + ")");
});
```

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<input type="button" value="Clique aqui" id="bt-botao">
```

Validando formulários: exemplo 3

- Suponha o formulário a seguir.
  - Desejamos avisar ao usuário que ele não deve deixar os campos **nome completo** e **endereço** em branco.
  - O formulário não deve ser enviado se qualquer um desses dois campos não estiver preenchido.

Código HTML: pagina.html

```
<script src="eventos.js" defer></script>

<form>
  <label for="tx-nome">Nome completo:</label><br>
  <input type="text" name="nome" id="tx-nome"><br>

  <label for="tx-endereco">Endereço:</label><br>
  <input type="text" name="endereco" id="tx-endereco"><br>

  <p id="erro"></p>

  <input type="submit" value="Enviar" id="bt-enviar">
</form>
```

Validando formulários

- Código JS para validar o formulário do slide anterior:

Código JS: eventos.js

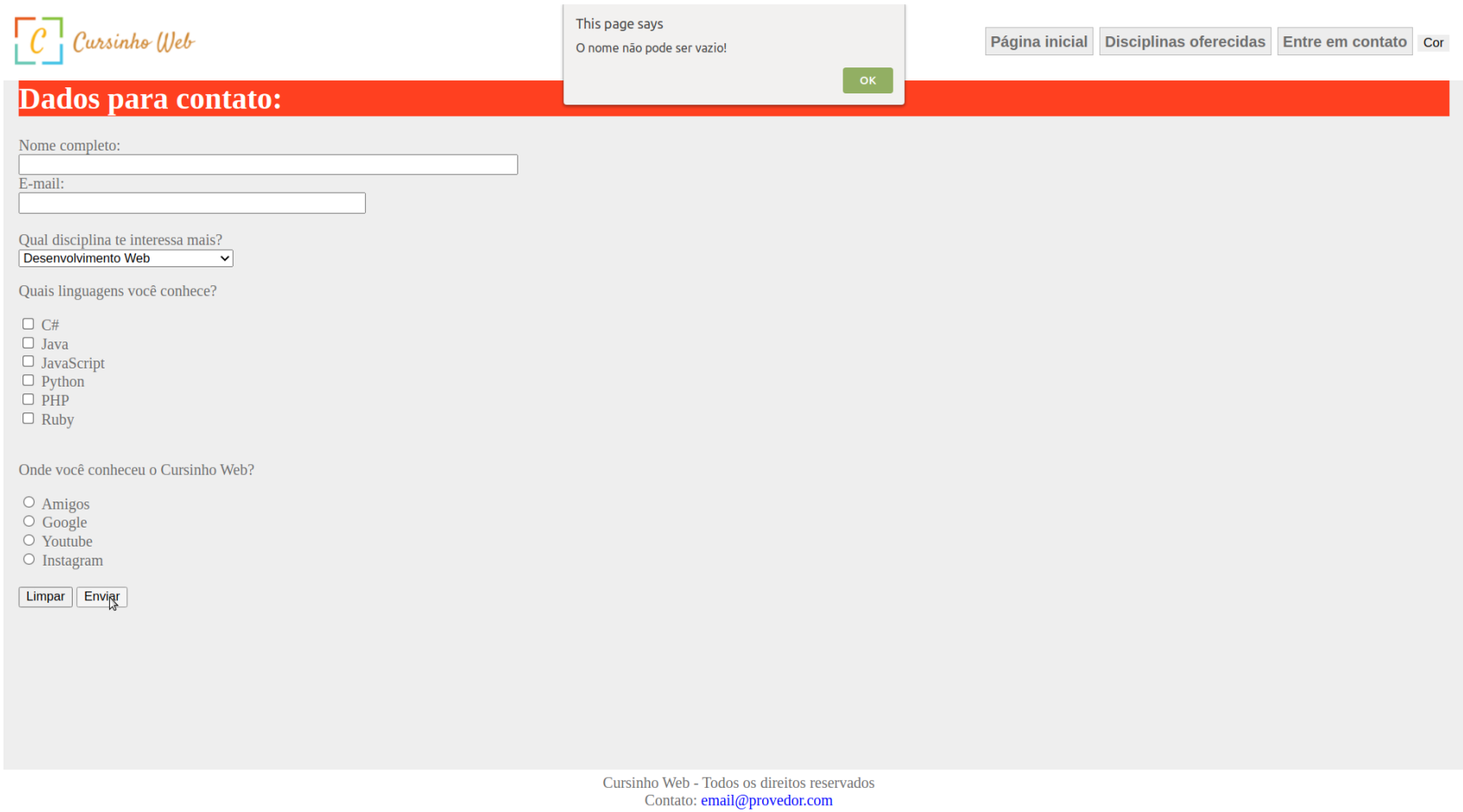
```
const tx_nome = document.getElementById("tx-nome");
const tx_endereco = document.getElementById("tx-endereco");
const par_erro = document.getElementById("erro");
const bt_enviar = document.getElementById("bt-enviar");

tx_nome.addEventListener("focusout", function(){
  if (tx_nome.value.trim() == ""){
    par_erro.innerText = "O campo nome completo deve ser preenchido!";
  }
});

tx_endereco.addEventListener("focusout", function(){
  if (tx_nome.value.trim() == ""){
    par_erro.innerText = "O campo endereço deve ser preenchido!";
  }
});

bt_enviar.addEventListener("click", function (event){
  if (tx_nome.value.trim() == "" || tx_endereco.value.trim() == "") {
    alert("O formulário contém dados incompletos!");
    event.preventDefault();   // impede que o formulário seja enviado
  }
});
```

Conteúdo

Exercício

- Faça a validação do formulário da página *contato.html*. Não deve ser permitido enviar o formulário caso os campos **Nome completo** ou *E-mail* estejam em branco.
  - Dica: lembre de criar variáveis no JS para fazer referência aos dois inputs (nome completo e e-mail) e ao botão enviar.
  - Para evitar que o formulário seja enviado, utilize o método `preventDefault()` do objeto de evento.
- Caso o nome completo ou o e-mail sejam vazios, devem ser exibidas mensagens (*alerts*) de acordo com as figuras a seguir.

Exercício: *contato.html* com validação do nome completo



Exercício: *contato.html* com validação do e-mail