DOM

Introdución

Aínda que a linguaxe JavaScript foi creada inicialmente para os navegadores web, hoxe en día pode usarse en diferentes contornas: un navegador, un servidor web ou un host. Cada unha destas contornas proporciona os seus propios obxectos e funcións adicionais ao núcleo da linguaxe.

En concreto, os navegadores son pezas de software formadas por diferentes partes, moitas das cales non poden ser manipuladas dende JavaScript por razóns de seguridade. A seguinte imaxe representa as partes principais dun navegador web:

- **Navigator**: representa o estado e identificación do navegador e pode usarse para obter información relativa a el. En JavaScript pode accederse a este obxecto usando a propiedade window.navigator.
- **Window**: representa unha ventá que contén un documento DOM. Nun navegador con pestanas, cada pestana é representada polo seu propio obxecto window. En JavaScript faise referencia a ela utilizando a variable global **window**.
- **Document**: representa a páxina actual cargada no navegador. En JavaScript é representado polo obxecto document, accesible a través da propiedade **window.document**. Serve de punto de entrada para acceder á árbore DOM, permitindo así a manipulación do código HTML e CSS da páxina web.

DOM

O **DOM** (*Document Object Model*) é unha representación dun documento web usando unha **estrutura en forma de árbore**, que simboliza a árbore de etiquetas aniñadas dunha páxina HTML.

Ademais, o DOM é unha interface de programación que define:

- os elementos HTML como obxectos.
- as propiedades de todos os elementos HTML
- os métodos de acceso aos elementos HTML
- os eventos para todos os elementos.

O DOM, en conxunto, ofrece un estándar para poder acceder, engadir, modificar ou eliminar etiquetas HTML, ademais de poder cambiar os seus atributos e contido.

O DOM non forma parte da linguaxe JavaScript. É unha API Web usada para manipular sitios web que está dispoñible nos navegadores, non así en Node.js.

No DOM, todos os elementos HTML son definidos como **obxectos**. A árbore DOM estará formada por obxectos de tipo Node (**nodos**) que representan as partes do documento: un parágrafo, unha imaxe, unha cadea de texto, un comentario, etc.

Node é unha clase abstracta na que se basean moitos dos obxectos da API DOM. A árbore DOM estará formada por diferentes tipos de nodos (subclases de **Node**), aínda que os máis habituais son:

- O nodo Document: representa o punto de entrada do DOM
- O nodo Element: representa nodos de elementos (etiquetas HTML).
- O nodo Attr: representa os atributos dun elemento.
- O nodo Text: representa nodos de texto (texto dentro dun elemento).
- O nodo Comment: representa nodos de comentarios.

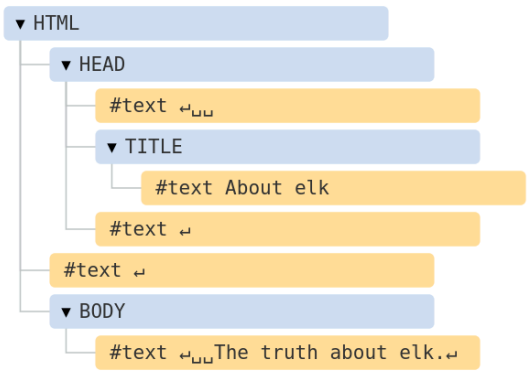
A API DOM organiza todos estes obxectos nunha estrutura xerárquica, onde os nodos descendentes herdan propiedades e métodos dos ascendentes.

Nunha árbore DOM:

- O nodo raíz é **document,** nodo do que colgan o resto de elementos HTML.
- As etiquetas HTML soen xerar dous nodos: a propia etiqueta e o seu contido.
- Os espazos, tabuladores e saltos de liña son caracteres totalmente válidos, ao igual que as letras e os díxitos, polo que tamén se converten en nodos de texto e forman parte do DOM.

O seguinte exemplo mostra unha páxina HTML e o seu DOM. Na imaxe faltaría o nodo raíz **document**, que moitas veces non se representa.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>About elk</title>
</head>
<body>
  The truth about elk.
</body>
</html>
```



Live DOM Viewer, permite visualizar o DOM correspondente a un código HTML.

NOTA: os espazos ao inicio/final da cadea e os nodos de texto que só conteñen espazos habitualmente están ocultos nas **ferramentas de desenvolvemento do navegador**. Tales espazos xeralmente non afectan á forma na que o documento é mostrado.

Cando o navegador encontra **HTML mal escrito** corríxeo automaticamente para crear unha árbore DOM correcta. Por exemplo, se nunha páxina faltan as etiquetas html, head ou body,

o navegador engádeas. Ás táboas tamén se lles engade a etiqueta **tbody,** se non aparece no código HTML.

O seguinte código demostra como acceder ao nodo document e ver as súas propiedades:

```
console.log(window.document);  // mostrar o elemento document
console.log(document);  // non é necesario escribir window
                        // por ser unha propiedade do obxecto global
console.dir(document);  // mostrar o listado de propiedades
```

O DOM é unha das APIs máis usadas, pois permite executar código no navegador para acceder e interactuar con calquera nodo do documento. É posible acceder a cada nodo e modificalo, eliminalo ou engadir novos nodos permitindo cambiar dinamicamente a páxina. Cada nodo da árbore é un obxecto coas súas propiedades e métodos.

Document

O obxecto document ten diferentes propiedades e métodos para traballar cos nodos do DOM. Algunhas destas propiedades son:

- document.documentElement: devolve o elemento raíz de document, é dicir, o elemento **<html>** para documentos HTML.
- document.head: devolve o elemento **<head>** de document.
- document.body: devolve o elemento **<body>** de document.

  NOTA: se un script está dentro de <head> e intenta acceder a document.body, este aínda non existe, polo que devolve **null**. Recordar as estratexias de carga de scripts.

Nos seguintes exemplos móstranse máis propiedades de document:

```
console.log(document.documentElement);  // elemento raíz
console.log(document.head);  // head
console.log(document.body);  // body

// HTMLCollection (obxecto similar a array) de elementos fillos
console.log(document.head.children);

console.log(document.doctype);
console.log(document.URL);
console.log(document.characterSet);
console.log(document.contentType);  // MIME type

