Resumen Tema 5 – DOM en JavaScript

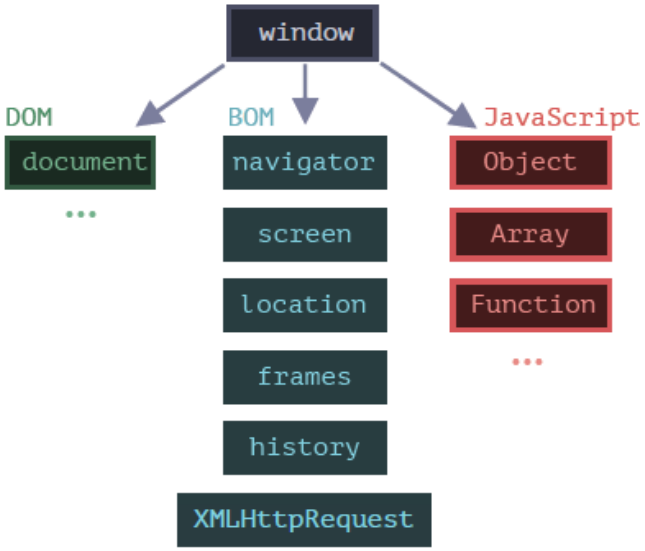
Desarrollo web en Entorno Cliente

28 DE NOVIEMBRE DE 2022
CIFP CARLOS III - CARTAGENA
SANTIAGO FRANCISCO SAN PABLO RAPOSO
2º CURSO DAW

Contenido

1.- Entorno del navegador, especificaciones.

Esto es con lo que cuenta JavaScript cuando se ejecuta en un navegador web:



- **Objeto "window" tiene dos roles**:
  - Objeto global para el código JavaScript. **Por ejemplo**:

```
function sayHi() {
    alert("Hola");
}

// Las funciones globales son métodos del objeto global:
window.sayHi();
```

  - Representa la "ventana del navegador" y proporciona métodos para controlarla. **Por ejemplo**:

```
alert(window.innerHeight); // altura interior de la ventana
```

1.1.- DOM (Modelo de Objetos del Documento).

Representa todo el contenido de la página como objetos que pueden ser modificados.

```
// cambiar el color de fondo a rojo
document.body.style.background = "red";
```

```
// deshacer el cambio después de 1 segundo
setTimeout(() => document.body.style.background = "", 1000);
```

DOM no es solo para navegadores: Por ejemplo, los scripts del lado del servidor que descargan páginas HTML y las procesan, también pueden usar DOM. Sin embargo, podrían admitir solamente parte de la especificación.

CSSOM para los estilos: CSSOM se usa junto con DOM cuando modificamos las reglas de estilo para el documento. Sin embargo, en la práctica rara vez se requiere CSSOM, porque rara vez necesitamos modificar las reglas CSS desde JavaScript.

1.2.- BOM (Modelo de Objetos del Navegador)

Son objetos adicionales proporcionados por el navegador (entorno host) para trabajar con todo excepto el documento.

- **navigator.userAgent**: acerca del navegador actual
- **navigator.platform**: acerca de la plataforma (ayuda a distinguir Windows/Linux/Mac, etc.).
- **Objeto location**: nos permite leer la URL actual y puede redirigir el navegador a una nueva.
- **Las funciones alert/confirm/prompt** también forman parte de BOM. Sin embargo, no necesitan llamarse por navigator.alert ni pertenecen al objeto document.

```
alert(location.href); // muestra la URL actual
if (confirm("Ir a wikipedia?")) { location.href = "https://wikipedia.org"; //
redirigir el navegador a otra URL
}
```

2.- Árbol del Modelo de Objetos del Documento (DOM).

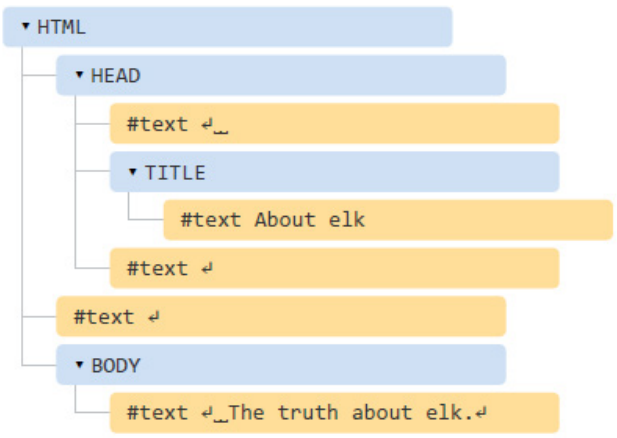
Según el Modelo de Objetos del Documento (DOM), **cada etiqueta HTML es un objeto**. Las etiquetas anidadas son llamadas "hijas" de la etiqueta que las contiene. El texto dentro de una etiqueta también es un objeto.

```
document.body.style.background = 'red'; // establece un color de fondo rojo
setTimeout(() => document.body.style.background = '', 3000); // volver atrás
```

Como se puede ver, podemos cambiar:

- Style.background
- innerHTML
- offsetWidth (ancho del nodo en píxeles).
- Etc.

Estructura del DOM:



El texto dentro de los elementos forma nodos de texto, etiquetados como #text. Un nodo de texto contiene solo una cadena. Esta puede no tener hijos y siempre es una hoja del árbol.

Hay que tener en cuenta los caracteres especiales en nodos de texto: una línea nueva: ↵ (en JavaScript se emplea \n para obtener este resultado) un espacio: ␣

Hay solo dos excepciones de nivel superior en donde no se cumple la regla de los espacios convertidos en nodos de texto:

- Los espacios y líneas nuevas antes de la etiqueta <head> son ignorados por razones históricas.
- Si colocamos algo después de la etiqueta </body>, automáticamente se sitúa dentro de body, al final, ya que, la especificación HTML necesita que todo el contenido esté dentro de la etiqueta <body>, no puedehaber espacios después de esta.

Los espacios al inicio/final de la cadena y los nodos de texto que solo contienen espacios en blanco, por lo general, están ocultos en las herramientas. De esta manera, las herramientas para desarrolladores ahorran espacio en la pantalla.

2.1.- Autocorrección.

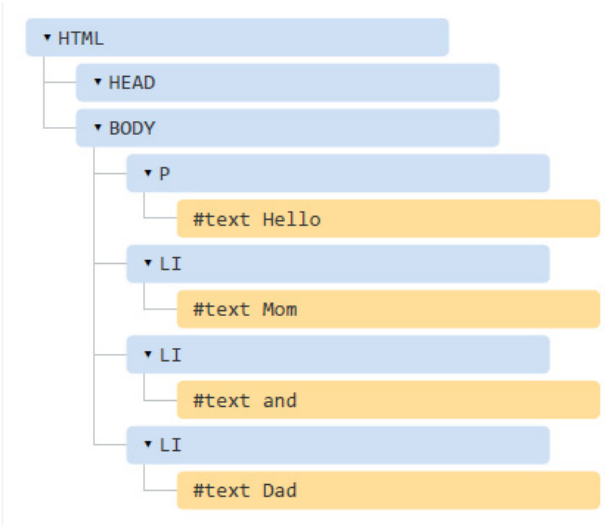
Si el navegador encuentra HTML mal escrito, lo corrige automáticamente al construir el DOM.

Por ejemplo, la etiqueta superior siempre será <html>. Incluso si no existe en el documento, ésta existirá en el DOM, puesto que, el navegador la creará. Sucede lo mismo con la etiqueta <body>.

