

 → El navegador: Documentos, Eventos e Interfaces

# Documento

Aquí aprenderemos a manipular una página web usando JavaScript.

1. Entorno del navegador, especificaciones
2. Árbol del Modelo de Objetos del Documento (DOM)
3. Recorriendo el DOM
4. Buscar: getElement\*, querySelector\*
5. Propiedades del nodo: tipo, etiqueta y contenido
6. Atributos y propiedades
7. Modificando el documento
8. Estilos y clases
9. Tamaño de elementos y desplazamiento
10. Tamaño de ventana y desplazamiento
11. Coordenadas



Lección anterior

Próxima lección



Compartir



Mapa del Tutorial

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 25 de junio de 2022

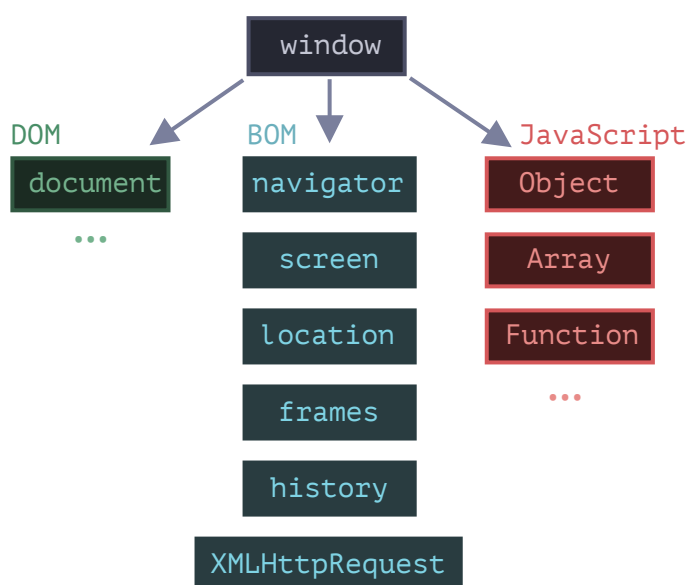
# Entorno del navegador, especificaciones

El lenguaje JavaScript fue creado inicialmente para los navegadores web. Desde entonces, ha evolucionado en un lenguaje con muchos usos y plataformas.

Una plataforma puede ser un navegador, un servidor web u otro *host* ("anfitrión"); incluso una máquina de café "inteligente", si puede ejecutar JavaScript. Cada uno de ellos proporciona una funcionalidad específica de la plataforma. La especificación de JavaScript llama a esto *entorno de host*.

Un entorno host proporciona sus propios objetos y funciones adicionales al núcleo del lenguaje. Los navegadores web proporcionan un medio para controlar las páginas web. Node.js proporciona características del lado del servidor, etc.

Aquí tienes una vista general de lo que tenemos cuando JavaScript se ejecuta en un navegador web:



Hay un objeto "raíz" llamado `window`. Tiene dos roles:

1. Primero, es un objeto global para el código JavaScript, como se describe en el capítulo [Objeto Global](#).
2. Segundo, representa la "ventana del navegador" y proporciona métodos para controlarla.

Por ejemplo, podemos usarlo como objeto global:

```
1 function sayHi() {  
2   alert("Hola");  
3 }  
4  
5
```



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

Y podemos usarlo como una ventana del navegador. Para ver la altura de la ventana:

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

Hay más métodos y propiedades específicos de `window`, los que cubriremos más adelante.

## DOM (Modelo de Objetos del Documento)

Document Object Model, o DOM, representa todo el contenido de la página como objetos que pueden ser modificados.

El objeto `document` es el punto de entrada a la página. Con él podemos cambiar o crear cualquier cosa en la página.

Por ejemplo:

```
1 // cambiar el color de fondo a rojo
2 document.body.style.background = "red";
3
4 // deshacer el cambio después de 1 segundo
5 setTimeout(() => document.body.style.background = "", 1000);
```

Aquí usamos `document.body.style`, pero hay muchos, muchos más. Las propiedades y métodos se describen en la especificación: [DOM Living Standard](#).

### **i** DOM no es solo para navegadores

La especificación DOM explica la estructura de un documento y proporciona objetos para manipularlo. Hay instrumentos que no son del navegador que también usan DOM.

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.

### **i** CSSOM para los estilos

También hay una especificación separada, [CSS Object Model \(CSSOM\)](#) para las reglas y hojas de estilo CSS, que explica cómo se representan como objetos y cómo leerlos y escribirlos.

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 (generalmente solo agregamos y eliminamos clases CSS, no modificamos sus reglas CSS), pero eso también es posible.

## BOM (Modelo de Objetos del Navegador)

El Modelo de Objetos del Navegador (Browser Object Model, BOM) son objetos adicionales proporcionados por el navegador (entorno host) para trabajar con todo excepto el documento.

Por ejemplo:

- El objeto `navigator` proporciona información sobre el navegador y el sistema operativo. Hay muchas propiedades, pero las dos más conocidas son: `navigator.userAgent` : acerca del navegador actual, y `navigator.platform` : acerca de la plataforma (ayuda a distinguir Windows/Linux/Mac, etc.).
- El objeto `location` nos permite leer la URL actual y puede redirigir el navegador a una nueva.

Aquí vemos cómo podemos usar el objeto `location` :

```
1 alert(location.href); // muestra la URL actual
2 if (confirm("Ir a wikipedia?")) {
3     location.href = "https://wikipedia.org"; // redirigir el navegador a otra URI
4 }
```

Las funciones `alert/confirm/prompt` también forman parte de BOM: no están directamente relacionadas con el documento, sino que representan métodos puros de comunicación del navegador con el usuario.

### Especificaciones

BOM es la parte general de la especificación de [HTML specification](https://html.spec.whatwg.org).

Sí, oíste bien. La especificación HTML en <https://html.spec.whatwg.org> no solo trata sobre el "lenguaje HTML" (etiquetas, atributos), sino que también cubre un montón de objetos, métodos y extensiones DOM específicas del navegador. Eso es "HTML en términos generales". Además, algunas partes tienen especificaciones adicionales listadas en <https://spec.whatwg.org>.

## Resumen

En términos de estándares, tenemos:

### La especificación del DOM

Describe la estructura del documento, las manipulaciones y los eventos; consulte <https://dom.spec.whatwg.org>.

### La especificación del CSSOM

Describe las hojas de estilo y las reglas de estilo, las manipulaciones con ellas y su vínculo a los documentos. Consulte <https://www.w3.org/TR/cssom-1/>.