// acceso a formularios
console.log(document.forms); // HTMLCollection
console.log(document.forms[0]);
console.log(document.forms[0].method);

// ligazóns
console.log(document.links); // HTMLCollection
console.log(document.links[0]);
```

```
// imaxes
console.log(document.images); // HTMLCollection
console.log(document.images[0]);
console.log(document.images[0].src);
```

NOTA: as propiedades anteriores raramente se usan directamente para acceder aos elementos do DOM. O habitual é usar métodos como **getElementById** e **querySelector** explicados nos seguintes apartados.

Acceso a nodos

No apartado anterior víronse diferentes propiedades do obxecto document, incluíndo algunhas que permitían seleccionar elementos do DOM en forma de **HTMLCollections**.

Neste apartado veranse outros métodos máis prácticos para seleccionar elementos:

- document.getElementById(id): devolve o elemento co id especificado. Dado que os IDs son únicos, é unha forma útil de acceder a un elemento específico rapidamente.

```
<p id="paragraph">Some text here</p>
```

```
console.log(document.getElementById('paragraph'));
```

- document.querySelector(selector): devolve o primeiro elemento do documento co selector especificado, que debe ser un selector CSS válido.

```
  <h1 id="app-title">Título</h1>
  <div class="container"></div>
```

```
console.log(document.querySelector('h1'));
console.log(document.querySelector('#app-title'));
console.log(document.querySelector('.container'));
```

  Tamén pode usarse no obxecto Element. Así, Element.querySelector(selector) devolve o primeiro elemento descendente do elemento dende o que se invoca o método que coincida co selector especificado como parámetro.

```
  <span>Love is Selfless.</span>
  <div id="parent">
    <span>Love is Kind.</span>
    <span>Love is Patient.</span>
  </div>
```

```
const parentElement = document.querySelector('#parent');
console.log(parentElement.querySelector('span'));
```

- document.querySelectorAll(selector): devolve un **NodeList** estático (colección de nodos) de elementos que coinciden co selector especificado.

  Aínda que NodeList non é un Array, é posible iterar sobre a colección usando un bucle **for** e tamén **for…of**. Tamén é posible utilizar os métodos forEach(), entries(), values() e keys().

  NOTA: non usar **for…in** para iterar sobre coleccións como NodeList. O bucle for…in itera sobre as propiedades enumerables dun obxecto e as coleccións teñen propiedades adicionais sobre as que non interesa iterar.

```
const paragrafos = document.querySelectorAll('p');
console.log(paragrafos);

paragrafos.forEach((p) => console.log(p));

console.log(paragrafos[1]);
```

- document.getElementsByTagName(tagName): devolve unha **HTMLCollection** viva de elementos coa etiqueta indicada como parámetro. Este método pode ser invocado no obxecto document ou en calquera elemento.

```
const paragrafos = document.getElementsByTagName('p');
console.log(paragrafos);
```

- document.getElementsByClassName(): devolve unha **HTMLCollection** viva de todos os elementos fillos co nome de clase especificada. Este método pode ser invocado no obxecto document ou en calquera outro elemento.

```
const paragrafos = document.getElementsByClassName('test');
console.log(paragrafos);
```

  NOTA: observar que neste caso escríbese só o nome da clase, mentres que querySelector precisa un selector CSS válido, que para clases inclúe o . (.class).

- document.getElementsByName(name): devolve unha **NodeList** viva de elementos que teñan o atributo “name” co valor especificado como parámetro.

NodeList vs HTMLCollection

Un obxecto HTMLCollection representa unha colección xenérica de elementos e ofrece métodos e propiedades para acceder a eles. Antigamente só podía conter elementos, hoxe en día pode conter tamén outros tipos de nodos.

Unha HTMLCollection é unha colección **viva**, que se actualiza automaticamente cando o documento no que se basea cambia.

Así, por exemplo, o método document.getElementsByTagName(tagName) devolve unha **HTMLCollection,** que é similar a Array. É posible iterar sobre a colección utilizando o bucle **for** ou o **for…of**, mais non se pode usar o método **forEach()**:

```
const paragrafos = document.getElementsByTagName('p');
console.log(paragrafos);

paragrafos.forEach((paragrafos) => { console.log(paragrafos); });  // non funciona
```

Dado que a colección está viva, é recomendable facer unha copia (usando Array.from) para iterar e operar sobre ela:

```
const paragrafos = Array.from(document.getElementsByTagName('p'));
paragrafos.forEach((paragrafos) => { console.log(paragrafos); });
```

Un NodeList é unha colección de nodos. Aínda que non é un array, é posible iterar sobre ela usando forEach().

Tanto NodeList como HTMLCollection son obxectos moi similares:

- Ambas son coleccións tipo array.
- Ambas teñen unha propiedade **length** que devolve o número de elementos da lista.
- Os elementos de ambas coleccións son accesibles mediante o índice, que empeza en 0.
- Os elementos de **HTMLCollection** tamén son accesibles a través do seu **id** e do valor do atributo **name**. Un NodeList non ten esta posibilidade.
- Ambas teñen un método item(), que permite acceder aos nodos ou elementos usando o índice.
- Unha HTMLCollection sempre é unha colección viva (dinámica). Unha NodeList é estática a maioría das veces, aínda que pode ser dinámica:
    - **estática**: os cambios no DOM non afectan ao contido da colección. Por exemplo, document.querySelectorAll() devolve unha NodeList estática.
    - **viva**: o seu contido actualízase cada vez que se engade ou elimina un elemento do DOM. Así, por exemplo, Node.childNodes devolve unha NodeList **viva** de nodos. Sen embargo, childNodes é unha propiedade de só lectura, xa que non se poden engadir nin eliminar elementos da colección directamente. Para facelo haberá que utilizar os métodos apropiados de modificación do DOM.

Exemplo:

```
<p id="id1" name="paragrafo1" class="test">Some text here</p>
<p id="id2" name="paragrafo2" class="container">More text here</p>
  <script>
    // HTMLCollection
    const paragrafos = document.getElementsByTagName('p');
    console.log(paragrafos);
    console.log(paragrafos.length);
```