Otro ejemplo: Un documento sin etiquetas de cierre:

```
<p>Hello<li>Mom<li>and<li>Dad
```

…se convertirá en un DOM normal a medida que el navegador lee las etiquetas y compone las partes faltantes:



Las tablas siempre tienen la etiqueta <tbody>

Un caso especial interesante son las tablas. **De acuerdo a la especificación DOM deben tener la etiqueta <tbody>, sin embargo el texto HTML puede omitirla**: el navegador crea automáticamente la etiqueta <tbody> en el DOM.

2.2.- Otro tipo de nodos: los comentarios.

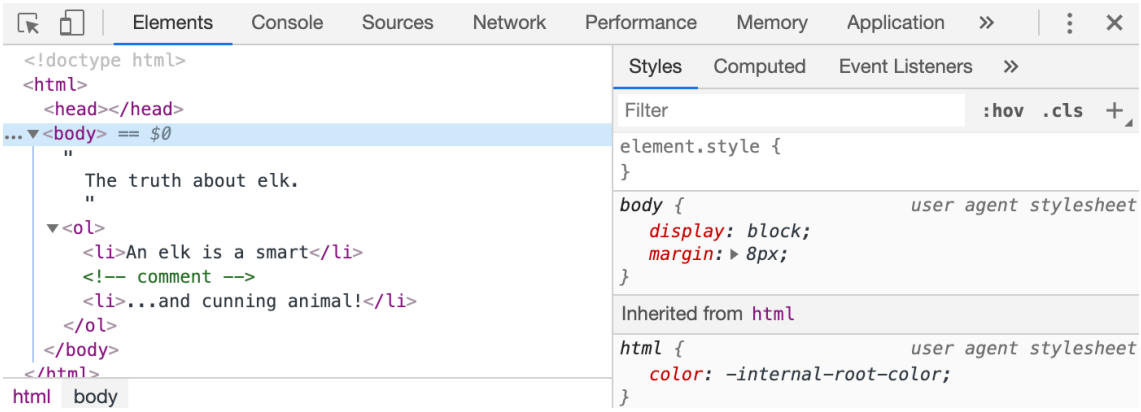
2.3.- En general, hay 12 tipos de nodos.

Hay 12 tipos de nodos. En la práctica generalmente trabajamos con 4 de ellos:

- **document** – el "punto de entrada" en el DOM.
- **nodos de elementos** – Etiquetas-HTML, los bloques de construcción del árbol.
- **nodos de texto** – contienen texto.
- **comentarios** – a veces podemos colocar información allí, no se mostrará, pero JS puede leerla desde el DOM.

2.4.- Véalo usted mismo.

Otra forma de explorar el DOM es usando la herramienta para desarrolladores del navegador. En realidad, eso es lo que usamos cuando estamos desarrollando.



La estructura DOM en la herramienta para desarrolladores está simplificada. Los nodos de texto se muestran como texto. Y absolutamente no hay nodos de texto con espacios en blanco. Esto está bien, porque la mayoría de las veces nos interesan los nodos de elementos.

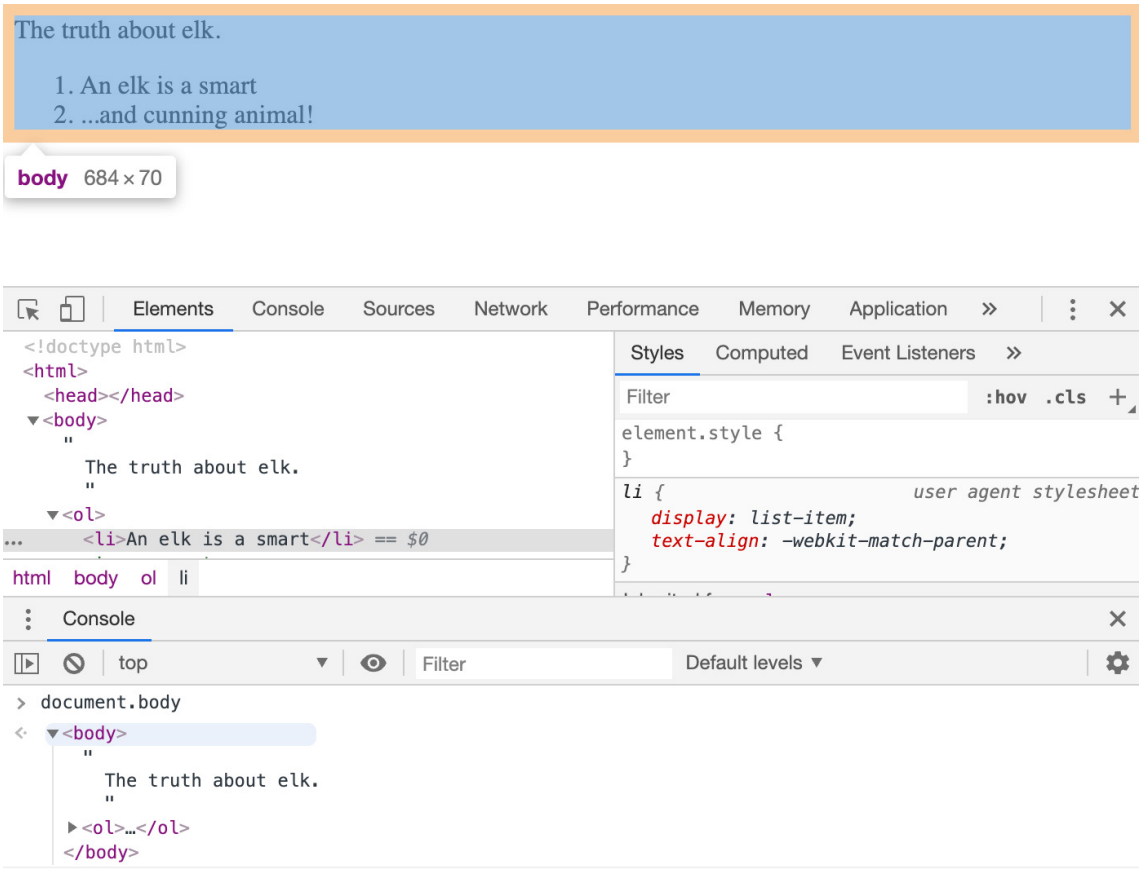
En la parte derecha de las herramientas encontramos las siguientes subpestañas:

- **Styles**: para ver los estilos que se le aplican a un determinado nodo.
- **Computed**: nos permite ver cada propiedad CSS aplicada al elemento. Para cada propiedad, podemos ver la regla que la provee (incluida la herencia CSS y demás).
- **Event Listeners**: nos ayuda a ver los listener de eventos adjuntos a elementos del DOM.

2.5.- Interacción con la consola.

Aquí hay algunos consejos para desplazarse entre la pestaña elementos y la consola.

