BACHARELADO EM SISTEMAS DE INFORMAÇÃO
CEFET-RJ NOVA FRIBURGO

Programação de Clientes Web

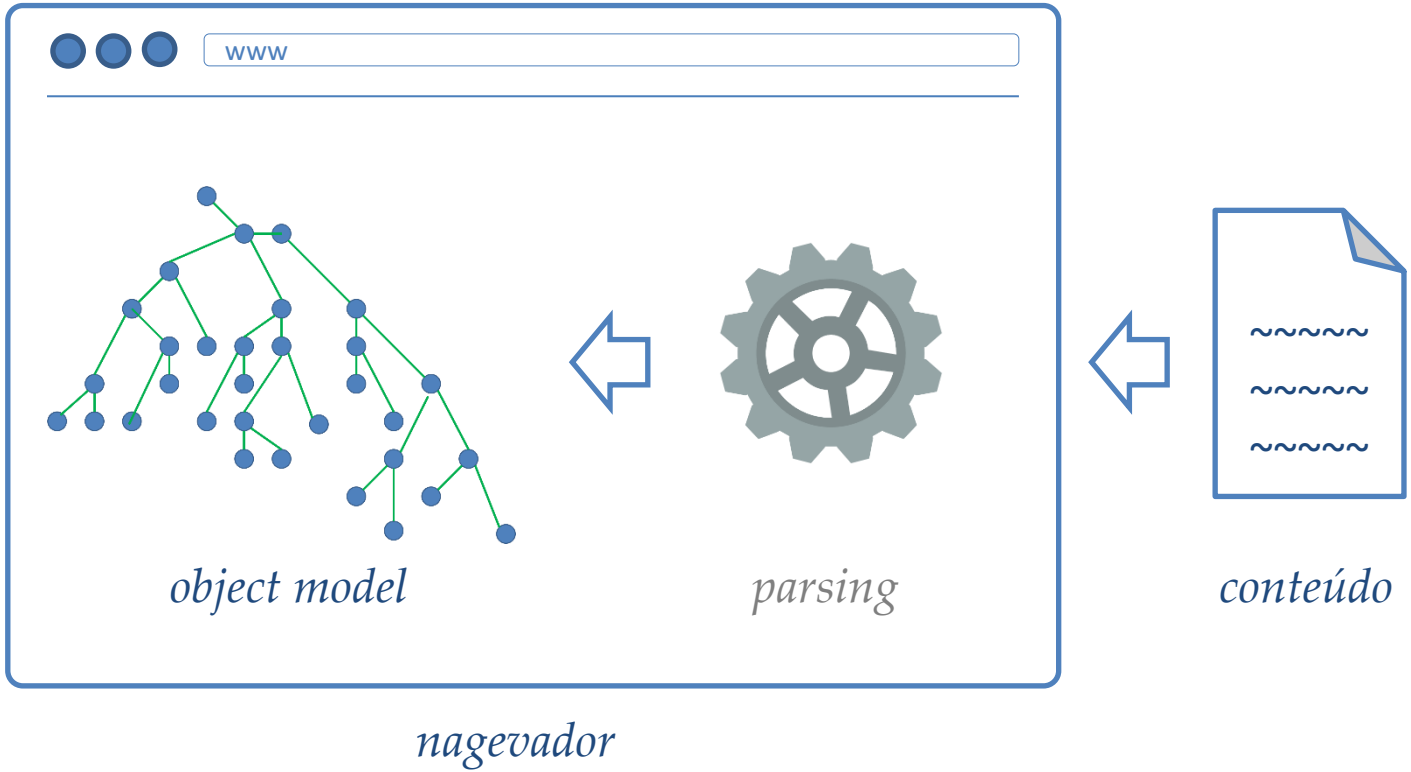
PROF. THIAGO DELGADO PINTO
thiago.pinto@cefet-rj.br