```
  // acceso mediante nome e id
  console.log(paragrafos.paragrafo1);
  console.log(paragrafos.id1);
  console.log(paragrafos["id1"]);
  console.log(paragrafos.item(0));

  // NodeList
  const paragrafos2 = document.querySelectorAll('p');
  console.log(paragrafos2);
  console.log(paragrafos2[0]);
</script>
```

Exercicios:

1. Descarga o código fonte 01-accesoNodos01.html e indica, polo menos, unha forma de acceder aos seguintes nodos, utilizando os métodos de acceso a nodos:
    - A táboa con id="age-table".
    - Todos os elementos label dentro da táboa (debería haber 3).
    - O primeiro td da táboa.
    - O formulario con name="search".
    - O primeiro input do formulario anterior.
    - O último input do formulario do apartado anterior.
2. Descarga o código fonte 01-accesoNodos02.html e indica, polo menos, unha forma de acceder aos seguintes nodos, utilizando os métodos de acceso a nodos:
    - O elemento con id “input2”.
    - A colección de parágrafos
    - Todos os parágrafos dentro do div con id “lipsum”.
    - O formulario
    - Todos os inputs
    - Só os inputs con nome “sexo”.
    - Os items da lista con clase “important”.

Recorrendo o DOM

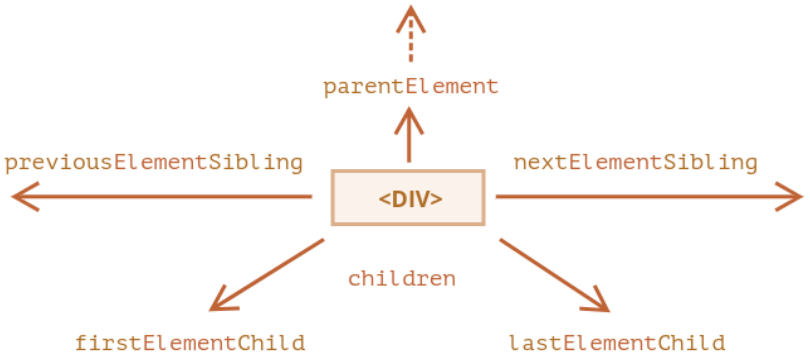
Dado que o DOM é unha estrutura en forma de árbore xerárquica, existen mecanismos para recorrer a árbore, sendo posible navegar por ela e acceder aos nodos fillos, aos descendentes (incluíndo todos os descendentes) e aos antecesores.

NOTA: recordar que a árbore DOM está formada por diferentes tipos de nodos (subclases de Node): Document, Element, Attr, Text, Comment, …

Recorrer elementos

A maioría das veces navegarase pola árbore DOM a través dos nodos que representan **etiquetas**, polo que haberá que usar métodos de navegación de Element.

A seguinte imaxe amosa diferentes **propiedades** dun elemento que permiten navegar polos nodos que son elementos:



Element.firstElementChild	devolve o primeiro elemento fillo ou null.
Element.lastElementChild	devolve o último elemento fillo ou null.
Element.children	devolve unha HTMLCollection, colección viva que contén todos os fillos que son elementos.
Element.nextElementSibling	devolve o seguinte elemento na lista de elementos fillos do pai
Element.previousElementSibling	devolve o elemento anterior na lista de elementos fillos do pai
Node.parentElement	devolve o **elemento** pai do **nodo** especificado, ou null se non ten pai ou se non é un elemento

Exemplo:

```
<div class="parent">
  <div class="child">Child 1</div>
  <div class="child">Child 2</div>
  <div class="child">Child 3</div>
</div>
<script>
  const parent = document.querySelector('.parent');
  console.log(parent);

  console.log(parent.children);

  console.log(parent.firstElementChild);
  console.log(parent.lastElementChild);

  console.log(parent.children[1].previousElementSibling);
  console.log(parent.children[1].nextElementSibling);

  console.log(parent.children[1].parentElement);
</script>
```

Recordar visualizar no Live DOM Viewer como é o DOM correspondente ao código HTML do exemplo anterior.

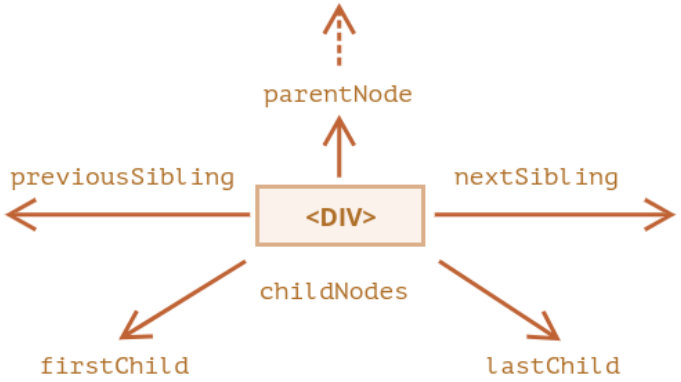
A propiedade Node.parentElement devolve o elemento pai do nodo especificado, ou null se non ten pai ou se non é un elemento. ¿Pode o elemento pai dun elemento non ser un elemento?

```
console.log(document.documentElement.parentNode);
console.log(document.documentElement.parentElement);
```

Recorrer nodos

Algunhas veces interesará recorrer os nodos da árbore DOM, para o que se usarán as propiedades de navegación de Node, que se aplican a todos os nodos, incluíndo nodos de elementos, texto e comentarios.

A seguinte imaxe amosa varias propiedades de Node que permiten navegar polos **nodos** do DOM:



Node.firstChild	devolve o primeiro fillo do nodo, ou null se non ten fillos.
Node.lastChild	devolve o último fillo do nodo, ou null se non ten fillos.
Node.childNodes	devolve unha **NodeList** viva de nodos fillo.
Node.nextSibling	devolve o nodo que está a continuación do actual, de entre os childNodes do pai, ou null se é o último fillo.
Node.previousSibling	devolve o nodo que está antes do actual de entre os childNodes do pai, ou null se é o primeiro da lista.
Node.parentNode	devolve o antecesor do nodo especificado. Dado que document é a raíz da árbore, non ten antecesores, polo que **document.parentNode** devolve null.
Node.hasChildNodes()	devolve un booleano indicando se o nodo ten fillos ou non.

Os nodos fillos (**childNodes**) son os fillos directos, é dicir, os descendentes inmediatos. Por exemplo, <head> e <body> son fillos directos de <html>.

Exemplo:

```
<div class="parent">
  <!-- Children -->
  <div class="child">Child 1</div>
  <div class="child">Child 2</div>
  <div class="child">Child 3</div>
</div>
<script>
  const parent = document.querySelector('.parent');
  console.log(parent);

  console.log(parent.childNodes);

  console.log(parent.firstChild);
  console.log(parent.lastChild);

  console.log(parent.childNodes[2].previousSibling);
  console.log(parent.childNodes[2].nextSibling);

  console.log(parent.childNodes[2].parentElement);
</script>
```

Recorrer táboas

Hai certos elementos do DOM que teñen propiedades específicas para recorrelos, por exemplo, as táboas.

Propiedades dispoñibles para o elemento táboa (HTMLTableElement):

- HTMLTableElement.tHead: devolve o <thead> da táboa ou null.
- HTMLTableElement.tFoot: devolve o <tfoot> da táboa ou null.
- HTMLTableElement.tBodies: devolve unha HTMLCollection viva dos <tbody> da táboa.
- HTMLTableElement.rows: devolve unha HTMLCollection viva das filas da táboa, incluíndo as filas de <thead>, <tfoot> e <tbody>.

  Tamén é posible acceder ás filas de tHead, tFoot e tBodies usando a propiedade rows, por exemplo, **table.tHead.rows**.

Exemplo:

```
<table>
  <tr>
    <td>1</td>
    <td>2</td>
  </tr>
  <tr>
    <td>3</td>
    <td>4</td>
  </tr>
</table>

<script>
  const table = document.getElementsByTagName('table')[0];
  console.log(table);
  console.log(table.tHead);
  console.log(table.tFoot);
  console.log(table.tBodies);
  console.log(table.rows);
</script>
```

O obxecto HTMLTableRowElement representa unha fila e ten propiedades para a súa manipulación:

- **HTMLTableRowElement.cells**: devolve unha HTMLCollection viva de celas da fila.
- **HTMLTableRowElement.rowIndex**: devolve a posición da fila no conxunto da táboa.
- **HTMLTableRowElement.sectionRowIndex**: devolve a posición da fila dentro da sección thead/tbody/tfoot.

O obxecto HTMLTableCellElement representa unha cela e ten propiedades para a súa manipulación:

- **HTMLTableCellElement.cellIndex**: devolve a posición da cela na coleción de <tr>

Exercicios:

1. Dado o seguinte código HTML:

```
<html>
<body>
  <div>Users:</div>
  <ul>
    <li>John</li>
    <li>Pete</li>
  </ul>
</body>
</html>
```

Utilizando as diferentes propiedades para recorrer o DOM, indica, polo menos, unha forma de acceder aos seguintes nodos:

- o nodo
- o nodo
- o segundo

2. Dado un elemento calquera dunha árbore DOM:
    a. ¿É certo que **elemento.lastChild.nextSibling** é sempre **null**?
    b. ¿É certo que **elemento.children[0].previousSibling** é sempre **null**?
3. Escribe o código para pintar as celas diagonais dunha táboa de vermello. Debes investigar en internet como cambiar a cor de fondo dunha cela mediante JavaScript.

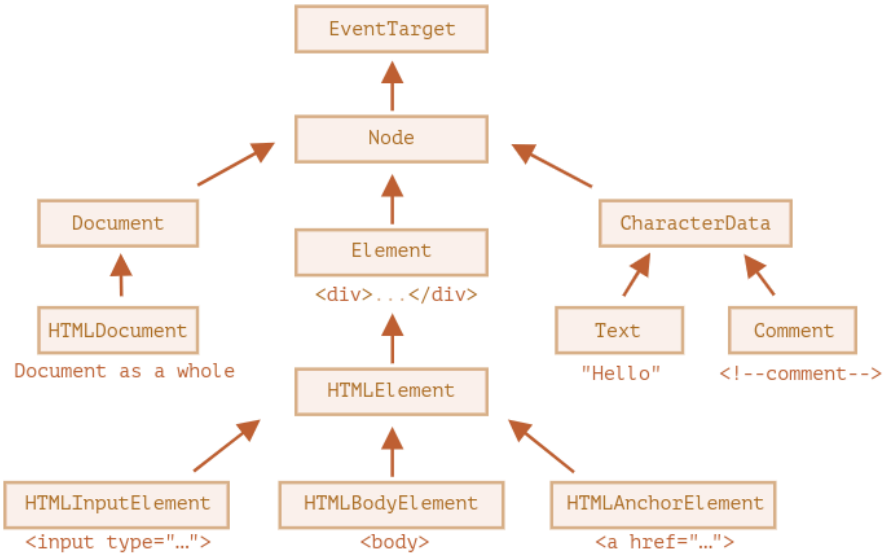
    A táboa debería quedar similar a esta:

1:1	2:1	3:1	4:1	5:1
1:2	2:2	3:2	4:2	5:2
1:3	2:3	3:3	4:3	5:3
1:4	2:4	3:4	4:4	5:4
1:5	2:5	3:5	4:5	5:5

Propiedades dun nodo

En JavaScript existen diferentes tipos de nodos definidos por interfaces diferentes, polo que cada un terá propiedades e métodos diferentes. Así, por exemplo, unha etiqueta <a>, representada pola interface HTMLAnchorElement, terá propiedades e métodos relacionados coa ligazón. Un <input>, representado pola interface HTMLInputElement, terá propiedades e métodos relacionados coa caixa de entrada <input>.

A seguinte imaxe mostra unha xerarquía obxectos, onde a raíz da árbore é EventTarget.



A estrutura xerárquica anterior está formada por diferentes clases de obxectos:

- EventTarget: interface raíz da árbore que é implementada por obxectos que poden recibir eventos aos que se lles poden engadir listeners.
- Node: clase abstracta que serve de base para os obxectos do DOM. Proporciona funcionalidade para navegar polos nodos do DOM (firstChild, lastChild, parentNode, …). As subclases de node herdan as súas funcionalidades.
- Document: representa a páxina web cargada no navegador e serve como punto de entrada á árbore DOM.
- CharacterData: interface abstracta que representa un obxecto que contén caracteres. É implementada por outras interfaces como:
    - Text: representa un nodo de texto do DOM, por exemplo “Hello” en Hello
    - Comment: representa os comentarios.
- Element: clase base para elementos do DOM. Proporciona propiedades e métodos comúns para todos os elementos, como por exemplo os métodos de navegación do DOM (children, firstElementChild, lastElementChild,…) e métodos para obter elementos, como por exemplo Element.querySelector().
  O DOM tamén pode usarse para documentos SVG, polo que Element tamén serve de base para SVGElement.
- HTMLElement: obxecto que representa un elemento HTML. Pode consultarse en Internet a lista completa elementos HTML, algúns deles son son: HTMLBodyElement, HTMLDivElement, HTMLInputElement, HTMLAnchorElement, …

O conxunto completo de propiedades e métodos dun nodo obtense como resultado da herdanza de varias clases.

Pode averiguarse a clase dun nodo accedendo á propiedade **constructor.name**:

```
console.log(document.body.constructor.name);
```

A continuación enuméranse algunhas das propiedades dun nodo, que son útiles para manipular o DOM:

- Element.innerHTML: permite ler/establecer o HTML dos descendentes dun elemento.

```
<h1 id="app-title">Shopping <u>List</u></h1>
<script>
  const title = document.getElementById('app-title');
  console.log(title.innerHTML);
  title.innerHTML = '<u>Hello world</u>';
</script>
```

- HTMLElement.innerText: representa o texto renderizado dun nodo e os seus descendentes. Aproxímase ao texto que se obtería se seleccionásemos co rato un texto e se copiase no portapapeis.

```
<h1 id="app-title">Shopping <u>List</u></h1>
<script>
  const title = document.getElementById('app-title');
  console.log(title.innerText);
  title.innerText = 'Hello Again';
</script>
```

- Node.textContent: é o contido de todo o que hai entre a etiqueta de apertura e a de cerre do elemento, incluíndo os espazos e o contido de todos os descendentes. É dicir, devolve unha concatenación do textContent de todos os descendentes incluíndo o contido de <script> interno e <style>. Pola contra, exclúe os comentarios. As etiquetas descendentes tampouco se inclúen na saída.

```
<h1 id="app-title">Shopping <u>List</u></h1>
<script>
  const title = document.getElementById('app-title');
  console.log(title.textContent);
</script>
```

  Diferencias entre innerText e textContent:

  - innerText só mostra *"human-readable" elements*, mentres textContent mostra o contido de todos os elementos, incluíndo <script> e <style>.
  - innerText non mostra os elementos ocultos, mentres que textContent devolve todos os elementos.

  NOTA: innerText utilízase con máis frecuencia que textContent.

Exemplo:

```
<style>
  u {
    display: none;
  }
</style>
<h1 id="app-title">Shopping <u>List</u></h1>

<script>
  const title = document.getElementById('app-title');
  console.log(title.innerText);
  console.log(title.textContent);
</script>
```

- Node.nodeName: propiedade que devolve o nome do nodo actual. Se o nodo é un elemento devolve a etiqueta en maiúsculas.

A maioría dos atributos HTML estándar convértense automaticamente en propiedades do obxecto DOM correspondente. Por exemplo, se a etiqueta é <body id="page">, entón o obxecto DOM ten unha propiedade **document.body.id**:

```
<body id="page">
```

```
console.log(document.body.id);
document.body.id = 'newId';
console.log(document.body.id);
```

Máis exemplos de atributos HTML accesibles mediante propiedades de obxectos do DOM:

- Element.className, propiedade que permite ler/establecer o valor do atributo "class" do elemento. Úsase className en lugar de class, xa que esta é unha palabra reservada en moitas linguaxes de programación.
- HTMLAnchorElement.href: devolve a cadea que contén a URL dunha etiqueta.
- .**value**: devolve o valor da propiedade "value" dun <input> (HTMLInputElement). Como os <inputs> non teñen etiqueta de cerre, non se pode usar .innerHTML nin .textContent.

```
<input type="text" id="elem" value="value">
```

Ademais, todos os atributos HTML, tanto os que son estándar como os que non o son, están accesibles usando os seguintes métodos:

- Element.getAttribute(name): obtén o valor do atributo especificado.
- Element.setAttribute(name, valor): establece o valor do atributo.
- Element.hasAttribute(name): comproba se un elemento ten o atributo especificado.
- Element.removeAttribute(nombre): elimina o atributo.

Os atributos non estándar, creados polas persoas desenvolvedoras, son accesibles usando os métodos anteriores.

```
<div id="order" class="order" order-state="cancelado">...</div>
<script>
  let div = document.getElementById('order');
  console.log(div);
  console.log(div.getAttribute('order-state'));
</script>
```

Sen embargo, usar atributos non estándar pode dar lugar a problemas se no futuro HTML decide estandarizar ese atributo. Para evitar este problema, utilízanse os atributos **data-**.

Os atributos que empezan por **data-** están reservados ser usados polas persoas desenvolvedoras e están accesibles na propiedade **HTMLElement.dataset.** En HTML o nome do atributo empezará por **data-,** mentres que en JavaScript o nome da propiedade non incluirá **data-** e converterase o nome do atributo a camelCase. Así, se un elemento ten un atributo chamado "data-about", estará dispoñible dende a propiedade elemento.dataset.about:

```
<div id="user" data-id="12345678" data-user="carinaanand" data-date-of-birth>...</div>