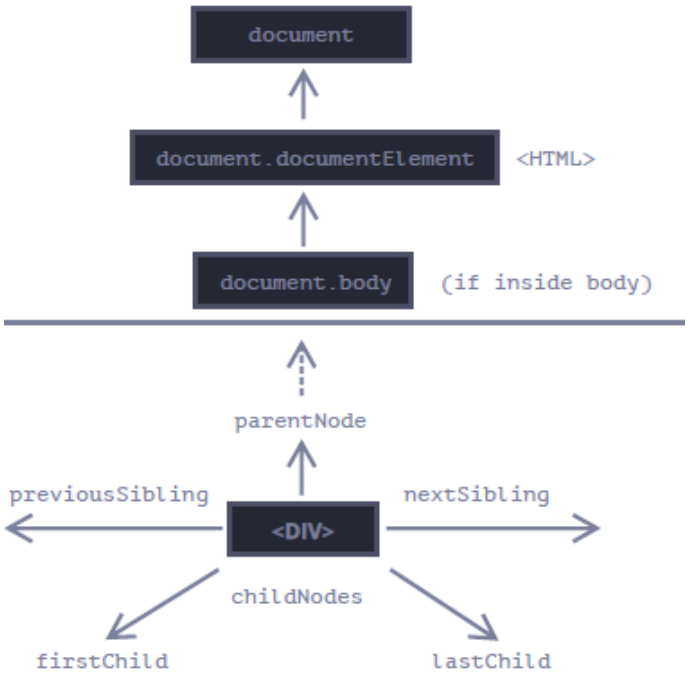
1. Seleccione un elemento con Inspeccionar elemento
2. **Presiona la tecla Esc – esto abrirá la consola** justo debajo de la pestaña de elementos.
3. Ahora el último elemento seleccionado esta disponible como $0, el seleccionado previamente es $1, etc.
    a. **Ejemplo de uso**: $0.style.background = 'red'.
4. **También hay un camino de regreso**. Si hay una variable que hace referencia a un nodo del DOM, usamos el comando **inspect(node)** en la consola para verlo en el panel de elementos.



Esto es para propósitos de depuración del curso.

3.- Recorriendo el DOM.

Todas las operaciones en el DOM comienzan con el objeto document. Este es el principal "punto de entrada" al DOM.



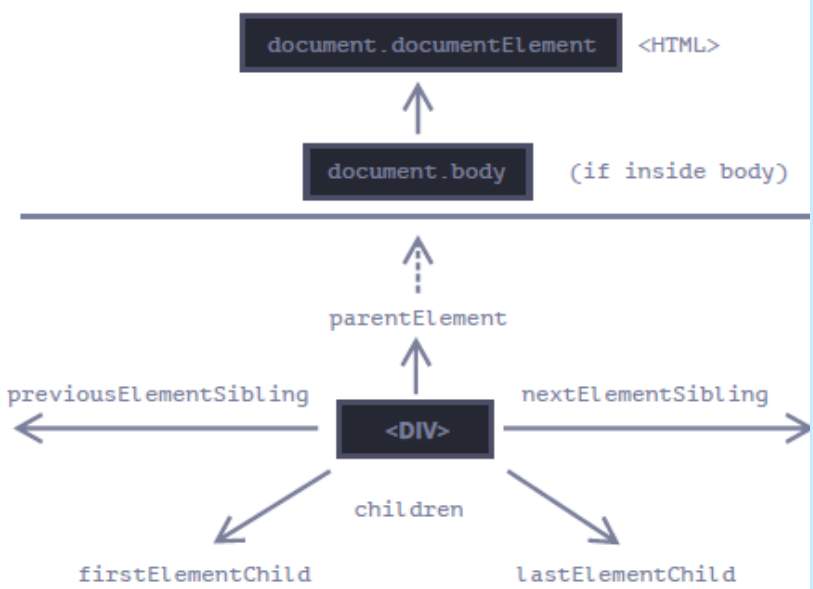
Formas de recorrer el DOM	Formas de hacerlo
Document. documentElement	Devuelve la etiqueta <html>
Document.body	Devuelve la etiqueta <body> Si el script está situado antes de escribir la etiqueta <body>, <body> puede ser null (no existir).
Document.head	• **Document.head**: devuelve la etiqueta head. • **Colección childNodes**:
Colección childNodes (otros Elementos de búsqueda).	Permite acceder a los descendientes inmediatos de un determinado elemento. **No es un array, sino un objeto especial iterable (tiene la propiedad Symbol.iterator)**. • Propiedad **firstChild** === elem.childNodes[0] • Propiedad **lastChild** === elem.childNodes[elem.childNodes.length – 1] • **Función** elem.**hasChildNodes**. Nos permite acceder a todos los nodos, incluso nodos de texto y nodos de comentarios. **Las colecciones DOM son solo de lectura**. También sus propiedades de navegación enumeradas en este apartado. **Por ejemplo**, no podemos reemplazar a un hijo por otro (si podemos acceder las propiedades de los elementos que contienen y modificarlas). **Cambiar el DOM necesita otros métodos**. Sin embargo, las colecciones DOM están vivas: cuando cambiemos algo en el DOM mediante dichos métodos, se verán reflejados los cambios en dichas colecciones (elem.childNodes).

Iterar childNodes	• **for..of**: permite recorrer los elementos (nodos hijos).
	`for (let node of document.body.childNodes) { alert(node); // enseña todos los nodos de la colección }`
	• **for..in**: **itera sobre las propiedades enumerables**. Y las colecciones tienen unas propiedades “extra” raramente usadas que normalmente no queremos obtener. **Por eso se usa más el for..of**.
	Los métodos de Array no funcionan, porque childNodes no es un Array. Sin embargo, podemos crear un Array “real” si es lo que queremos, haciendo por ejemplo:
	`Array.from(document.body.childNodes)`

Hermanos y padres	• **nextSibling**: hermano siguiente. • **previousSibling**: hermano anterior. • **parentNode**: padre inmediato.

Navegación solo por elementos

Tiene sus propiedades análogas, pero con la palabra Element. Aquí se pueden ver. Siguen la misma lógica que para el caso de los nodos.



- **children** – solo esos hijos que tienen el elemento nodo.
- **firstElementChild, lastElementChild** – el primer y el último elemento hijo.
- **previousElementSibling, nextElementSibling** – elementos vecinos.
- **parentElement** – elemento padre.
  - **Suele ser equivalente a parentNode**. Pero hay una sutil diferencia:

```
alert( document.documentElement.parentNode ); // documento
alert( document.documentElement.parentElement ); // null
```

La razón es que **el nodo raíz document.documentElement** (<html>) **tiene a document como su padre**. **Pero document no es un elemento nodo**, por lo que parentNode lo devuelve y parentElement no lo hace.

Elementos con propiedades adicionales: tablas

Table:

- **Table.rows**: colección de <tr>.
- **Table.caption/thead/tfoot**: referencias a elementos <caption>, <thead>, <tfoot>.
- Table.tBodies: la colección de elementos <tbody>. Siempre habrá al menos uno (ya que el navegador lo pondrá en el DOM).

thead, tfoot, tbody:

- tbody.rows – la colección dentro de <tr>.

tr:

- **tr.cells**: la colección de celdas <td> y <th>.
- **tr.sectionRowIndex:** la posición (índice) del <tr> dado dentro del <thead>/<tbody>/<tfoot>

	• **tr.rowIndex**: el número de <tr> en la tabla en su conjunto (incluyendo todas las filas de una tabla).
	td y th: td.cellIndex – el número de celdas dentro del adjunto <tr>.

Ejemplo de recorrido de tablas:

```
<table id="table">
    <tr>
        <td>one</td><td>two</td>
    </tr>
    <tr>
        <td>three</td><td>four</td>
    </tr>
</table>

<script>
    // seleccionar td con "dos" (primera fila, segunda columna)
    let td = table.rows[0].cells[1];
    td.style.backgroundColor = "red"; // destacarlo
</script>
```

3.1.- Resumen rápido del punto 3.

Hay dos conjuntos principales de propiedades de navegación:

- **Para todos los nodos**: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.
- **Para los nodos elementos**: parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.

4.- Buscar: getElement*, querySelector*.

Sirven para buscar elementos del DOM cuando los elementos no están cerca unos de otros.

Método	Ejemplos

Document. getElementById	
	 Elemento
	
	<script>
	// obtener el elemento
	let elem = document.getElementById('elem');
	// También podemos simplemente referirnos al elemento así:
	//elem.style.background = 'red';
	
	// hacer que su fondo sea rojo
	elem.style.background = 'red';
	</script>

La forma de nombrar directamente al elemento no siempre se puede aplicar. Un elemento con id="elem-id", al llevar guion, no se puede referenciar directamente.