DOM

versão: 2022.05.24

introdução

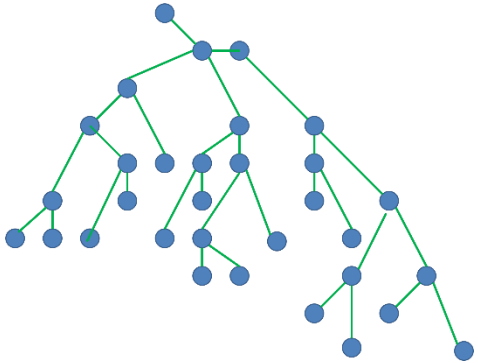
1 de 2



Imagens de https://i.stack.imgur.com/s9wGx.png e https://icons-for-free.com/iconfiles/png/512/cog-1319949677080075 76.png

um **navegador analisa** (*parse*) **conteúdos** e os transforma em um **modelo** estruturado como uma **árvore de objetos**

esse modelo é chamado de *Object Model* (OM)



object model

Imagem de https://i.stack.imgur.com/s9wGx.png

modelos de objetos

- *Document Object Model* (DOM) é a árvore de objetos criada a partir de **elementos HTML**

- *Cascading Style Sheets Object Model* (CSSOM) é a árvore de objetos criada a partir de **definições em CSS**

- *Acessible Rich Internet Applications Object Model* (ARIAOM) é a árvore criada a partir de dados de **acessibilidade** (ARIA) do DOM

montagem do DOM

documento vira `document`

tag `<html>` vira `document.documentElement`

tag `<head>` vira `document.head`

tag `<body>` vira `document.body`

quando analisa uma *tag*, o navegador cria um objeto correspondente e define seus atributos

exemplo:

```
<input type="email" />
```

faz o navegador fazer algo como:

```
const e = document.createElement('input');
e.setAttribute('type', 'email');
parent.appendChild(e);
```

montagem do DOM

`document.createElement` recebe o *tipo* de elemento desejado e procura a **classe** associada, para criar uma **instância**

- **se não encontrar**, cria instância de `HTMLUnknownElement`
  - ou `HTMLElement` se for um nome de *tag* válido (incluir um "-")

ex.: `document.createElement('input')` criará um objeto da classe `HTMLInputElement`

e é possível registrar novos elementos!