### La especificación del HTML

Describe el lenguaje HTML (por ejemplo, etiquetas), y también el BOM (modelo de objeto del navegador) que describe varias funciones del navegador como `setTimeout` , `alert` , `location` , etc. Esta toma la especificación DOM y la extiende con muchas propiedades y métodos adicionales. Consulta <https://html.spec.whatwg.org>.

Adicionalmente, algunas clases son descritas separadamente en <https://spec.whatwg.org/>.

Ten en cuenta los enlaces anteriores, ya que hay tantas cosas que es imposible cubrir y recordar todo.

Cuando desees leer sobre una propiedad o un método, el manual de Mozilla en <https://developer.mozilla.org/es/search> es un buen recurso, pero leer las especificaciones correspondientes puede ser mejor: es más complejo y hay más para leer, pero hará que su conocimiento de los fundamentos sea sólido y completo.

Para encontrar algo, a menudo es conveniente usar una búsqueda como “WHATWG [término]” o “MDN [término]”. Por ejemplo <https://google.com?q=whatwg+localstorage>, <https://google.com?q=mdn+localstorage>.

Ahora nos concentraremos en aprender el DOM, porque `document` juega el papel central en la interfaz de usuario.

 Lección anterior

Próxima lección

Compartir  

 Mapa del Tutorial

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 24 de octubre de 2022

# Árbol del Modelo de Objetos del Documento (DOM)

La estructura de un documento HTML son las etiquetas.

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.

Todos estos objetos son accesibles empleando JavaScript, y podemos usarlos para modificar la página.

Por ejemplo, `document.body` es el objeto que representa la etiqueta `<body>`.

Ejecutar el siguiente código hará que el `<body>` sea de color rojo durante 3 segundos:

```
1 document.body.style.background = 'red'; // establece un color de fondo rojo
2
3 setTimeout(() => document.body.style.background = '', 3000); // volver atrás
```

En el caso anterior usamos `style.background` para cambiar el color de fondo del `document.body`, sin embargo existen muchas otras propiedades, tales como:

- `innerHTML` – contenido HTML del nodo.
- `offsetWidth` – ancho del nodo (en píxeles).
- ..., etc.

Más adelante, aprenderemos otras formas de manipular el DOM, pero primero necesitamos conocer su estructura.

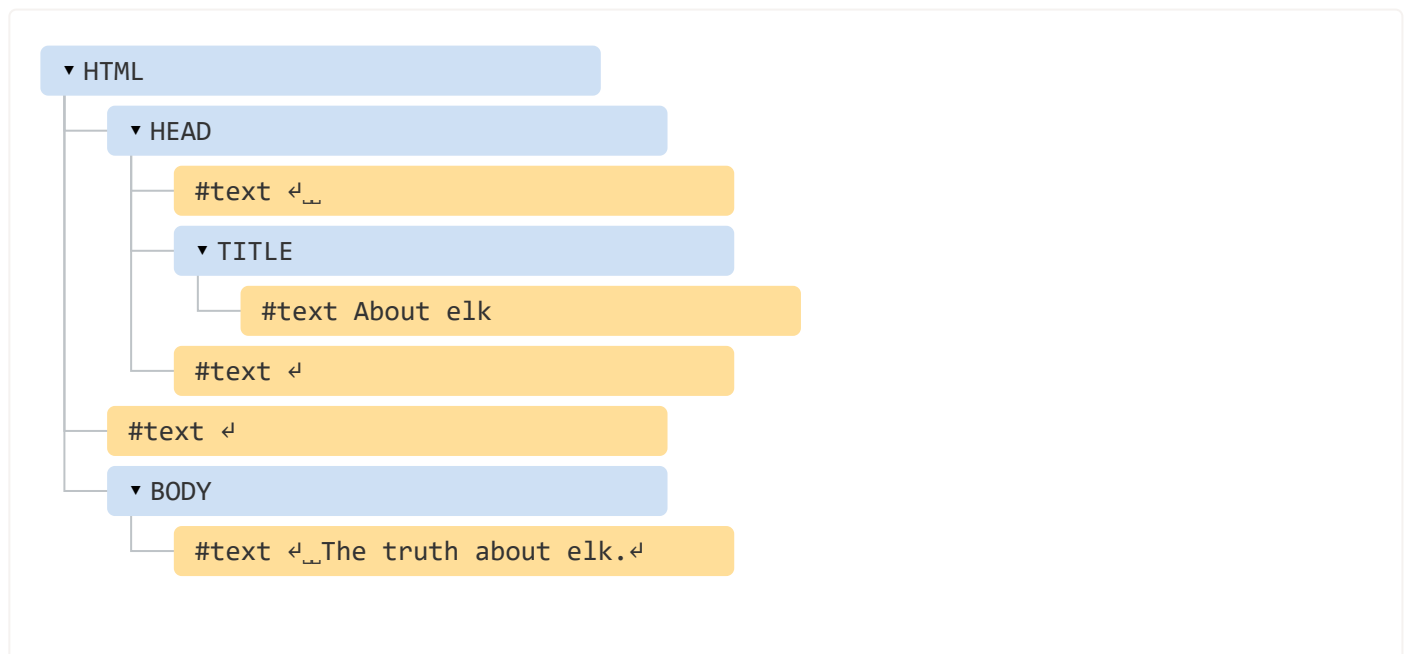
## Un ejemplo del DOM

Comencemos con el siguiente documento simple:

```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <title>About elk</title>
5 </head>
6 <body>
7   The truth about elk.
8
```

```
9 </body>
  </html>
```

El DOM representa HTML como una estructura de árbol de etiquetas. A continuación podemos ver como se muestra:



En la imagen de arriba, puedes hacer clic sobre los nodos del elemento y como resultado se expanden/colapsan sus nodos hijos.

Cada nodo del árbol es un objeto.

Las etiquetas son *nodos de elementos* (o solo elementos) y forman la estructura del árbol: `<html>` está ubicado en la raíz del documento, por lo tanto, `<head>` y `<body>` son sus hijos, etc.

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.

Por ejemplo, la etiqueta `<title>` tiene el texto "About elk".

Hay que tener en cuenta los caracteres especiales en nodos de texto:

- una línea nueva: `<br>` (en JavaScript se emplea `\n` para obtener este resultado)
- un espacio: `< >`

Los espacios y líneas nuevas son caracteres totalmente válidos, al igual que letras y dígitos. Ellos forman nodos de texto y se convierten en parte del DOM. Así, por ejemplo, en el caso de arriba la etiqueta `<head>` contiene algunos espacios antes de la etiqueta `<title>`, entonces ese texto se convierte en el nodo `#text`, que contiene una nueva línea y solo algunos espacios.

Hay solo dos excepciones de nivel superior:

1. Los espacios y líneas nuevas antes de la etiqueta `<head>` son ignorados por razones históricas.
2. 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 puede haber espacios después de esta.