La forma de nombrar directamente al elemento, **se cumple a menos que declaremos una variable de JavaScript con el mismo nombre, entonces ésta tiene prioridad**:

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

<script>
    let elem = 5; // ahora elem es 5, no una referencia
a <div id="elem">

    alert(elem); // 5
</script>
```

La forma de nombrar directamente al elemento, es una mala práctica. **Puede haber conflictos de nombres**. Además, cuando uno lee el código de JS y no tiene el HTML a la vista, no es obvio de dónde viene la variable.
En la vida real document.getElementById es el método preferente. No obstante, el id debe ser único.

elem. querySelectorAll (elem)

Devuelve todos los elementos dentro de elem que coinciden con el selector CSS dado.

Aquí buscamos todos los elementos <li> que son los últimos hijos:

```
<ul>
    <li>La</li>
    <li>prueba</li>
</ul>
<ul>
    <li>ha</li>
    <li>pasado/li>
</ul>
<script>
    let elements = document.querySelectorAll('ul > li:last-
child');

    for (let elem of elements) {
        alert(elem.innerHTML); // "prueba", "pasado"
    }
</script>
```

Devuelve **colecciones estáticas**:

```
<div>Primer div</div>

<script>
    let divs = document.querySelectorAll('div');
    alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
    alert(divs.length); // 1
</script>
```

elem. querySelector

La llamada a elem.querySelector(css) devuelve el primer elemento para el selector CSS dado.

Devuelve el mismo resultado que devolvería: Elem.querySelectorAll(css)[0]

elem.matches

Comprueba si un elemento coincide con el selector CSS.

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
    // puede ser cualquier colección en lugar de document.body.children
    for (let elem of document.body.children) {
        if (elem.matches('a[href$="zip"]')) {
            alert("La referencia del archivo: " + elem.href );
        }
    }
</script>
```

elem.closest

Devuelve el ancestro (padre) más cercano que coincide con el selector CSS. El propio elem se incluye en la búsqueda.

```
<h1>Contenido</h1>

<div class="contents">
    <ul class="book">
        <li class="chapter">Capítulo 1</li>
        <li class="chapter">Capítulo 2</li>
    </ul>
</div>

<script>
    let chapter = document.querySelector('.chapter'); // LI

    alert(chapter.closest('.book')); // UL
    alert(chapter.closest('.contents')); // DIV

    alert(chapter.closest('h1')); // null (porque h1 no es un ancestro)
</script>
```

getElementsBy*

- **elem.getElementsByTagName(tag)** busca elementos con la etiqueta dada y devuelve una colección conellos. El parámetro tag también puede ser un asterisco "*" para “cualquier etiqueta”.
- **elem.getElementsByClassName(className**) devuelve elementos con la clase dada.
- **document.getElementsByName(name)**: devuelve elementos con el atributo name dado, en todo el documento. Muy raramente usado.

Estos métodos devuelven colecciones, no elementos individuales. Además, estas colecciones **SON COLECCIONES VIVAS**. Es decir, siempre reflejan el estado actual del documento y se auto-actualizan cuando cambian.

```
<div>Primer div</div>

<script>
    let divs = document.getElementsByTagName('div');
    alert(divs.length); // 1
</script>

<div>Segundo div</div>

<script>
    alert(divs.length); // 2
</script>
```

4.1.- Resumen rápido del punto 4.

Método	Busca por...	¿Puede llamar a un elemento?	¿Vivo?
querySelector	selector CSS	✓	-
querySelectorAll	selector CSS	✓	-
getElementById	id	-	-
getElementsByName	name	-	✓
getElementsByTagName	etiqueta o '*'	✓	✓
getElementsByClassName	class	✓	✓

Aparte de eso:

- **elem.matches(css):** para comprobar si elem coincide con el selector CSS dado.
- **elem.closest(css)**: para buscar el ancestro más cercano que coincida con el selector CSS dado. El propio elem también se comprueba.

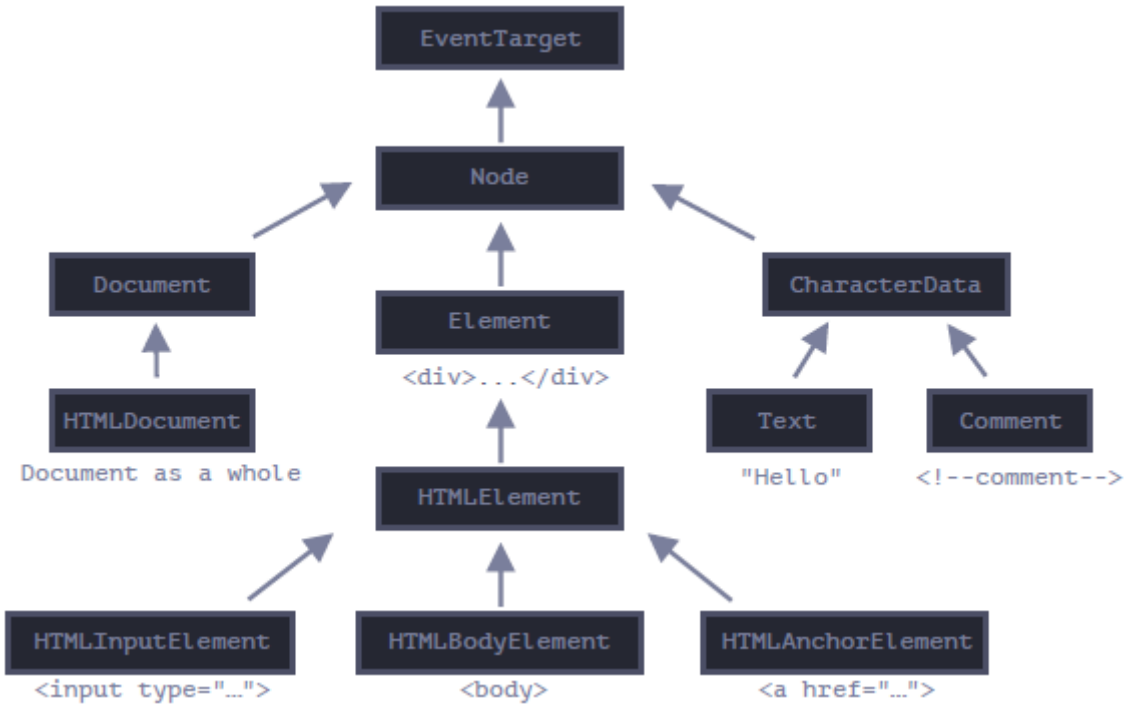
Un elemento más a mencionar:

- **elemA.contains(elemB):** devuelve true si elemB está dentro de elemA (un descendiente de elemA) ocuando elemA==elemB.

5.- Propiedades del nodo: tipo, etiqueta y contenido.

Los diferentes nodos DOM pueden tener diferentes propiedades. Por ejemplo, un nodo de elemento correspondiente a la etiqueta <a> tiene propiedades relacionadas con el enlace.

Sin embargo, **los nodos de texto no son lo mismo que los nodos de elementos**. Pero **también hay propiedades y métodos comunes entre todos ellos**.



La raíz de la jerarquía es EventTarget, que es heredada por Node, y otros nodos DOM heredan de él.