- são os chamados *web components*

estrutura

DOM – hierarquia parcial

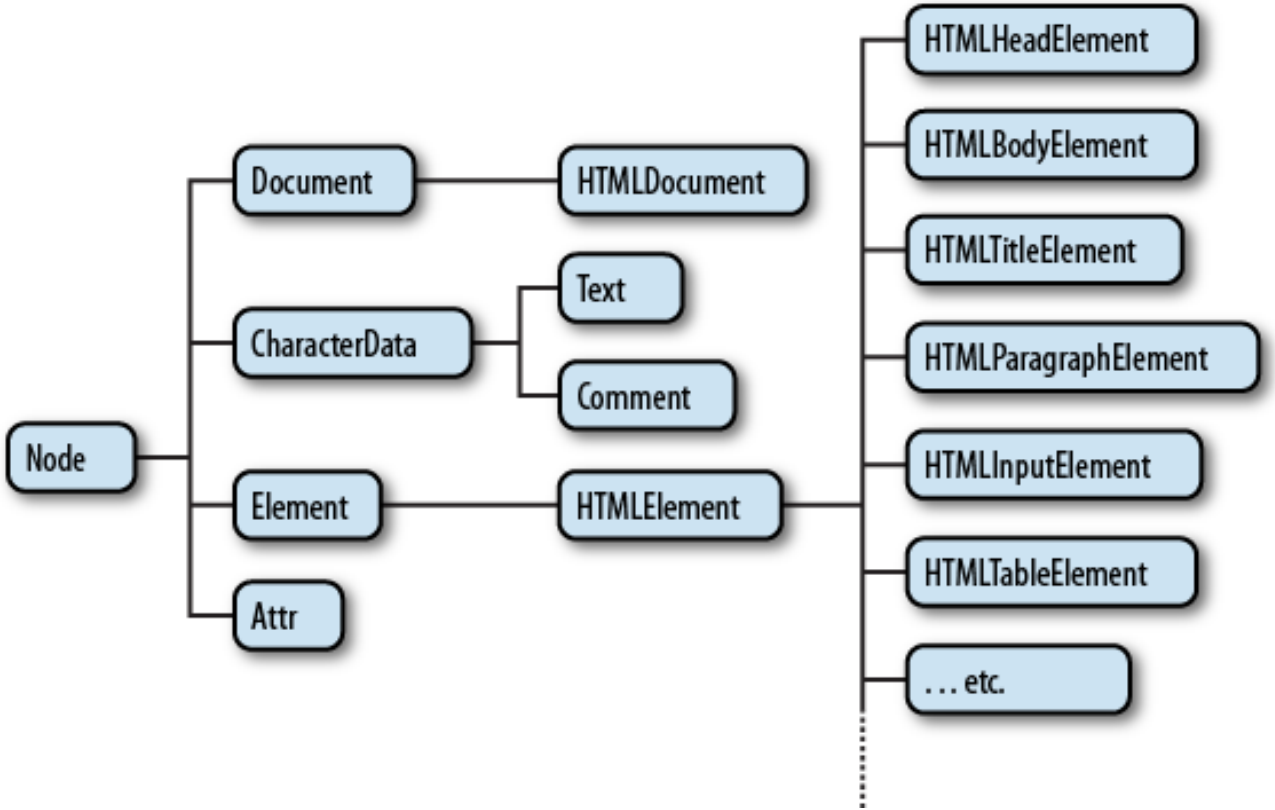
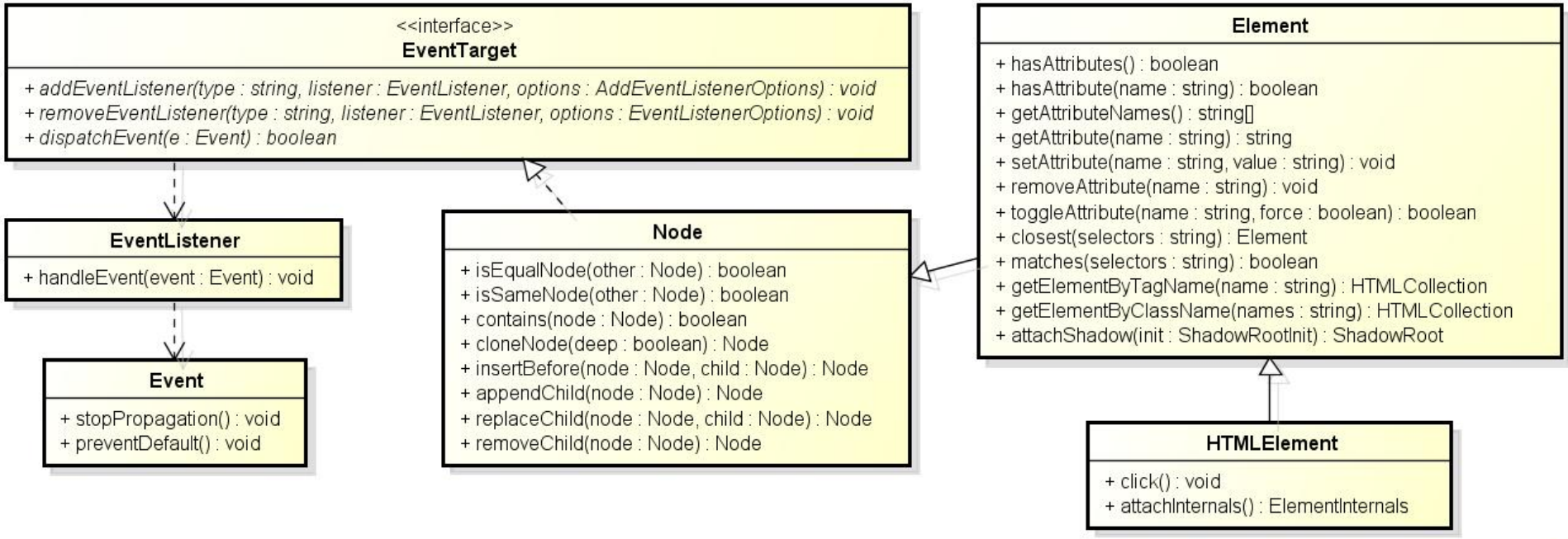


Imagem de https://web.stanford.edu/class/cs98si/slides/the-document-object-model.html

DOM – principais métodos*



*Com exceção de eventos e métodos relacionados. Algumas interfaces/classes foram omitidas.