En otros casos todo es sencillo – si hay espacios (como cualquier carácter) en el documento, se convierten en nodos de texto en el DOM, y si los eliminamos, entonces no habrá ninguno.

En el siguiente ejemplo, no hay nodos de texto con espacios en blanco:

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



### **i 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**

Las herramientas del navegador (las veremos más adelante) que trabajan con DOM usualmente no muestran espacios al inicio/final del texto y nodos de texto vacíos (saltos de línea) entre etiquetas.

De esta manera, las herramientas para desarrolladores ahorran espacio en la pantalla.

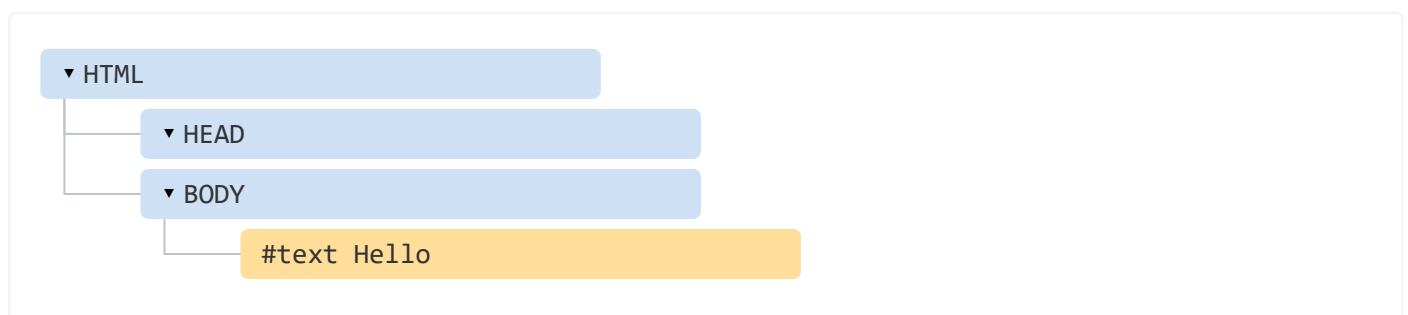
En otras representaciones del DOM, las omitiremos cuando sean irrelevantes. Tales espacios generalmente no afectan la forma en la cual el documento es mostrado.

## 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>` .

Como ejemplo de esto, si el archivo HTML es la palabra **"Hello"** , el navegador lo envolverá con las etiquetas `<html>` y `<body>` , y añadirá la etiqueta `<head>` la cual es requerida, basado en esto, el DOM resultante será:



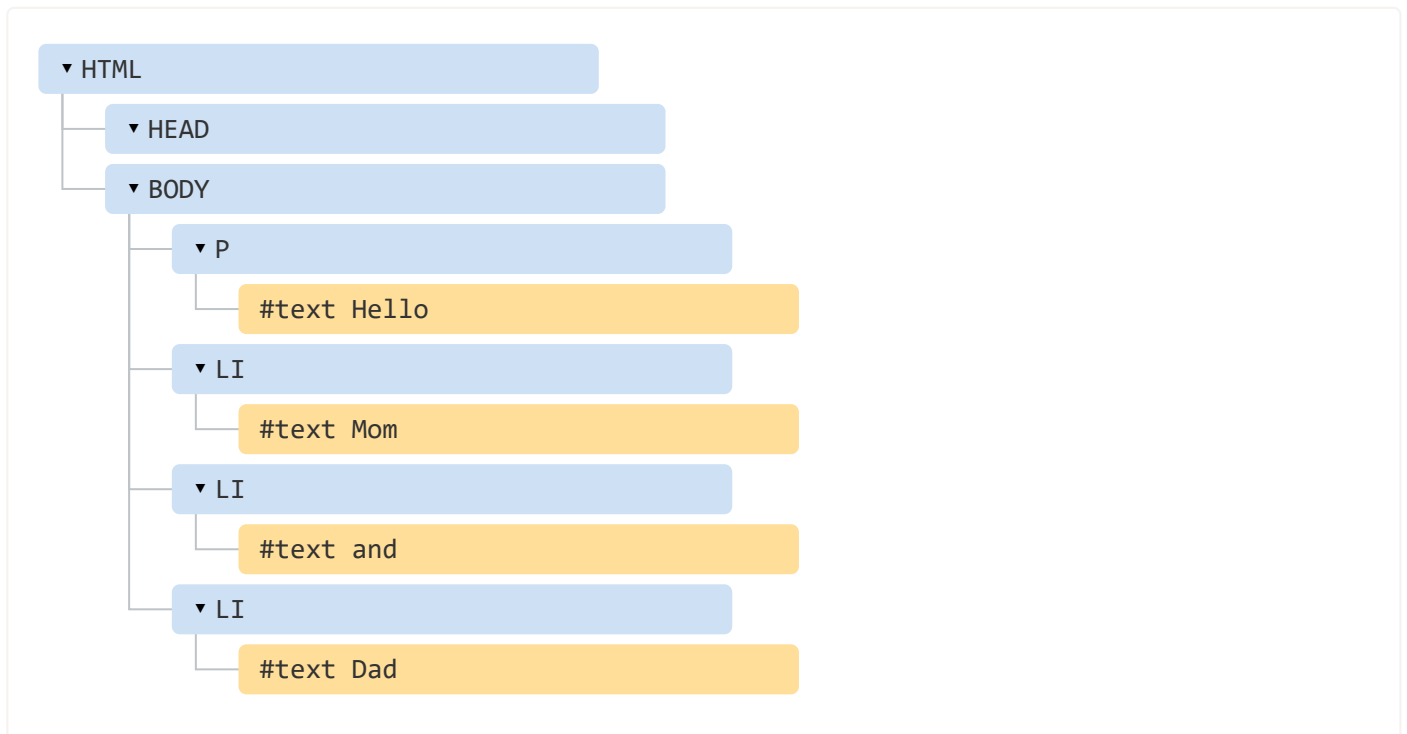


Al generar el DOM, los navegadores procesan automáticamente los errores en el documento, cierran etiquetas, etc.

Un documento sin etiquetas de cierre:

```
1 <p>Hello
2 <li>Mom
3 <li>and
4 <li>Dad
```

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



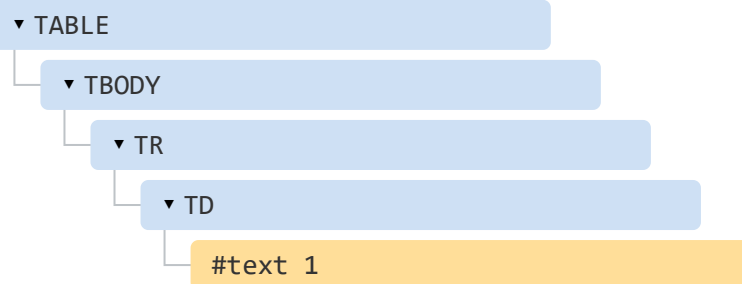
## ⚠ Las tablas siempre tienen la etiqueta `<tbody>`

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.