- **EventTarget**: es una clase raíz "abstracta". Los objetos de esta clase nunca se crean. Sirve como base para soportar los eventos sobre los objetos.
- **Node**: también es una clase "abstracta" (no se crean objetos tipo Node directamente), sirve como base para los nodos DOM.
    - **Proporciona la funcionalidad del árbol principal**: parentNode, nextSibling, childNodes y demás.
- **Document**: hereda de Node directamente. Por razones históricas, heredado a menudo por HTMLDocument, **es el documento como un todo**.
    - El **objeto global document pertenece** exactamente **a esta clase**.

Por ejemplo: la cadena de herencia del objeto HTMLInputElement sería: HTMLInputElement > HTMLElement > Element > Node > EventTarget > Object (de cuya clase se heredan métodos de "objeto simple" como *hasOwnProperty*).

Para ver el nombre de la clase del nodo DOM:

```
alert( document.body.constructor.name ); // HTMLBodyElement
```

Otra alternativa:

```
alert( document.body ); // [object HTMLBodyElement]
```

También Podemos usar instanceof para verificar la herencia:

```
alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
```

```
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Los nodos DOM son objetos regulares de JavaScript. Usan clases basadas en prototipos parala herencia. Esto se puede ver si escribimos en consola ***console.dir(elem)***. Podremos ver HTMLElement.prototype, Element.prototype, etc.

Console.dir(elem) vs console.log(elem):

- **console.log(elem)** muestra el árbol DOM del elemento.
- **console.dir(elem)** muestra el elemento como un objeto DOM, es bueno para explorar sus propiedades.

IDL en la especificación: lenguaje especial de descripción de interfaces. Ejemplo:

```
// Definir HTMLInputElement
// Los dos puntos ":" significan que HTMLInputElement hereda de HTMLElement
interface HTMLInputElement: HTMLElement {
    // aquí van las propiedades y métodos de los elementos <input>

    // "DOMString" significa que el valor de una propiedad es un string
    attribute DOMString accept;
    attribute DOMString alt;
    attribute DOMString autocomplete;
    attribute DOMString value;

    // Propiedad de valor booleano (true/false)
    attribute boolean autofocus;
    ...
    // ahora el método: "void" significa que el método no devuelve ningún
valor
    void select();
    ...
}
```

5.1.- La propiedad “nodeType” de elem.

Forma anticuada de obtener el “tipo” de un nodo DOM. (Aviso: nodeType es de solo lectura).

- elem.nodeType == 1 para nodos de elementos,
- elem.nodeType == 3 para nodos de texto,
- elem.nodeType == 9 para el objeto de documento,
- hay algunos otros valores en la especificación.

Por ejemplo:

```
<body>
    <script>
    let elem = document.body;

    // vamos a examinar: ¿qué tipo de nodo es elem?
    alert(elem.nodeType); // 1 => elemento

    // Y el primer hijo es...
    alert(elem.firstChild.nodeType); // 3 => texto

    // para el objeto de tipo documento, el tipo es 9
```

```
    alert( document.nodeType ); // 9
    </script>
</body>
```

En los scripts modernos, podemos usar instanceof, que es mucho más cómodo y fácil en general.

5.2.- Tag: elem.nodeName y elem.tagName.

Ambas sirven para leer el nombre de etiqueta de un nodo. Pero:

- La propiedad **tagName** existe solo para los nodos Element.
- El **nodeName** se define para cualquier Node:
  - para los elementos, significa lo mismo que tagName.
  - para otros tipos de nodo (texto, comentario, etc.) tiene una cadena con el tipo de nodo.

En otras palabras, **tagName solo es compatible con los nodos de elementos** (ya que se origina en la clase Element), mientras que **nodeName puede decir algo sobre otros tipos de nodos**.

El nombre de la etiqueta siempre está en mayúsculas, excepto en el modo XML.

5.3.- innerHTML: los contenidos.

La propiedad **innerHTML permite obtener el HTML dentro del elemento como un string**. También podemos modificarlo. Además, **podemos intentar insertar HTML no válido**, el navegador corregirá nuestros errores:

```
<body>

    <script>
        document.body.innerHTML = '<b>prueba'; // olvidé cerrar la etiqueta
        alert( document.body.innerHTML ); // <b>prueba</b> (arreglado)
    </script>

</body>
```

Si se insertan etiquetas <script> haciendo uso de innerHTML, se convierte en parte de HTML, pero no se ejecuta.

<u>“innerHTML+=” sobreescribe el código completo:</u>

```
chatDiv.innerHTML += "<div>Hola<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "¿Cómo vas?";
```

En otras palabras, innerHTML+= hace esto:

- Se elimina el contenido antiguo.
- En su lugar, se escribe el nuevo innerHTML (una concatenación del antiguo y el nuevo).

Como el contenido se "pone a cero" y se reescribe desde cero, todas las imágenes y otros recursos se volverán a cargar.

También hay otros efectos secundarios. Por ejemplo, si el texto existente se seleccionó con el mouse, la mayoría de los navegadores eliminarán la selección al reescribir innerHTML. Todo lo que hubiera antes, ahora ya aparece como por primera vez.

Afortunadamente, hay otras formas de agregar HTML además de innerHTML, y las estudiaremos pronto.

5.4.- outerHTML: HTML completo del elemento.

La propiedad outerHTML contiene el HTML completo del elemento. Eso es **como innerHTML más el elemento en sí**.

Cuidado: a diferencia de innerHTML, escribir en outerHTML no cambia el elemento. En cambio, lo reemplaza en el DOM. Sí, suena extraño, y es extraño, por eso hacemos una nota aparte al respecto aquí. Echa un vistazo.

```
<div>¡Hola, mundo!</div>

<script>
    let div = document.querySelector('div');

    // reemplaza div.outerHTML con <p>...</p>
    div.outerHTML = '<p>Un nuevo elemento</p>'; // (*)

    // ¡Guauu! ¡'div' sigue siendo el mismo!
    alert(div.outerHTML); // <div>¡Hola, mundo!</div> (**)
</script>
```

Entonces, lo que sucedió en div.outerHTML=... es:

- div fue eliminado del documento.
- Otro fragmento de HTML Un nuevo elemento se insertó en su lugar.
- div todavía tiene su antiguo valor. El nuevo HTML no se guardó en ninguna variable.

Es muy fácil cometer un error aquí: modificar div.outerHTML y luego continuar trabajando con div como si tuviera el nuevo contenido. Pero no es así. **Esto es correcto para innerHTML, pero no para outerHTML**.

5.5.- nodeValue/data: contenido del nodo de texto.

Los nodos de texto, tienen su contraparte: propiedades **nodeValue** = innerHTML y **data** = outerHTML.

```
<body>
    Hola
    <!-- Comentario -->
    <script>
        let text = document.body.firstChild;
        alert(text.data); // Hola

        let comment = text.nextSibling;
```

```
        alert(comment.data);  //  Comentario
    </script>
</body>
```

Para los nodos de texto podemos imaginar una razón para leerlos o modificarlos, pero… ¿por qué modificar los comentarios**? A veces, los desarrolladores incorporan información o instrucciones de plantilla en HTML, así**:

```
<!--  if  isAdmin  -->
    <div>¡Bienvenido,  administrador!</div>
<!--  /if  -->
```

…Entonces JavaScript puede leerlo desde la propiedad **data** y procesar las instrucciones integradas.

5.6.- textContent: texto puro.

Proporciona acceso al texto dentro del elemento: solo texto. **Ejemplo**:

```
<div  id="news">
    <h1>¡Titular!</h1>
    <p>¡Los  marcianos  atacan  a  la  gente!</p>
</div>

<script>
    //  ¡Titular!  ¡Los  marcianos  atacan  a  la  gente!
    alert(news.textContent);
</script>
```

Escribir en textContent es mucho más útil, porque permite escribir texto de “forma segura”.