Fonte: https://html.spec.whatwg.org/#elements-in-the-dom

Veja propriedades e mais métodos em: https://www.w3schools.com/jsref/dom_obj_all.asp

propriedades em EcmaScript 6

propriedades são métodos especiais que simulam o acesso direto aos atributos

mas invocam métodos

prefixo **`get`** em função que **retorna** valor

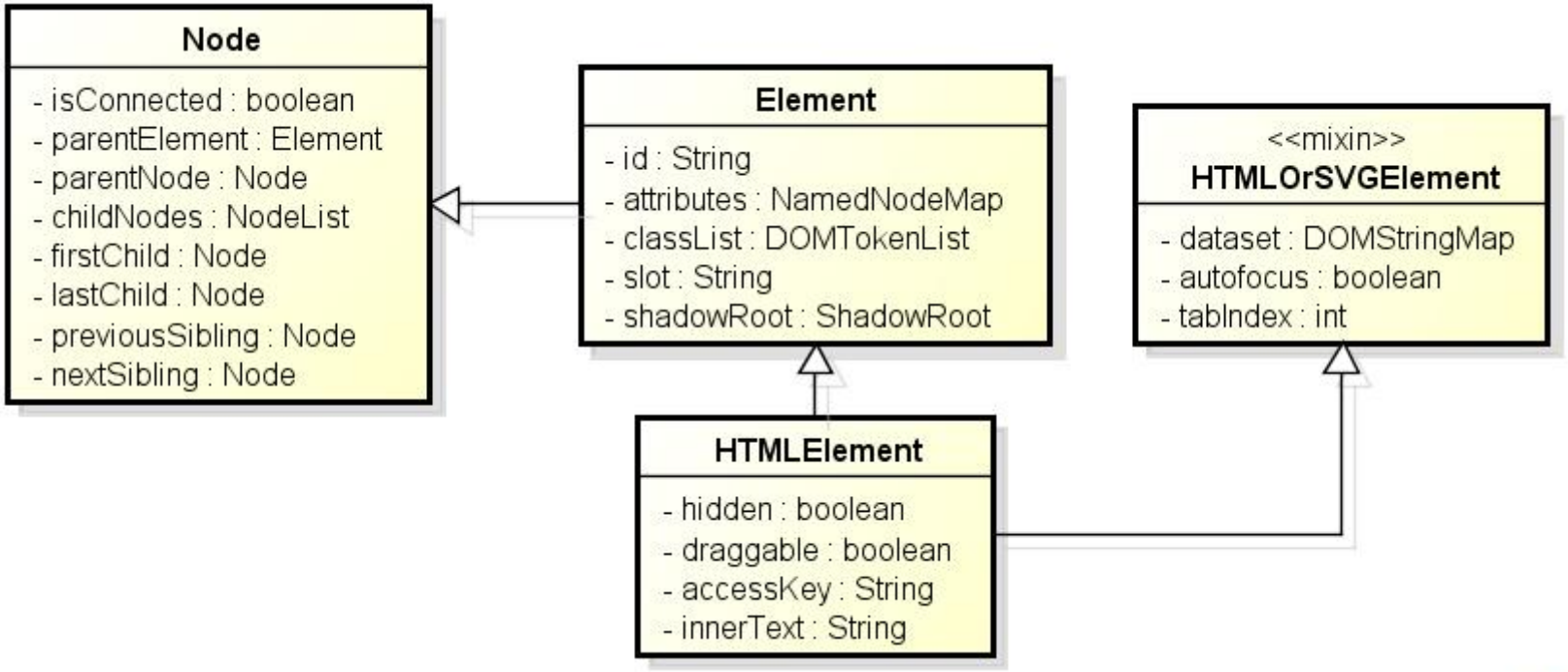
prefixo **`set`** em função que **define** valor

propriedades em EcmaScript 6 – exemplo


```
class Pessoa {
  _idade = 0; // atributo a ser manipulado
  get idade() {
    return this._idade;
  }
  set idade(valor) {
    this._idade = valor;
  }
}

const p = new Pessoa();
p.idade = 20; // invoca set idade
console.log( p.idade ); // invoca get idade
```

propriedades de `HTMLElement`*



*Algumas propriedades, interfaces e classes foram omitidas, por simplificação.