Para el HTML:

```
1 <table id="table"><tr><td>1</td></tr></table>
```

La estructura del DOM será:



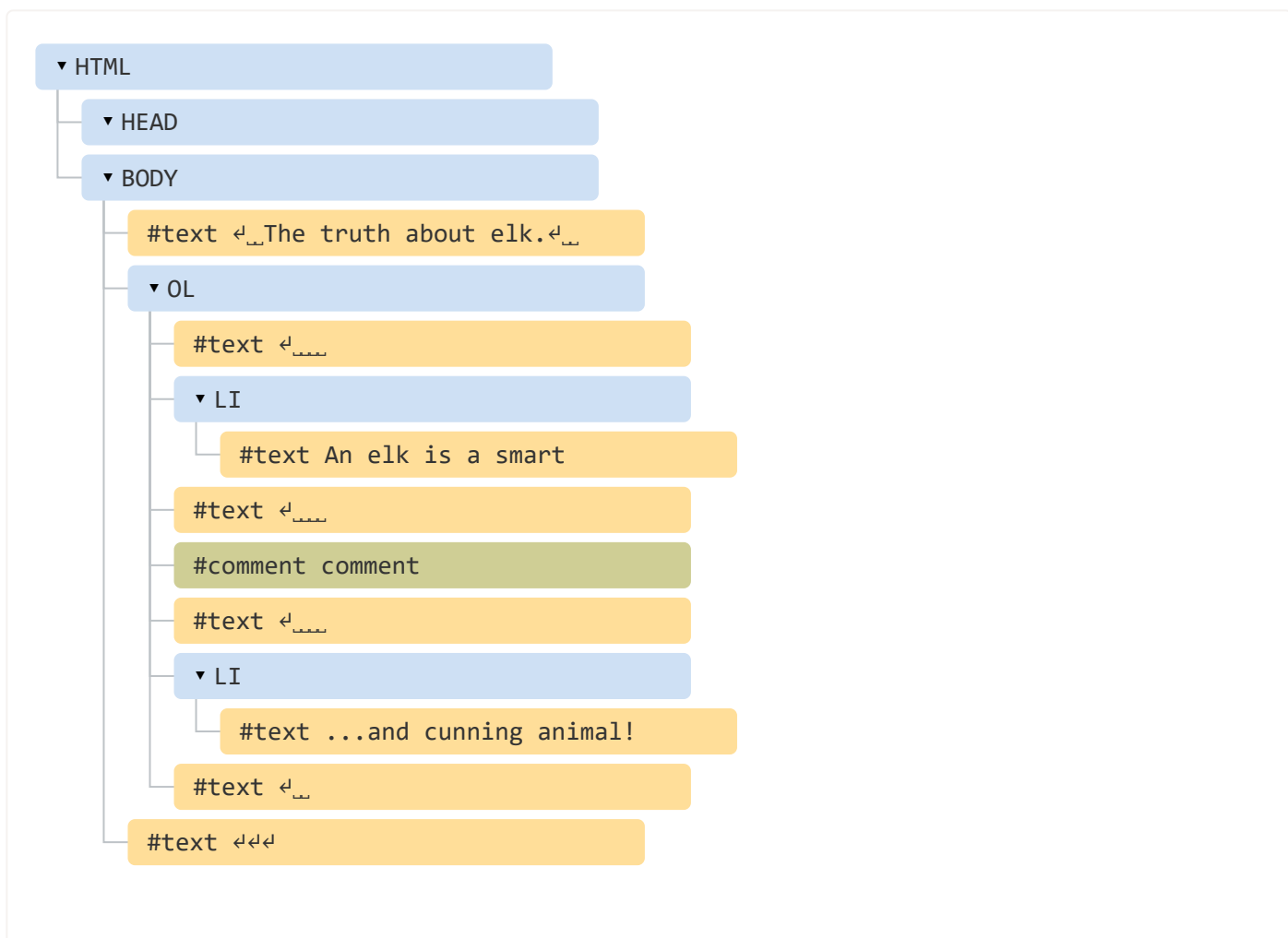
¿Lo ves? La etiqueta `<tbody>` apareció de la nada. Debemos tener esto en cuenta al trabajar con tablas para evitar sorpresas.

## Otros tipos de nodos

Existen otros tipos de nodos además de elementos y nodos de texto.

Por ejemplo, los comentarios:

```
1 <!DOCTYPE HTML>
2 <html>
3 <body>
4   The truth about elk.
5   <ol>
6     <li>An elk is a smart</li>
7     <!-- comentario -->
8     <li>...y el astuto animal!</li>
9   </ol>
10 </body>
11 </html>
```



Aquí podemos ver un nuevo tipo de nodo de árbol – *nodo de comentario*, etiquetado como `#comment`, entre dos nodos de texto.

Podemos pensar – ¿Por qué se agrega un comentario al DOM? Esto no afecta la representación de ninguna manera. Pero hay una regla – si algo está en el código HTML, entonces también debe estar en el árbol DOM.

**Todo en HTML, incluso los comentarios, se convierte en parte del DOM.**

Hasta la declaración `<!DOCTYPE...>` al principio del HTML es un nodo del DOM. Su ubicación en el DOM es justo antes de la etiqueta `<html>`. No vamos a tocar ese nodo, por esa razón ni siquiera lo dibujamos en diagramas, pero esta ahí.

El objeto `document` que representa todo el documento es también, formalmente, un nodo DOM.