<script>
  const el = document.querySelector('#user');

  console.log(el);
  console.log(el.id);    // el.id === 'user'
  console.log(el.dataset.id);     // el.dataset.id === '12345678'
  console.log(el.dataset.user); // el.dataset.user === 'carinaanand'
  // os atributos de varias palabras (data-date-of-birth) convértense en camelCase:
  el.dataset.dateOfBirth = "1960-10-03";
  console.log(el.dataset.dateOfBirth);    // el.dataset.dateOfBirth === "1960-10-03";
</script>
```

NOTA: Named Element IDs Can Be Referenced As JavaScript Globals | CSS-Tricks

javascript - Do DOM tree elements with IDs become global properties? - Stack Overflow

O valor do atributo **id** de todos os elementos e o valor do atributo **name** dalgúns elementos están dispoñibles como propiedades do obxecto global Window e fan referencia ao elemento do DOM con este atributo, aínda que este comportamento non está estandarizado en todos os navegadores.

Acceder a estes elementos usando a propiedade do obxecto global non é seguro e pode ser perigoso. Recoméndase usar outros métodos de acceso a nodos como **querySelector()** ou **getElementById()**.

Exercicios:

1. Descarga o código fonte 03-propiedadesNodo01.html e indica, polo menos, unha forma de acceder ao seguinte contido e mostralo por consola:
    - O innerHTML, innerText e textContent da etiqueta “Escolle sexo”:
    - O valor do primeiro input de sexo
    - O valor do sexo que estea seleccionado.
    - O texto de cada un dos elementos
    - Indica cantos elementos <li> hai.
    - Indica o valor do atributo data-widget-name

2. Descarga o código fonte 03-propiedadesNodo02.html e mostra por consola:
    - O número de ligazóns da páxina.
    - A dirección da penúltima ligazón.
    - O número de ligazóns que apuntan a http://proba
    - O número de ligazóns do terceiro parágrafo.
    - Modifica o estilo das ligazóns que apuntan a http://proba para que teñan o texto de cor laranxa.

Modificando o documento

É posible crear novos elementos e modificar o contido existente da páxina ofrecendo dinamismo. Para crear novos nodos utilizaranse os seguintes métodos:

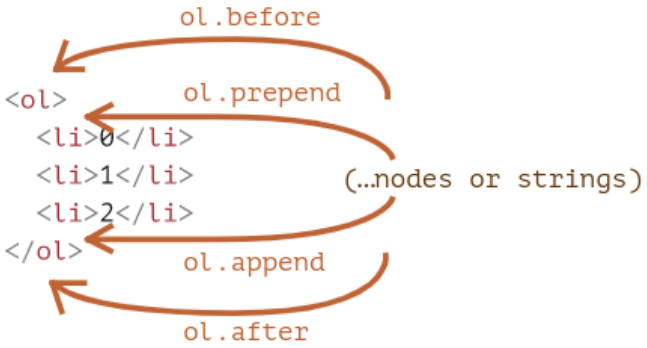
- Document.createElement(etiqueta): crea un elemento HTML do tipo especificado na etiqueta.

```
const div = document.createElement('div');
div.className = 'clase';
div.id = 'id';
div.setAttribute('title', 'Título');
div.innerText = 'Hello World';
console.log(div);
```

- Document.createTextNode(data): crea un nodo de texto.

```
const text = document.createTextNode('Hello World');
console.log(text);
```

Unha forma de engadir os elementos creados ao DOM é obter unha referencia a un elemento existente e inserir novos elementos usando algún dos métodos indicados na seguinte imaxe, onde ao mesmo tempo se indica os puntos onde serán inseridos:



A continuación dáse máis información sobre os métodos para facer estas inserciónss:

- Element.append(parametros): insire un conxunto de nodos ou cadeas despois do último fillo de Element.

```
let div = document.createElement('div');
let p = document.createElement('p');
div.append(p);
// p.append('Hello World');
const text = document.createTextNode('Hello World');
p.append(text);
console.log(div);
document.body.append(div);
```

As cadeas son inseridas como nodos de texto.

```
let ul = document.createElement('ul');
ul.append('<li>Elemento 1</li>');
let li = document.createElement('li');
li.append('Elemento 2');
ul.append(li);
document.body.append(ul);
```

Outro exemplo:

```
<ul id="app">
  <li>JavaScript</li>
</ul>
```

```
<script>
  let app = document.querySelector('#app');
  let langs = ['TypeScript', 'HTML', 'CSS'];

  let nodes = langs.map((lang) => {
    let li = document.createElement('li');
    li.append(lang);
    return li;
  });

  app.append(...nodes);
</script>
```

- Element.prepend(parametros): insire un conxunto de nodos ou cadeas antes do primeiro fillo de Element. As cadeas son inseridas como nodos de texto.
- Element.before(parametros): insire un conxunto de nodos ou cadeas na lista de fillos do pai de Element, xusto antes deste Element. As cadeas son inseridas como nodos de texto.
- Element.after(parametros): insire un conxunto de nodos ou cadeas na lista de fillos do pai de Element, xusto despois de Element. As cadeas son inseridas como nodos de texto.

Exemplo:

```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
```

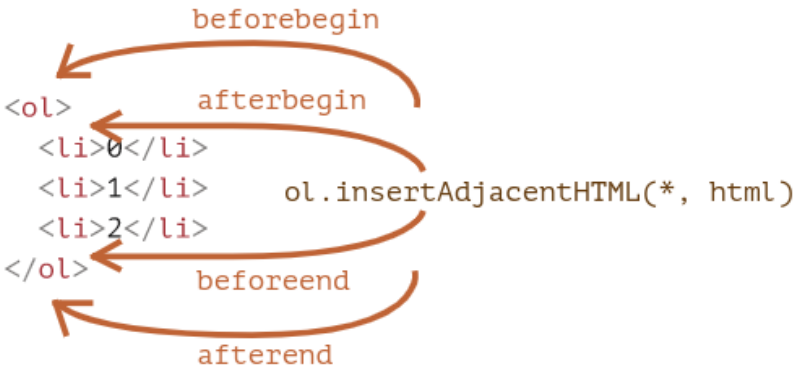
```
id="app">
      <li>JavaScript</li>
    </ul
```