Fonte: https://html.spec.whatwg.org/#elements-in-the-dom

Veja propriedades e mais métodos em: https://www.w3schools.com/jsref/dom_obj_all.asp

acesso a elementos

`parentElement: Element | null` – nó pai como Elemento

`parentNode: Node | null` – nó pai

`childNodes: NodeList` – nós filhos

`firstChild: Node | null` – primeiro nó filho

`lastChild: Node | null` – último nó filho

`previousSibling: Node | null` – nó irmão anterior

`nextSibling: Node | null` – próximo nó irmão

acesso a elementos – exemplo 1

```
<form id="cadastro">
  <input placeholder="Nome"     type="text" />
  <input placeholder="Telefone" type="tel"    id="tel" />
  <input placeholder="E-mail"   type="email" />
  <input text="Enviar"          type="submit" />
</form>
<script>
  const tel = document.getElementById( 'tel' );
  console.log( tel.parentElement );     // form
  console.log( tel.previousSibling );   // Name
  console.log( tel.nextSibling );       // E-mail
</script>
```

acesso a elementos – exemplo 2

```
<form id="cadastro">
  <input placeholder="Nome"     type="text" />
  <input placeholder="Telefone" type="tel"    id="tel" />
  <input placeholder="E-mail"   type="email" />
  <input text="Enviar"          type="submit" />
</form>
<script>
  const f = document.getElementsByTagName( 'form' )[0];
  console.log( f.childNodes );
  // ^Nós Nome, Telefone, Email e Enviar
  console.log( f.firstChild ); // Nome
  console.log( f.lastChild );  // Enviar
</script>
```

acesso a elementos – exemplo 3

```
<form id="cadastro">
  <input placeholder="Nome"     type="text" />
  <input placeholder="Telefone" type="tel"    id="tel" />
  <input placeholder="E-mail"   type="email" />
  <input text="Enviar"          type="submit" />
</form>
<script>
  const f = document.getElementsByTagName( 'form' )[0];
  // Sempre use for..of para iterar nos elementos
  for ( const elemento of f ) {
    console.log( elemento );
  }
</script>
```

observação sobre `childNodes`

`childNodes` retorna o tipo `NodeList`, não um `Array`

é iterável com `for..of`, mas **não** tem os métodos de `Array`

para converter para um `Array`, use:

```
Array.from( elemento.childNodes );
```

propriedade `dataset`

mantém **parâmetros** do tipo `data-*`

exemplo: `<div data-codigo="123" ></div>`

```
const e = document.getElementByTagName('div')[0];
console.log( e.dataset['codigo'] ); // 123
```

propriedade `attributes`

mantém os **parâmetros declarados** de uma *tag*

quando lidos do HTML ou ajustados via **`dataset`**

exemplo: `<input type="email" data-x="123" />`

```
const e = document.getElementByTagName('input')[0];
console.log( e.attributes['type'].value ); // email
console.log( e.attributes['data-x'].value ); // 123
```

propriedades reflexivas

podem existir em **`attributes`**

somente se forem lidas do HTML

exemplos: **`id`**, **`name`**, **`value`**, **`type`**, ...

e outras → variam conforme tipo de elemento

sempre as defina diretamente → ativa método **`set`**

ex. `meuInput.value = "100";`

outras propriedades

não são atributos da *tag*

e *não são* acessíveis via **`attributes`**

são propriedades do objeto (**`get`**/**`set`**)

acessíveis via código

ex. `<div innerText="Oi" ></div>` *não* ajusta o texto

porém `objeto.innerText = "Oi"` irá

manipulando classes

fácil pela propriedade `classList`

que retorna **`DOMTokenList`**

é iterável e possui métodos como

```
add(token1: string, [token2: string, ...])
remove(token1 : string, [token2: string, ...])
toggle(token: string, force: boolean): boolean
contains(token: string): boolean
```