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

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

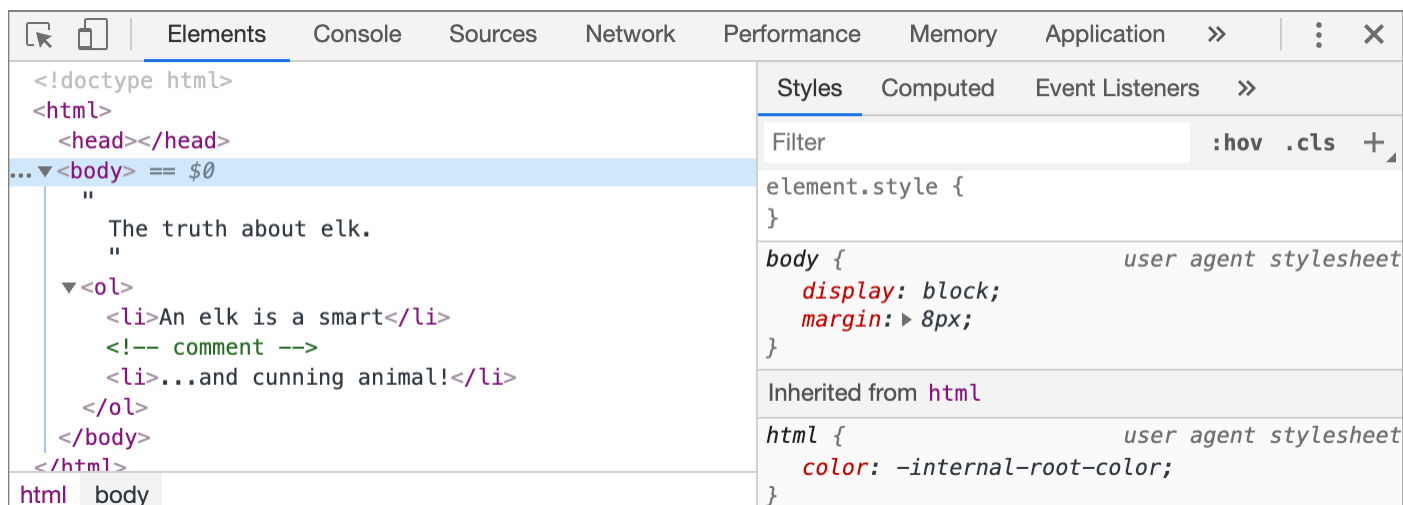
## Véalo usted mismo

Para ver la estructura del DOM en tiempo real, intente [Live DOM Viewer](#). Simplemente escriba el documento, y se mostrará como un DOM al instante.

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


Para hacerlo, abra la página web [elk.html](#), active las herramientas para desarrolladores del navegador y cambie la pestaña a elementos.

Debe verse así:

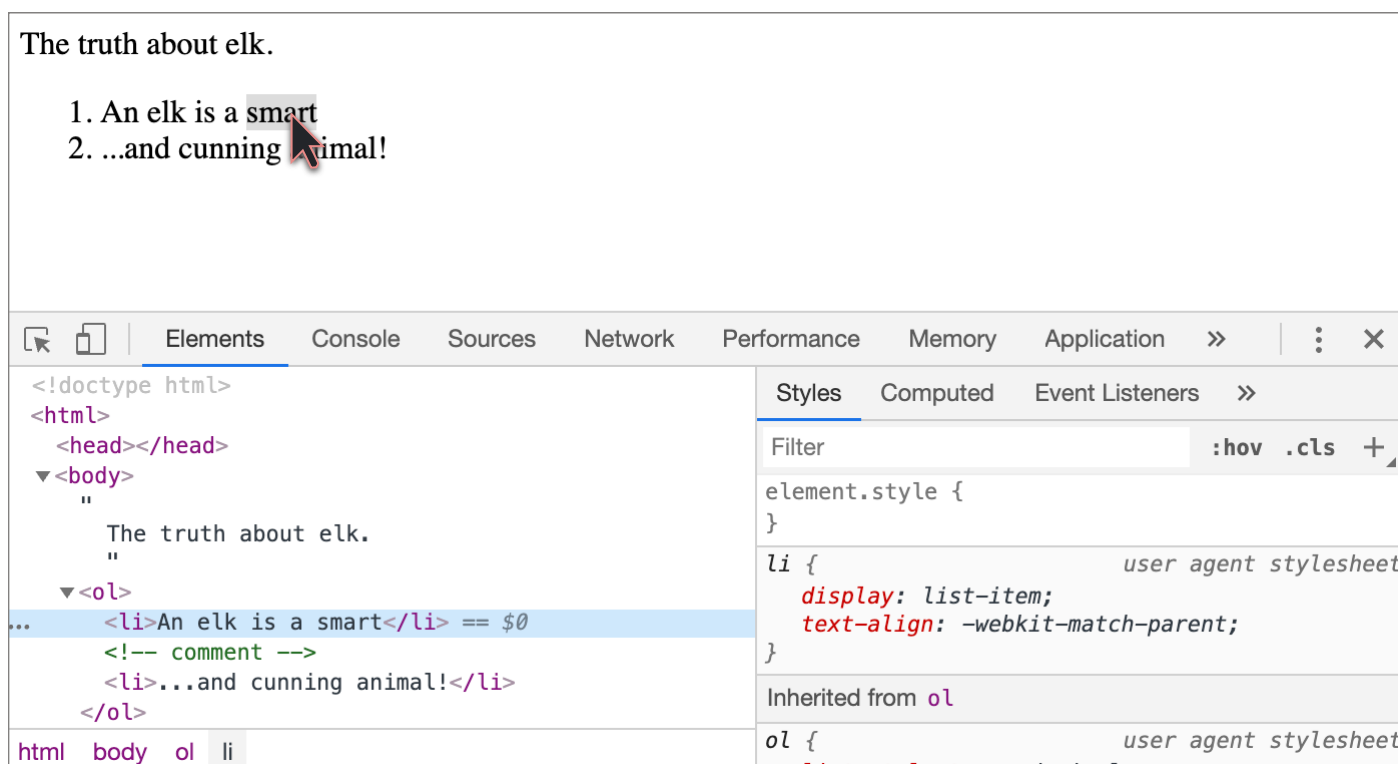


Puedes ver el DOM, hacer clic sobre los elementos, ver sus detalles, etc.

Tenga en cuenta que 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.

Hacer clic en el botón  ubicado en la esquina superior izquierda nos permite elegir un nodo desde la página web utilizando un "mouse" (u otros dispositivos de puntero) e "inspeccionar" (desplazarse hasta él en la pestaña elementos). Esto funciona muy bien cuando tenemos una página HTML enorme (y el DOM correspondiente es enorme) y nos gustaría ver la posición de un elemento en particular.

Otra forma de realizarlo sería hacer clic derecho en la página web y en el menú contextual elegir la opción "Inspeccionar Elemento".



En la parte derecha de las herramientas encontramos las siguientes sub-pestañas:

- **Styles** – podemos ver CSS aplicado al elemento actual regla por regla, incluidas las reglas integradas (gris). Casi todo puede ser editado en el lugar, incluyendo las dimensiones/márgenes/relleno de la siguiente caja.
- **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 (los cubriremos en la siguiente parte del tutorial).
- ...,etc.

La manera de estudiarlos es haciendo clic en ellos. Casi todos los valores son editables en el lugar.