```
<div  id="elem1"></div>
<div  id="elem2"></div>

<script>
    let  name  =  prompt("¿Cuál  es  tu  nombre?",  "<b>¡Winnie-Pooh!</b>");

    elem1.innerHTML  =  name;
    elem2.textContent  =  name;
</script>
```

De esta forma, mientras que en el innerHTML se interpretarían las etiquetas b, en textContent no.

5.7.- La propiedad “hidden”.

El atributo “hidden” y la propiedad DOM especifican si el elemento es visible o no. Es equivalente a style=”display:none”, pero es más corto. Ejemplo de uso para un elemento parpadeante:

```
<div  id="elem">Un  elemento  parpadeante</div>

<script>
    //elem.hidden  =  true;
    setInterval(()  =>  elem.hidden  =  !elem.hidden,  1000);
</script>
```

5.8.- Más propiedades.

Los elementos Dom también tienen propiedades adicionales, en particular aquellas que dependen de la clase:

- **value**: el valor para <input>, <select> y <textarea> (HTMLInputElement, HTMLSelectElement…).
- **href**: el "href" para <a href="..."> (HTMLAnchorElement).
- **id**: el valor del atributo "id", para todos los elementos (HTMLElement).

5.9.- Resumen rápido del apartado 5.

- **nodeType**: podemos usarla para ver si un nodo es un texto o un elemento. Solo lectura. Tiene un valor numérico:
  - 1 para elementos, 3 para nodos de texto y algunos otros para otros tipos de nodos.
- **instanceof**: lo mismo que nodeType, pero más sencillo.
- **nodeName/tagName**: para los nodos que no son elementos, nodeName describe lo que es. Solo lectura.
- **innerHTML**: devuelve el contenido HTML del elemento. Puede modificarse.
- **outerHTML**: devuelve el HTML completo del elemento, elemento incluido.
  - Una operación de escritura en elem.outerHTML no toca a elem en sí. En su lugar, se reemplaza con el nuevo HTML en el contexto externo.
- **nodeValue (=innerHTML)/data (=outerHTML)**: el contenido de un nodo que no es un elemento (text, comment). Estos dos son casi iguales, usualmente usamos data. Puede modificarse.
- **textContent**: el texto dentro del elemento. Escribir en él coloca el texto dentro del elemento. Puede insertar de forma segura texto generado por el usuario y protegerse de inserciones HTML no deseadas.
- **hidden**: cuando se establece en true, hace lo mismo que CSS display:none.

Sin embargo, **los atributos HTML y las propiedades DOM no siempre son iguales**, como veremos en el próximo capítulo.

6.- Atributos y propiedades.

Para los nodos de elementos, **la mayoría de los atributos HTML estándar se convierten automáticamente en propiedades de los objetos DOM**. **¡Pero el mapeo de propiedades y atributos no es uno a uno!**.

6.1.- Propiedades DOM.

Podemos crear nuestras propiedades y métodos DOM a un elemento. Por ejemplo:

```
document.body.myData = {
    name: 'Cesar',
    title: 'Emperador'
};

alert(document.body.myData.title); // Emperador
```

Agregando un método:

```
document.body.sayTagName = function() {
    alert(this.tagName);
};

document.body.sayTagName(); // BODY (el valor de 'this' en el método es
document.body)
```

También podemos modificar prototipos incorporados como Element.prototype y agregar nuevos métodos a todos los elementos de un mismo tipo de elemento:

```
Element.prototype.sayHi = function() {
    alert(`Hola, yo soy ${this.tagName}`);
};

document.documentElement.sayHi(); // Hola, yo soy HTML
document.body.sayHi(); // Hola, yo soy BODY
```

6.2.- Atributos HTML.

Cuando un elemento tiene id o cualquier otro atributo estándar, **se crea la propiedad correspondiente**. **Pero eso no sucede si el atributo no es estándar**.

```
<body id="test" something="non-standard">
    <script>
        alert(document.body.id); // prueba
        // el atributo no estándar no produce una propiedad
        alert(document.body.type); // undefined
    </script>
</body>
```

Por ejemplo, "type" es estándar para <input> (HTMLInputElement), pero no para <body>. Los atributos estándar se describen en la especificación para la clase del elemento correspondiente.

Para poder acceder a atributos no estándares para una determinada etiqueta:

- **elem.attributes**: es una colección que contiene todos los atributos de un elemento.
- **elem.hasAttribute(nombre)** – comprueba si existe.
- **elem.getAttribute(nombre)** – obtiene el valor.
- **elem.setAttribute(nombre, valor)** – establece el valor.
- **elem.removeAttribute(nombre)** – elimina el atributo.

Además, los **atributos HTML** (a diferencia de las propiedades del DOM), **tienen las siguientes características**:

- Su **nombre no distingue entre mayúsculas y minúsculas** (id es igual a ID).
- Sus **valores son siempre strings**.

```
<body>
    <div id="elem" about="Elephant"></div>

    <script>
        alert( elem.getAttribute('About') ); // (1) 'Elephant', leyendo
```

```
        elem.setAttribute('Test', 123); // (2), escribiendo, podemos escribir
números, pero él automáticamente lo convertirá a string

        alert( elem.outerHTML ); // (3), ver si el atributo está en HTML (sí)

        for (let attr of elem.attributes) { // (4) listar todo
            alert( `${attr.name} = ${attr.value}` );
        }
    </script>
</body>
```

6.3.- Sincronización de propiedad y atributo.

Cuando cambia un atributo estándar, la propiedad correspondiente se actualiza automáticamente, y (con algunas excepciones) viceversa. **Ejemplo de esas excepciones donde solo se sincroniza del atributo a la propiedad, pero no de regreso**:

```
<input>

<script>
    let input = document.querySelector('input');

    // atributo -> propiedad
    input.setAttribute('value', 'text');
    alert(input.value); // text

    // NO propiedad -> atributo
    input.value = 'newValue';
    alert(input.getAttribute('value')); // text (¡no actualizado!)
</script>
```

Esa "característica" en realidad puede ser útil, porque las acciones del usuario pueden conducir a cambios de value**, y luego, si queremos recuperar el valor "original" de HTML, está en el atributo**.

6.5.- Las propiedades DOM tienen tipo.

Las propiedades DOM no siempre son strings. Por ejemplo, la propiedad input.checked (para casillas de verificación) es un booleano:

```
<input id="input" type="checkbox" checked> checkbox

<script>
    alert(input.getAttribute('checked')); // el valor del atributo es: string
vacía
    alert(input.checked); // el valor de la propiedad es: true
</script>
```

Otro ejemplo ocurre con los estilos CSS:

```
<div id="div" style="color:red;font-size:120%">Hola</div>

<script>
    // string
    alert(div.getAttribute('style')); // color:red;font-size:120%

    // object
    alert(div.style); // [object CSSStyleDeclaration]
    alert(div.style.color); // red
</script>
```

Muy raramente, incluso si un tipo de propiedad DOM es un string, puede diferir del atributo. Por ejemplo, la propiedad DOM href siempre es una URL completa, incluso si el atributo contiene una URL relativa o solo un #hash.

```
<a id="a" href="#hola">link</a>
<script>
    // atributo
    alert(a.getAttribute('href')); // #hola

    // propiedad
    alert(a.href ); // URL completa de http://site.com/page#hola
</script>
```

Si necesitamos el valor de href o cualquier otro atributo exactamente como está escrito en el HTML, podemos usar getAttribute.

6.6.- Atributos no estándar, dataset.

A veces, los atributos no estándar **se utilizan para pasar datos personalizados de HTML a JavaScript**, o para "marcar" elementos HTML para JavaScript.