Veja mais em: https://developer.mozilla.org/en-US/docs/Web/API/DOMTokenList

manipulando classes – exemplo

```
<p class="a b c" ></p>
<script>
  const p = document.querySelector('p');
  console.log( p.classList );           // a b c
  console.log( p.classList.length );    // 3
  console.log( p.classList.item( 0 ) ); // a
  p.classList.add( "d" );     // fica "a b c d"
  p.classList.remove( "d" ); // fica "a b c"
  p.classList.toggle( "d" ); // fica "a b c d"
  p.classList.toggle( "d" ); // fica "a b c"
</script>
```

consulta de elementos

opções

`getElementById(id: string): Element | null`

pela propriedade **`id`**

`getElementsByTagName(tagName: string): HTMLCollection`

pelo nome de uma *tag*

`getElementsByClassName(names: string): HTMLCollection`

pelo nome de uma ou mais classes CSS, separadas por espaço em branco

`getElementsByName(name: string): NodeList`

pela propriedade **`name`**

`querySelector(selectors: string): Element | null`

por um ou mais seletores CSS, separados por vírgula; retorna o primeiro que encontrar

`querySelectorAll (selectors: string): NodeList`

por um ou mais seletores CSS, separados por vírgula

recomendação

prefira métodos específicos a usar **`querySelector/querySelectorAll`**

métodos específicos são bem mais rápidos (+2x)

- **`getElementById`** se tiver **`id`**
- **`getElementsByClassName`** se for localizar pela **`classe`**
- **`getElementsByName`** se tiver **`name`**

observação 1

métodos que retornam `HTMLCollection` ou `NodeList` são iteráveis com `for..of`

porém, não são do tipo `Array`

para converter:

```
Array.from( htmlCollectionOuNodeList )
```

observação 2

métodos **`getElementsBy*`** retornam uma coleção que é automaticamente atualizada pelo DOM

- *getElementsByTagName*
- *getElementsByClassName*
- *getElementsByName*

ou seja, a coleção fica "**viva**" e reflete o **estado atual do documento**, quando acessada

resumo

Método	Encontra pelo...	Posso chamar em um elemento?	Fica "vivo"?
`querySelector`	seletor CSS	✓	-
`querySelectorAll`	seletor CSS	✓	-
`getElementById`	id	-	-
`getElementsByName`	name	-	✓
`getElementsByTagName`	tag ou '*'	✓	✓
`getElementsByClassName`	classe	✓	✓

Obs.: `getElementById` e `getElementsByName` são invocáveis em `document` apenas.

seletores úteis

gerais

pelo id, ex.: `#estoque`

pela tag, ex.: `table`

pela classe, ex.: `.sucesso`

atributo

por atributo: tag[atributo="valor"]

ex.: input[type="text"]

um input com tipo "text"

ex.: a[href="https://exemplo.org"]

uma âncora (link) com "https://exemplo.org"

ex.: a[href$=".org"]

uma âncora (link) termina com ".org"

ex.: div[class~="form"]

uma div com classe que contém "form"

negação por atributo: tag:*not*([atributo])

ex.: img:not[alt]

irmãos e descendentes

espaço indica algum descendente, ex.: div p
(um p que fica dentro de div, em qualquer nível)

maior indica um descendente direto, ex.: div > p
(um p que é filho imediato de div)

mais indica o irmão logo após, ex.: div + p
(um p que é irmão de div e fica *logo após* ele)

til indica um qualquer irmão localizado após, ex.: div ~ p
(um p que é irmão de div e fica em algum lugar após ele)

em índice

enésimo filho: tag:nth-child(índice)

```
ex.: ul:nth-child(1)    // Primeiro
ex.: ul:nth-child(2n)   // A cada 2
ex.: ul:nth-child(odd)  // Que for ímpar
ex.: ul:nth-child(even) // Que for par
```

primeiro, último, texto selecionado

primeiro filho:	:first-child
último filho:	:last-child
primeira linha de texto:	::first-line
última linha de texto:	::last-line
texto selecionado:	::selection

manipulação de elementos

propriedade `innerHTML`

`innerHTML` permite definir o **conteúdo interno** de um elemento textualmente, via HTML

ele faz *parsing* de todos os elementos

é simples de usar

ex.: `elemento.innerHTML = '<div><button>Oi</button></div>';`

porém ele...