```
before
<ol id="ol">
  <li>prepend</li>
  <li>0</li>
  <li>1</li>
  <li>2</li>
  <li>append</li>
</ol>
after
```

Tamén é posible inserir HTML, texto ou Elementos mediante os métodos Element.insertAdjacentHTML(position, text), Element.insertAdjacentText(position, data) e Element.insertAdjacentElement(position, element), respectivamente. Estes métodos reciben como primeiro parámetro a posición de inserción, que pode tomar un dos seguintes valores:

- “beforebegin”: antes do elemento.
- “afterbegin”: dentro do elemento, antes do primeiro fillo.
- “beforeend”: dentro do elemento, despois do último fillo.
- “afterend”: despois do elemento.

Tomando como exemplo o método insertAdjacentHTML, na seguinte imaxe poden verse graficamente as posicións onde se inserirá o HTML:



Os tres métodos funcionan de forma similar:

- Element.insertAdjacentHTML(position, text): parsea o texto como HTML e insire na posición indicada da árbore DOM os nodos resultado.
- Element.insertAdjacentText(position, data): insire un novo nodo de texto na posición indicada.
- Element.insertAdjacentElement(position, element): insire o elemento pasado como parámetro na posición indicada.

Ademais de inserir elementos, nunha árbore DOM tamén poden realizarse operacións de **reemprazo** e **eliminación** de elementos e nodos:

- Element.replaceWith(parametros): substitúe o Elemento por un conxunto de nodos ou cadeas. As cadeas son inseridas como nodos de texto.
- Element.remove(): elimina o elemento do DOM.

Se o que se quere é mover un nodo dun lugar a outro do DOM, non hai necesidade de eliminalo. Todos os métodos de inserción quitan automaticamente o nodo do lugar vello.

```
<div id="first">Primero</div>
<div id="second">Segundo</div>
```

```
<script>
  // non hai necesidade de chamar a "remove"
  document.getElementById("second").after(document.getElementById("first"));
</script>
```

Os métodos vistos ata agora son métodos modernos e flexibles, xa que permiten engadir múltiples elementos á vez. **Antigamente** usábanse outros métodos que se van listar aquí para poder entender scripts antigos:

- Node.appendChild(node): engade un nodo ao final da lista de fillos do nodo especificado. A diferencia deste método, Element.append() permite pasar múltiples parámetros e tamén cadeas.
- Node.insertBefore(newNode, referenceNode): insire newNodo como fillo de Node e antes do nodo especificado como referencia. Exemplo:

  parentElem.insertBefore(newNode, nextSibling)

- Node.removeChild(child): elimina o nodo fillo especificado por parámetro.
- Node.replaceChild(newChild, oldChild): substitúe o nodo fillo dun nodo por outro pasado por parámetro. A orde de parámetros, o novo fillo antes do vello, non é habitual. O método Element.replaceWith() é máis fácil de ler e usar.

InnerHTML vs createElement

Cando se constrúe unha árbore DOM pode utilizarse a propiedade innerHTML ou crear cada elemento usando o método createElement, que é máis eficiente. A utilización de innerHTML implica que o navegador teña que parsear o código, construír os nodos e inserilos na árbore, que é máis custoso que crear os nodos manualmente e engadilos á árbore:

```
const ul = document.createElement("ul");
const li = document.createElement("li");
li.innerHTML = '<a href="www.google.com">Google</a>';
ul.append(li);
console.log(ul);
```

```
// versión máis eficiente.
const ul2 = document.createElement("ul");
const li2 = document.createElement("li");
const a = document.createElement("a");
a.href = "www.google.com"
const text = document.createTextNode("Google");
a.append(text);
li2.append(a);
ul2.append(li2);
console.log(ul2);
```

Ademais, createElement é unha opción máis seguira pois mitiga os riscos XSS. Tamén facilita establecer manexadores de eventos ao nodo.

A opción innerHTML parece máis sinxela para contido básico, sen embargo createElement ofrece mellor capacidade de mantemento, seguridade e rendemento para manipulacións dinámicas e complexas do DOM.

Clonado de nodos

O método Node.cloneNode() permite facer un clon dun nodo. Este método devolve un duplicado do nodo dende o que se invoca.

Pode pasarse un parámetro ao método cloneNode(), que indicará se tamén se clona a subárbore. Se o parámetro é **true** clónase toda a subárbore e se é **false,** só se clona o nodo.

NOTA: cloneNode() pode dar lugar a **IDs duplicados** no DOM. Se un nodo ten id, haberá que modificar o valor na copia.

Exercicios:

1. Imaxinar que a variable **elemento** fai referencia a un elemento do DOM e text é unha variable con unha cadea de texto que inclúe etiquetas HTML. ¿Cales dos seguintes comandos farán exactamente o mesmo?:

    a. elemento.append(document.createTextNode(text));
    b. elemento.innerHTML = text;
    c. elemento.textContent = text;

2. Dada unha lista <ol> con varios elementos <li>, crea o código necesario para eliminar todos os <li> da lista.

3. Dado o seguinte código, ¿por que segue aparecendo o “Texto” despois de borrar a táboa?

<table id="taboa"> Texto <tr> <td>Test</td> </tr> </table>	let taboa = document.getElementById("taboa"); taboa.remove();

4. Crea un documento HTML que conteña un elemento <ul>. Dende JavaScript crea 4 elementos <li> e engádeos á lista <ul>, de tal forma que sexan visibles no navegador.

5. Escribe o código JavaScript para inserir “
2
3” entre os dous <li> seguintes:

```
<ul id="listaULExercicio5">
  <li id="one">1</li>
  <li id="two">4</li>
</ul>
```

6. Dado un obxecto como o seguinte:

```
let arbore = {
  Fish: {
    trout: {},
    salmon: {},
  },

  Tree: {
    Huge: {
      sequoia: {},
      oak: {},
    },
    Flowering: {
      'apple tree': {},
      magnolia: {},
    },
  },
};
```

- Fish
  - trout
  - salmon
- Tree
  - Huge
    - sequoia
    - oak
  - Flowering
    - apple tree
    - magnolia

Crea unha función **createTree(data)** que devolva unha lista ul/li coma a da imaxe da dereita, para os datos proporcionados.

7. Escribe unha función **crearCalendario(elemento, ano, mes)** que engada ao elemento pasado como parámetro un calendario do ano e mes indicados.

L	M	Me	X	V	S	D
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

O calendario debe ser unha táboa, onde cada semana é un <tr> e cada día un <td>. A cabeceira da táboa está creada con <th>.

Por exemplo, o calendario resultado de chamar á función cos seguintes parámetros vese na imaxe seguinte. Observar que se aplicaron estilos CSS para mellorar o aspecto.

crearCalendario(calendario, 2022, 11);

8. Ordena a seguinte táboa pola columna “Nome”. Escribe un código que funcione independentemente do número de filas da táboa.