## Interacción con la consola

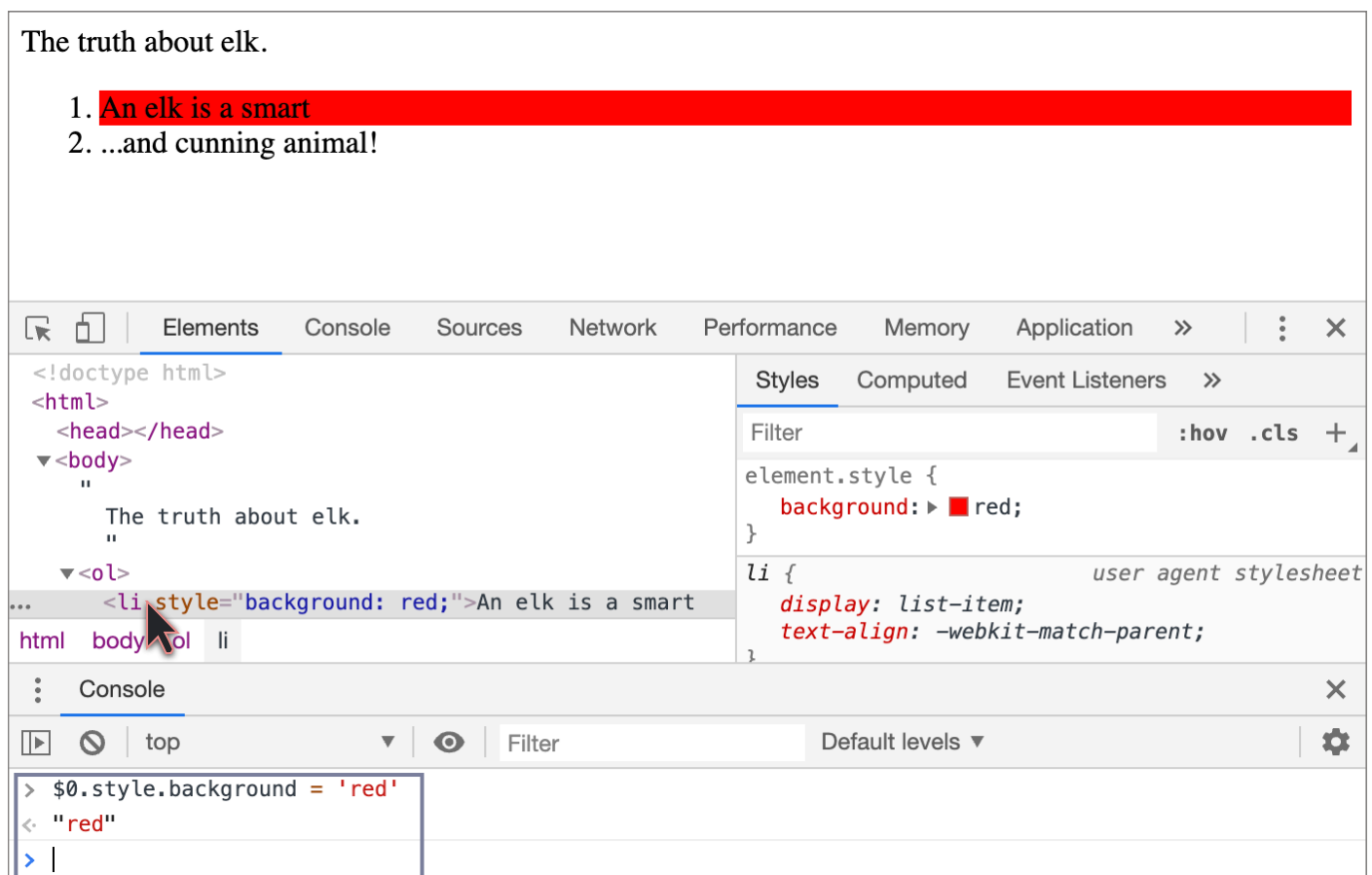
A medida que trabajamos el DOM, también podemos querer aplicarle JavaScript. Al igual que: obtener un nodo y ejecutar algún código para modificarlo, para ver el resultado. Aquí hay algunos consejos para desplazarse entre la pestaña elementos y la consola.

Para empezar:

1. Seleccione el primer elemento `<li>` en la pestaña elementos.
2. Presiona la tecla `Esc` – esto abrirá la consola justo debajo de la pestaña de elementos.

Ahora el último elemento seleccionado esta disponible como `$0`, el seleccionado previamente es `$1`, etc.

Podemos ejecutar comandos en ellos. Por ejemplo, `$0.style.background = 'red'` hace que el elemento de la lista seleccionado sea rojo, algo así:



The truth about elk.

1. An elk is a smart
2. ...and cunning animal!

Elements Console Sources Network Performance Memory Application >> ⋮ ✕

```
<!doctype html>
<html>
  <head></head>
  <body>
    "
      The truth about elk.
    "
    <ol>
      <li style="background: red;">An elk is a smart
    </ol>
  </body>
</html>
```

Styles Computed Event Listeners >>

Filter :hov .cls +

```
element.style {
  background: red;
}

li {
  display: list-item;
  text-align: -webkit-match-parent;
}
```