1. é beeeeeeeem lento 😓
2. é sujeito a ataques de injeção de script

ex. conteúdo do usuário/externo malicioso

propriedade `innerHTML`

use apenas quando:

1. a aplicação não precisar ser rápida; e
2. o conteúdo é fixo e foi feito por você, ou teve todas as suas *tags* verificadas/sanitizadas.

prefira **`appendChild`/`append`** sempre que possível

sobretudo em aplicações para o mercado

conteúdo HTML externo

conteúdos externos (ex. de arquivos) precisam ser inseridos via **`innerHTML`**

para isso, é necessário que sejam:

1. **sanitizados**
   use uma biblioteca como DOMPurify
2. adicionados como um **fragmento**
   evita reflow (recálculo da posição de cada elemento do DOM)

conteúdo HTML externo – exemplo


```
<script crossorigin src="https://unpkg.com/dompurify" ></script>
<script>
const conteudoHTML = /* ... conteúdo de algum lugar ... */;
const fragmento = document.createDocumentFragment();
fragmento.innerHTML = DOMPurify.sanitize( conteudoHTML );
destino.appendChild( fragmento );
</script>
```

propriedade `textContent`

`textContent` permite definir um conteúdo como **texto**

é simples de usar

ex.: `elemento.textContent = 'Bom dia, ' + nome.value;`

é seguro, pois não faz *parsing* de *tags*

recomendação

1. crie elementos com `document.createElement`
   se for um texto, use `document.createTextNode`

2. monte elementos adicionando seus filhos com `appendChild` ou `append`
   existem ainda outros métodos (ex. `insertNode`)

3. adicione o elemento ao DOM (idem acima)

adição ao DOM de uma só vez é **mais rápido → recomendado!**
parser não precisa recalcular renderização a cada adição ("reflow")

observação

`appendChild(node: Node): Node`

- recebe somente um nó
- retorna o nó adicionado

`append(child1: Node|DOMString [, child2: Node|DOMString, ...]): void`

- recebe um ou mais nós ou textos sem formatação
  - o texto será transformado em um objeto do tipo `Text`
- não retorna nada

exemplo – formulário dinâmico

```
const form = document.createElement('form');
form.action = 'destino.php';
form.method = 'POST';

const rotuloEmail = document.createElement('label');
rotuloEmail.setAttribute('for', 'email');
rotuloEmail.innerText = 'E-mail:';

const campoEmail = document.createElement('input');
campoEmail.setAttribute('name', 'email');
campoEmail.setAttribute('type', 'email');

const botaoEnviar = document.createElement('input');
botaoEnviar.setAttribute('type', 'submit');

form.appendChild( rotuloEmail );
form.appendChild( campoEmail );
form.appendChild( botaoEnviar );

document.body.appendChild( form ); // Agora faz parte do DOM
```

exemplo – tabela dinâmica

```
const tabela = document.createElement('table');
const thead = tabela.appendChild( document.createElement( 'thead' ) );
const tbody = tabela.appendChild( document.createElement( 'tbody' ) );

const linhaTitulo = thead.appendChild( document.createElement( 'tr' ) );
linhaTitulo.appendChild( document.createElement( 'td' ) ).append( 'Id' );
linhaTitulo.appendChild( document.createElement( 'td' ) ).append( 'Descrição' );
linhaTitulo.appendChild( document.createElement( 'td' ) ).append( 'Preço' );

const linhaCorpo1 = tbody.appendChild( document.createElement( 'tr' ) );
linhaCorpo1.appendChild( document.createElement( 'td' ) ).append( '1' );
linhaCorpo1.appendChild( document.createElement( 'td' ) ).append( 'Guaraná Taí' );
linhaCorpo1.appendChild( document.createElement( 'td' ) ).append( '3.00' );

const linhaCorpo2 = tbody.appendChild( document.createElement( 'tr' ) );
linhaCorpo2.appendChild( document.createElement( 'td' ) ).append( '2' );
linhaCorpo2.appendChild( document.createElement( 'td' ) ).append( 'Grapette Uva' );
linhaCorpo2.appendChild( document.createElement( 'td' ) ).append( '2.50' );

document.body.appendChild( tabela ); // Agora faz parte do DOM
```