```
<!-- marque el div para mostrar "nombre" aquí -->
<div show-info="nombre"></div>
<!-- y "edad" aquí -->
<div show-info="edad"></div>

<script>
  // el código encuentra un elemento con la marca y muestra lo que se
solicita
    let user = {
        nombre: "Pete",
        edad: 25
    };

    for(let div of document.querySelectorAll('[show-info]')) {
        // inserta la información correspondiente en el campo
        let field = div.getAttribute('show-info');
        div.innerHTML = user[field]; // primero Pete en "nombre", luego 25 en
"edad"
    }
</script>
```

También se puede usar como método para cambiar el CSS de un elemento en función de su estado:

```
<style>
    /* los estilos se basan en el atributo personalizado "order-state" */
    .order[order-state="nuevo"] {
        color: green;
    }

    .order[order-state="pendiente"] {
        color: blue;
    }

    .order[order-state="cancelado"] {
        color: red;
    }
</style>

<div class="order" order-state="nuevo">
    Un nuevo pedido.
```

```
</div>

<div  class="order"  order-state="pendiente">
    Un  pedido  pendiente.
</div>

<div  class="order"  order-state="cancelado">
    Un  pedido  cancelado
</div>
```

¿Por qué hacerlo así y no con clases? Porque un atributo es más conveniente de administrar

```
//  un  poco  más  simple  que  eliminar/agregar  clases
div.setAttribute('order-state',  'cancelado');
```

Para evitar problemas con futuros estándares HTML de que este incluya esos atributos que nosotros habíamos creado, **se utiliza la nomenclatura "data-*"** para nombrar a dichos atributos.

Todos los atributos que comienzan con "data-" están reservados para el uso de los programadores. Están disponibles en la propiedad dataset.

Por ejemplo: **si** un **elem tiene** un **atributo** llamado "**data-about**", **está disponible como elem.dataset.about**.

Los atributos de varias palabras como data-order-state **se convierten en camel-case: dataset.orderState**.

6.7.- Resumen rápido del apartado 6.

- **Atributos**: es lo que está escrito en HTML.
- **Propiedades**: es lo que hay en los objetos DOM.

Una pequeña comparación:

	Propiedades	Atributos
Tipo	Cualquier valor, las propiedades estándar tienen tipos descritos en la especificación	Un string
Nombre	El nombre distingue entre mayúsculas y minúsculas	El nombre no distingue entre mayúsculas y minúsculas

Los métodos para trabajar con atributos son:

- **elem.hasAttribute(nombre)** – para comprobar si existe.
- **elem.getAttribute(nombre)** – para obtener el valor.
- **elem.setAttribute(nombre, valor)** – para dar un valor.
- **elem.removeAttribute(nombre)** – para eliminar el atributo.
- **elem.attributes**: es una colección de todos los atributos.

Para la mayoría de las situaciones, es preferible usar las propiedades DOM. Deberíamos referirnos a los atributos solo cuando las propiedades DOM no nos convienen. **Por ejemplo**:

- **Necesitamos un atributo no estándar**. Pero si comienza con data-, entonces deberíamos usar dataset.
- **Queremos leer el valor "como está escrito" en HTML**. El valor de la propiedad DOM puede ser diferente.
    - **Por ejemplo**: la propiedad href siempre es una URL completa, y es posible que queramos obtener el valor "original ".

7.- Modificando el documento.

La modificación del DOM es la clave para crear páginas "vivas", dinámicas. Aquí veremos cómo crear nuevos elementos "al vuelo" y modificar el contenido existente de la página.

<table>
<tr><th>Operaciones</th><th>Métodos disponibles/ejemplos</th></tr>
<tr><td rowspan="2">Métodos para crear nuevos nodos</td><td>Con esto no es suficiente para insertarlo en el HTML que el usuario puede ver.</td></tr>
<tr><td>• document.createElement(tag) – crea un elemento con la etiqueta HTML dada
• document.createTextNode(value) – crea un nodo de texto (raramente usado)
• elem.cloneNode(deep) – clona el elemento. Si deep==true, lo clona con todos sus descendientes.</td></tr>
<tr><td rowspan="3">Inserción y eliminación de código (nodos o strings directamente)</td><td>Con esto sí hacemos que sea visible para el usuario en el HTML.
Podremos insertar los elementos que creemos con los métodos para crear nuevos nodos.</td></tr>
<tr><td>• node.append(...nodes or strings) – inserta en node, al final.
<pre><script>
    let div = document.createElement('div');
    div.className = "alert";
    div.innerHTML = "¡Hola! Usted ha leído un importante mensaje.";

    document.body.append(div);
</script></pre>• node.prepend(...nodes or strings) – inserta en node, al principio
• node.before(...nodes or strings) –- inserta inmediatamente antes de node
• node.after(...nodes or strings) –- inserta inmediatamente después de node</td></tr>
</table>

	• **node.replaceWith**(...nodes or strings) –- reemplaza node
	• **node.remove()** –- quita el node.
	
	Los **strings** de texto son **insertados "como texto"**. Esto es: no podremos insertar código directamente.
	Por ejemplo: mediante: div.before(' Hola ', document.createElement('hr')).
	
	Todos los métodos de inserción automáticamente quitan el nodo del lugar viejo.
	`<div id="first">Primero</div>`
	`<div id="second">Segundo</div>`
	`<script>`
	`  // no hay necesidad de llamar "remove"`
	`  second.after(first); // toma #second y después inserta #first`
	`</script>`

Inserción y eliminación: Métodos "de vieja escuela"	Todos estos métodos **devuelven el node** insertado/quitado. Pero lo usual es que el valor no se use y solo ejecutemos el método. • **parent.appendChild**(node): Añade node como último hijo de parentElem • **parent.insertBefore**(node, nextSibling): Inserta node antes de nextSibling dentro de parentElem. • **parent.removeChild**(node): reemplaza oldChild con node entre los hijos de parentElem • **parent.replaceChild**(newElem, node): quita node de parentElem (asumiendo que node es su hijo).
elem. insertAdjacentHTML y similares	Esto **permite introducir código HTML directamente**, a diferencia de node.append y sus variantes. • "beforebegin" – inserta html inmediatamente antes de elem • "afterbegin" – inserta html en elem, al principio • "beforeend" – inserta html en elem, al final • "afterend" – inserta html inmediatamente después de elem También hay métodos similares, **elem.insertAdjacentText y elem.insertAdjacentElement, que insertan strings de texto y elementos**, pero son raramente usados. En su lugar, se usan los métodos para insertar nuevos strings/nodos.
document.write	**Para agregar HTML a la página antes de** que haya terminado de **cargar**. **Desventaja**: **después** de que la página fue cargada tal llamada **borra el documento**. Puede verse principalmente en scripts viejos. **Ventaja**: cuando es llamado document.write mientras el navegador está leyendo el HTML entrante ("parsing"), y escribe algo, el navegador lo consume como si hubiera estado inicialmente allí, en el texto HTML. **Así que funciona muy rápido,**

porque no hay una "modificación de DOM" Escribe sin ni si quiera estar construido aún el DOM.

7.1.- DocumentFragment.

Sirve como contenedor para trasladar listas de nodos, por ejemplo, para insertar varios nodos de golpe.

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>
```

Aunque podemos devolver un array de nodos:

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append +  el operador "..." = ¡amigos!
</script>
```

Mencionamos DocumentFragment principalmente porque hay algunos conceptos asociados a él, como el elemento template, que cubriremos mucho después.

8.- Estilos y clases.

Por lo general, hay dos formas de dar estilo a un elemento:

1. **Crear una clase css y agregarla**:
2. **Escribir las propiedades directamente en style**: .

JavaScript puede modificar ambos, clases y las propiedades de style. Pero preferimos usar clases css en lugar de style. Solo debemos usar "style" si las clases no pueden manejarlo.

Ejemplo: “style” es aceptable si nosotros calculamos las coordenadas de un elemento dinámicamente.