html body li

Console ✕

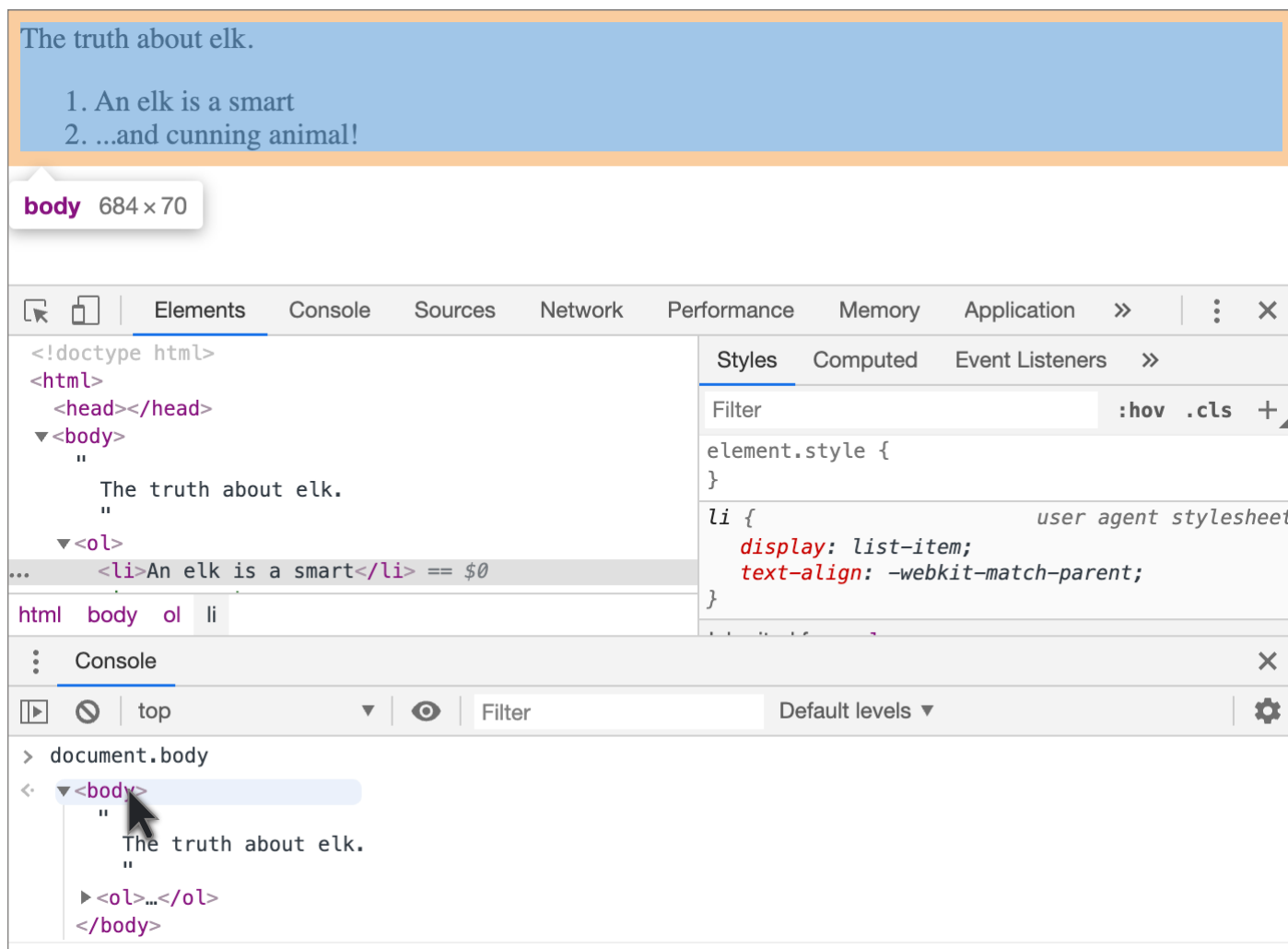
top Filter Default levels ⚙

```
> $0.style.background = 'red'
< "red"
> |
```

Así es como en la consola, se obtiene un nodo de los elementos.

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.

O simplemente podemos generar el nodo del DOM en la consola y explorar en el lugar, así como `document.body` a continuación:



Desde luego, eso es para propósitos de depuración del curso. A partir del siguiente capítulo accederemos y modificaremos el DOM usando JavaScript.

Las herramientas para desarrolladores del navegador son de mucha ayuda en el desarrollo: podemos explorar el DOM, probar cosas y ver que sale mal.

## Resumen

Un documento HTML/XML está representado dentro del navegador como un árbol de nodos (DOM).

- Las etiquetas se convierten en nodos de elemento y forman la estructura.
- El texto se convierte en nodos de texto.
- ...etc, todos los elementos de HTML tienen su lugar en el DOM, incluso los comentarios.

Podemos utilizar las herramientas para desarrolladores para inspeccionar el DOM y modificarlo manualmente.

Aquí cubrimos los conceptos básicos, las acciones más importantes y más utilizadas, para comenzar. Hay una extensa documentación acerca de las herramientas para desarrolladores de Chrome en <https://developers.google.com/web/tools/chrome-devtools>. La mejor forma de aprender a usar las herramientas

es hacer clic en ellas, leer los menús: la mayoría de las opciones son obvias. Más adelante, cuando tenga conocimiento general sobre ellas, lea los documentos y elija el resto.

Los nodos del DOM tienen propiedades y métodos que nos permiten desplazarnos entre ellos, modificarlos, moverlos por la página, y más. Empezaremos a realizar todo esto en los siguientes capítulos.

 Lección anterior

Próxima lección

Compartir  

 [Mapa del Tutorial](#)

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

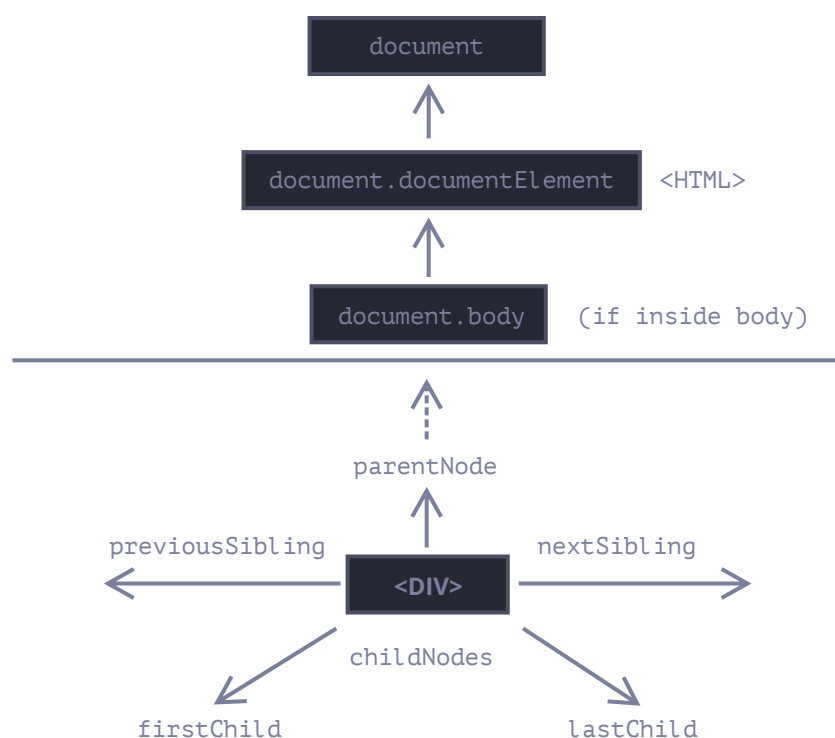
📅 24 de octubre de 2022

# Recorriendo el DOM

El DOM nos permite hacer cualquier cosa con sus elementos y contenidos, pero lo primero que tenemos que hacer es llegar al objeto correspondiente del DOM.

Todas las operaciones en el DOM comienzan con el objeto `document`. Este es el principal "punto de entrada" al DOM. Desde ahí podremos acceder a cualquier nodo.