exemplo – removendo todos os filhos

```
// Remove filhos de um elemento qualquer
while ( elemento.firstChild ) {
  elemento.removeChild( elemento.firstChild );
}
```

exercício 1

Crie e preencha dinamicamente um **`select`** com cinco cidades da região, que venham de um *array* de objetos. Cada cidade do *array* deve ter os atributos **`nome`** e **`codigo`**. No **`select`**, as opções devem exibir o valor de **`nome`** e guardar o valor de **`codigo`**.

Então, acrescente um botão que mostre, em uma **`div`**, a cidade selecionada.

exercício 2

Crie dinamicamente uma tabela que apresente os dados de um *array* contendo cinco cidades da região, contendo `nome` e `codigo`.

exercício 3

Com base no exercício anterior, acrescente um formulário criado dinamicamente, contendo os campos **`nome`** e **`codigo`** e o botão "**`Adicionar`**". Quando esse botão for clicado, uma cidade deve ser acrescentada ao *array* de cidades e incluída na tabela.

exercício 4

Com base no exercício anterior, crie uma classe CSS chamada "selecionado", que modifique a cor de fundo. Faça com que, quando uma linha for clicada, essa classe CSS seja adicionada à linha, caso não a contenha. Se a linha já contiver a classe CSS, retire-a.

exercício 5

Com base no exercício anterior, faça com que, quando uma linha for clicada e se tornar selecionada, seus dados sejam exibidos no formulário.

DICAS: `tr` tem uma propriedade `sectionRowIndex`, que é o índice dele dentro de seu `thead`/`tbody`/`tfoot`. Também há `rowIndex`, que é o índice em relação à todas as linhas da tabela (incluindo títulos de `thead`).

exercício 6

Com base no exercício anterior, acrescente um botão "`Atualizar`" que faça com que os dados da linha selecionada da tabela, bem como o respectivo item do *array* de cidades, sejam substituídos pelos dados que estão no formulário. Se nenhuma linha da tabela estiver selecionada, o botão deve informar: **"Selecione uma cidade"**.

exercício 7

Com base no exercício anterior, acrescente um botão "**`Remover`**" que remova a linha selecionada da tabela e o respectivo item do *array* de cidades. Se nenhuma linha da tabela estiver selecionada, o botão deve informar: "Selecione uma cidade".

exercício 8

Com base no exercício anterior, faça com que, ao clicar no título de uma coluna, as linhas da tabela sejam ordenadas de forma crescente, pela coluna. Ao clicar em uma coluna que já estiver ordenada de forma crescente, ela deve ser ordenada de forma decrescente, e vice-versa.

exercício 9

Com base no exercício anterior, crie uma caixa de texto para pesquisa e um botão "Pesquisar". Quando clicado, o texto deve ser pesquisado e, caso encontre, a primeira linha correspondente deve ser selecionada. Ao não encontrar, nenhuma linha deve ficar selecionada.

referências e leituras adicionais

WHATWG. *HTML Living Standard*. Disponível em: https://html.spec.whatwg.org/

WHATWG. *DOM Living Standard*. Disponível em: https://dom.spec.whatwg.org/

JavaScript.Info. *The Modern JavaScript Tutorial*. Disponível em: https://javascript.info

leituras adicionais:

Google. **Minimizar o Reflow do navegador**. Disponível em: https://developers.google.com/speed/docs/insights/browser-reflow?csw=1

fim

2022.05.24 – Melhoria dos slides 41 e 42 (conteúdo HTML externo). Faz melhorias visuais.
2021.05.04 – Corrige ortografia no slide 14.
2020.09.04 – Faz pequenos ajustes visuais.
2020.08.27 – Versão completa.