```
let top = /* cálculos complejos */;
let left = /* cálculos complejos */;

elem.style.left = left; // ej. '123px', calculado en tiempo de ejecución
elem.style.top = top; // ej. '456px'
```

Para otros casos como convertir un texto en rojo, agregar un icono de fondo. **Escribir eso en CSS y luego agregar la clase** (JavaScript puede hacer eso), **es más flexible y más fácil** de mantener.

8.1.- className y classList.

Cambiar una clase es una de las acciones más utilizadas.

El elem.className corresponde al atributo "class".

```
<body class="main page">
  <script>
    alert(document.body.className); // página principal
  </script>
</body>
```

Si asignamos algo a elem.className, **reemplaza toda la cadena de clases**. A veces es lo que necesitamos, pero a menudo queremos agregar o eliminar una sola clase. **Hay otra propiedad para eso**: elem.classList.

- **elem.classList.add(“class”)**: añadir.
- **elem.classList.remove(“class”)**: eliminar.
- **elem.classList.toggle(“class”)**: alternar.
- **elem.classList.contains(“class”)**: verifica si tiene la clase dada..

```
<body class="main page">
  <script>
    // agregar una clase
    document.body.classList.add('article');

    alert(document.body.className); // clase "article" de la página principal
  </script>
</body>
```

8.2.- style de un elemento.

La propiedad elem.style es un objeto que corresponde a lo escrito en el atributo "style". Establecer elem.style.width="100px" es equivalente a tener en el atributo style una cadena con width:100px.

Para propiedades de varias palabras se usa camelCase:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Por ejemplo:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Propiedades con prefijos del navegador: como -moz-border-radius, -webkit-border-radius: también siguen la misma regla del camelCase. Además, un guión significa mayúscula:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

8.3.- Reseteando la propiedad "style".

A veces podemos querer asignar una propiedad de estilo y luego removerla. **Ejemplo**:

```
// si ejecutamos este código, el <body> parpadeará
document.body.style.display = "none"; // ocultar

setTimeout(() => document.body.style.display = "", 1000); // volverá a lo normal
```

En lugar de hacer eso, podemos usar **elem.style.removeProperty('style property')**. **Ejemplo**:

```
document.body.style.background = 'red'; //establece background a rojo

setTimeout(() => document.body.style.removeProperty('background'), 1000); // quitar
background después de 1 segundo
```

Reescribir todo usando style.cssText: Normalmente, podemos usar style.* para asignar propiedades de estilo individuales (como hemos visto hasta ahora). **Sin embargo, no podemos establecer todo el estilo de golpe** como div.style="color: red; width: 100px", **porque div.style es un objeto y es solo de lectura**. Para ello, hay una propiedad especial: **style.cssText**.

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

<script>
  // podemos establecer estilos especiales con banderas como "important"
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;

  alert(div.style.cssText);
</script>
```

¡Cuidado! Cualquier asignación a style.cssText remueve todos los estilos, es decir, los reemplaza. **Por ello**, **es rara vez usada**.

8.4.- Cuidado con las unidades CSS.

No olvidar agregar las unidades CSS a los valores.

Por ejemplo, nosotros no debemos establecer elem.style.top a 10, sino más bien a 10px. De lo contrario no funcionaría:

```
<body>
  <script>
    // ¡no funciona!
    document.body.style.margin = 20;
    alert(document.body.style.margin); // '' (cadena vacía, la asignación es ignorada)

    // ahora agregamos la unidad CSS (px) y esta sí funciona
    document.body.style.margin = '20px';
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
  </script>
</body>
```

Tenga en cuenta: **el navegador "desempaqueta" la propiedad *style.margin*** en las últimas lineas e **infiere *style.marginLeft* y *style.marginTop*** de eso.

8.5.- Estilos calculados: getComputedStyle.

Ya hemos aprendido a modificar estilos. Pero… **con getComputedStyle podremos leer cascadas de CSS (incluyendo estilos + clases aplicadas)**.

Por ejemplo: aquí no podemos ver el margen aplicado en style a la etiqueta body:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  El texto en rojo
  <script>
    alert(document.body.style.color); // vacío
    alert(document.body.style.marginTop); // vacío
  </script>
</body>
```

Pero… ¿Y si necesitamos incrementar el margen en 20 px? Necesitaremos leer el margen actual para que la suma de 25 px.

Para ello, tenemos **getComputedStyle(element, [pseudo])**, donde:

- **element**: es el elemento del cual se va a leer el valor.
- **pseudo**: por si quieres especificar un pseudoelemento, por ejemplo, ::before. Una cadena vacía o sin argumento significa el elemento mismo.

El resultado de este método es un objeto con estilos, **como elem.style**, pero ahora con respecto a todas las clases CSS.

Por ejemplo: vamos a leer los valores del margen y el color:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>

  <script>
    let computedStyle = getComputedStyle(document.body);
```

```
    // ahora podemos leer los márgenes y el color de ahí

    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>

</body>
```

Dos conceptos CSS: valores calculados y valores resueltos:

- **Valores calculados**: valor final de aplicar todas las reglas y herencias CSS, como resultado de la cascada CSS.
- **Valores resueltos**: es la que finalmente se aplica al elemento. Valores como 1em o 125% son relativos. El navegador toma el valor calculado y hace que todas las unidades sean fijas y absolutas, por ejemplo: height:20px o font-size:16px

getComputedStyle fue creado para obtener los valores calculados, pero los valores resueltos son muchos más convenientes, y el estándar cambió. Por tanto, **hoy en día devuelve los valores resueltos**.

El método getComputedStyle requiere el nombre completo de la propiedad:

Siempre deberíamos preguntar por la propiedad exacta que queremos, **como paddingLeft o marginTop o borderTopWidth, en lugar de padding o border** a secas. De lo contrario, no se garantiza el resultado correcto.

Los estilos aplicados a los enlaces :visited están ocultos:

Los enlaces visitados pueden ser coloreados usando la pseudo-clase:visited de CSS. Pero getComputedStyle (es decir, **JavaScript**) **no proporciona acceso a ese color**, porque de lo contrario **una página cualquiera podría averiguar si el usuario visitó un enlace creándolo en la página y verificar los estilos**.

También hay una lim**itación en CSS que prohíbe la aplicación de estilos de cambio de geometría en :visited**. Eso es **para** garantizar **que no haya forma para que una página maligna pruebe si un enlace fue visitado y vulnere la privacidad**.

8.6.- Resumen rápido del apartado 8.

Para manejar clases, hay dos propiedades del DOM:

- **className**: el valor de la cadena, perfecto para manejar todo el conjunto de clases.
- **classList**: el objeto con los **métodos**: add/remove/toggle/contains, perfecto para clases individuales.

Para cambiar los estilos:

- **La propiedad style es un objeto con los estilos en camelcase**. Leer y escribir tiene el mismo significado que modificar propiedades individuales en el atributo "style". Para ver **cómo aplicar important** y otras cosas raras, hay una lista de métodos en MDN.

- **La propiedad style.cssText** corresponde a **todo el atributo "style"**, la cadena completa de estilos.

Para leer los estilos resueltos (con respecto a todas las clases, después de que se aplica todo el CSS y se calculan los valores finales):

- **El método getComputedStyle**(elem, [pseudo]) retorna el objeto de estilo con ellos (solo lectura).