```
<table id="taboaOrdenar">
   <thead>
     <tr>
       <th>Nome</th>
       <th>Apelido</th>
       <th>Idade</th>
     </tr>
   </thead>
   <tbody>
     <tr>
       <td>John</td>
       <td>Smith</td>
       <td>10</td>
     </tr>
     <tr>
       <td>Pete</td>
       <td>Brown</td>
       <td>15</td>
     </tr>
     <tr>
       <td>Ann</td>
       <td>Lee</td>
       <td>5</td>
     </tr>
   </tbody>
 </table>
```

9. Dada unha lista como a seguinte, escribe o código que engada o número de descendentes.

```
<ul id="listaAnimais">
  <li>
    Animals
    <ul>
      <li>
        Mammals
        <ul>
          <li>Cows</li>
          <li>Donkeys</li>
          <li>Dogs</li>
          <li>Tigers</li>
        </ul>
      </li>
      <li>
        Other
        <ul>
          <li>Snakes</li>
          <li>Birds</li>
          <li>Lizards</li>
        </ul>
      </li>
    </ul>
  </li>
  <li>
    Fishes
    <ul>
      <li>
        Aquarium
        <ul>
          <li>Guppy</li>
          <li>Angelfish</li>
        </ul>
      </li>
      <li>
        Sea
        <ul>
          <li>Sea trout</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

- Animals [2]
  - Mammals [4]
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other [3]
    - Snakes
    - Birds
    - Lizards
- Fishes [2]
  - Aquarium [2]
    - Guppy
    - Angelfish
  - Sea [1]
    - Sea trout

Estilos e clases

Antes de manipular dende JavaScript as clases e estilos CSS, convén recordar que hai dúas formas de aplicar estilos a un elemento:

- Crear unha **clase** CSS e engadila ao elemento: ** .** Versión recomendada.
- Escribir as propiedades directamente no atributo style: ** **

Dende JavaScript pódense modificar os estilos CSS, independentemente da forma usada para aplicalos.

Element.className é unha propiedade que permite ler/establecer o valor do atributo "class" do elemento.

NOTA: utilízase **className** como nome da propiedade en lugar de **class** por conflitos coa palabra reservada "class" en linguaxes que se utilizan para manipular o DOM.

```
<body class="main page">
  <script>
    console.log(document.body.className); // main page
  </script>
</body>
```

Observar que **className** obtén a lista de clases. Se se modifica o **className** modifícase toda a lista de clases.

Ás veces interesa agregar ou eliminar só unha clase, polo que é mellor utilizar a propiedade **Element.classList**.

Element.classList é uha propiedade de só lectura que devolve unha lista viva (DOMTokenList) cos atributos class do elemento. Aínda que a propiedade classList é de só lectura, é posible modificala usando os métodos add(), remove(), replace() e toggle().

```
<body class="main page">
  <script>
    document.body.classList.add('article');
    console.log(document.body.className); // main page article
  </script>
</body>
```

Métodos para modificar a DOMTokenList:

- DOMTokenList.add(tokens): engade os tokens á lista, omitindo os que están duplicados.
- DOMTokenList.remove(tokens): elimina os tokens especificados da lista.
- DOMTokenList.replace(oldToken, newToken): substitúe un token existente por outro novo.
- DOMTokenList.toggle(token): se o token está na lista elimínao e devolve **false**. Se o token non está na lista, engádeo e devolve **true**. O valor devolto indica se o token está na lista despois de executar o método.
- DOMTokenList.contains(token): devolve true se a lista contén o token e false en caso contrario.

A propiedade HTMLElement.style é un obxecto de tipo CSSStyleDeclaration, que contén unha lista das propiedades de estilo con valores asignados só para os atributos definidos no atributo "style" do elemento. É posible ler/modificar as propiedades CSS accedendo directamente ás propiedades coa notación **camelCase**:

```
element.style.backgroundColor = "red"
```

NOTA: observar que cando o estilo en CSS son dúas palabras unidas por guión (background-color) en JavaScript utilízase a notación camelCase.

A propiedade style só permite acceder aos estilos en liña asignados directamente a un elemento. Sen embargo, os estilos efectivos aplicados a un elemento son a consecuencia da aplicación de todas as regras CSS e herdanza. Para acceder aos estilos aplicados debe usarse Window.getComputedStyle()

```
<html lang="en">
  <head>
    <style> body { color: red; margin: 5px; font: 2rem/2 sans-serif;} </style>
  </head>
  <body>
    <script style="color: blue">
      console.log(document.body.style.color);
      console.log(document.body.style.marginTop);

      let computedStyle = getComputedStyle(document.body);
      console.log(computedStyle.marginTop);
      console.log(computedStyle.color);
      console.log(computedStyle.fontSize);
    </script>
  </body>
</html>
```

NOTAS:

- observar o valor devolto pola propiedade **computedStyle.fontSize** e comparalo co estilo CSS asignado. Fixarse que se fai o cálculo e asígnaselle o tamaño en px.
- getComputedStyle ten os valores de propiedades CSS con nomes longos (non usar os nomes abreviados). Por exemplo, a propiedade CSS **font** é o nome abreviado para propiedades como font-style, font-weight, font-size/line-height, font-family, … Sempre se debe utilizar a propiedade exacta. Usar os nomes abreviados non está estandarizado, polo que poden obterse resultados diferentes en distintos navegadores.

Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:

- JavaScript | MDN
- Introduction to the DOM - Web APIs | MDN
- Client-side web APIs - Learn web development | MDN
- JavaScript Tutorial - w3schools
- Desarrollo Web en Entorno Cliente | materials
- Javascript en español - Lenguaje JS
- The Modern JavaScript Tutorial
- Eloquent JavaScript