Esta imagen representa los enlaces que nos permiten viajar a través de los nodos del DOM:



Vamos a analizarlos con más detalle.

## En la parte superior: documentElement y body

Los tres nodos superiores están disponibles como propiedades de `document`:

`<html> = document.documentElement`

El nodo superior del documento es `document.documentElement`. Este es el nodo del DOM para la etiqueta `<html>`.

`<body> = document.body`

Otro nodo muy utilizado es el elemento `<body>` – `document.body`.



`<head> = document.head`

La etiqueta `<head>` está disponible como `document.head`.

### ⚠ Hay una trampa: `document.body` puede ser `null`

Un script no puede acceder a un elemento que no existe en el momento de su ejecución.

Por ejemplo, si un script está dentro de `<head>`, entonces `document.body` no está disponible, porque el navegador no lo ha leído aún.

Entonces, en el siguiente ejemplo `alert` muestra `null`:

```
1 <html>
2
3 <head>
4   <script>
5     alert( "From HEAD: " + document.body ); // null, no hay <body> aún
6   </script>
7 </head>
8
9 <body>
10
11   <script>
12     alert( "From BODY: " + document.body ); // HTMLBodyElement, ahora exist
13   </script>
14
15 </body>
16 </html>
```

### 📌 En el mundo del DOM `null` significa “no existe”

En el DOM, el valor `null` significa que “no existe” o “no hay tal nodo”.

## Hijos: `childNodes`, `firstChild`, `lastChild`

Existen dos términos que vamos a utilizar de ahora en adelante:

- **Nodos hijos (`childNodes`)** – elementos que son hijos directos, es decir sus descendientes inmediatos. Por ejemplo, `<head>` y `<body>` son hijos del elemento `<html>`.
- **Descendientes** – todos los elementos anidados de un elemento dado, incluyendo los hijos, sus hijos y así sucesivamente.

Por ejemplo, aquí `<body>` tiene de hijos `<div>` y `<ul>` (y unos pocos nodos de texto en blanco):

```
1 <html>
2 <body>
```

```

3   <div>Begin</div>
4
5   <ul>
6     <li>
7       <b>Information</b>
8     </li>
9   </ul>
10 </body>
11 </html>

```

...Y los descendientes de `<body>` no son solo los hijos `<div>`, `<ul>` sino también elementos anidados más profundamente, como `<li>` (un hijo de `<ul>`) o `<b>` (un hijo de `<li>`) – el subárbol entero.

**La colección `childNodes` enumera todos los nodos hijos, incluidos los nodos de texto.**

El ejemplo inferior muestra todos los hijos de `document.body` :

```

1  <html>
2  <body>
3    <div>Begin</div>
4
5    <ul>
6      <li>Information</li>
7    </ul>
8
9    <div>End</div>
10
11   <script>
12     for (let i = 0; i < document.body.childNodes.length; i++) {
13       alert( document.body.childNodes[i] ); // Texto, DIV, Texto, UL, ..., SCR:
14     }
15   </script>
16   ...más cosas...
17 </body>
18 </html>

```

Por favor observa un interesante detalle aquí. Si ejecutamos el ejemplo anterior, el último elemento que se muestra es `<script>`. De hecho, el documento tiene más cosas debajo, pero en el momento de ejecución del script el navegador todavía no lo ha leído, por lo que el script no lo ve.

**Las propiedades `firstChild` y `lastChild` dan acceso rápido al primer y al último hijo.**

Son solo atajos. Si existieran nodos hijos, la respuesta siguiente sería siempre verdadera:

```

1 elem.childNodes[0] === elem.firstChild
2 elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

```

También hay una función especial `elem.hasChildNodes()` para comprobar si hay algunos nodos hijos.

## Colecciones del DOM

Como podemos ver, `childNodes` parece un array. Pero realmente no es un array, sino más bien una *colección* – un objeto especial iterable, simil-array.

Hay dos importantes consecuencias de esto:

1. Podemos usar `for...of` para iterar sobre él:

```
1 for (let node of document.body.childNodes) {  
2   alert(node); // enseña todos los nodos de la colección  
3 }
```

Eso es porque es iterable (proporciona la propiedad `Symbol.iterator`, como se requiere).

2. Los métodos de Array no funcionan, porque no es un array:

```
1 alert(document.body.childNodes.filter); // undefined (¡No hay método filter!)
```

La primera consecuencia es agradable. La segunda es tolerable, porque podemos usar `Array.from` para crear un array “real” desde la colección si es que queremos usar métodos del array:

```
1 alert( Array.from(document.body.childNodes).filter ); // función
```

### ⚠ Las colecciones DOM son solo de lectura

Las colecciones DOM, incluso más-- *todas* las propiedades de navegación enumeradas en este capítulo son sólo de lectura.

No podemos reemplazar a un hijo por otro elemento asignándolo así `childNodes[i] = ...`.

Cambiar el DOM necesita otros métodos. Los veremos en el siguiente capítulo.

### ⚠ Las colecciones del DOM están vivas

Casi todas las colecciones del DOM, salvo algunas excepciones, están *vivas*. En otras palabras, reflejan el estado actual del DOM.

Si mantenemos una referencia a `elem.childNodes`, y añadimos o quitamos nodos del DOM, entonces estos nodos aparecen en la colección automáticamente.

## ⚠ No uses `for..in` para recorrer colecciones

Las colecciones son iterables usando `for..of`. Algunas veces las personas tratan de utilizar `for..in` para eso.

Por favor, no lo hagas. El bucle `for..in` itera sobre todas las propiedades enumerables. Y las colecciones tienen unas propiedades “extra” raramente usadas que normalmente no queremos obtener:

```
1 <body>
2 <script>
3   // enseña 0, 1, longitud, item, valores y más cosas.
4   for (let prop in document.body.childNodes) alert(prop);
5 </script>
6 </body>
```



## Hermanos y el padre

Los *hermanos* son nodos que son hijos del mismo padre.

Por ejemplo, aquí `<head>` y `<body>` son hermanos:

```
1 <html>
2   <head>...</head><body>...</body>
3 </html>
```

- `<body>` se dice que es el hermano “siguiente” o a la “derecha” de `<head>`,
- `<head>` se dice que es el hermano “anterior” o a la “izquierda” de `<body>`.

El hermano siguiente está en la propiedad `nextSibling` y el anterior – en `previousSibling`.

El padre está disponible en `parentNode`.

Por ejemplo:

```
1 // el padre de <body> es <html>
2 alert( document.body.parentNode === document.documentElement ); // verdadero
3
4 // después de <head> va <body>
5 alert( document.head.nextSibling ); // HTMLBodyElement
6
7 // antes de <body> va <head>
8 alert( document.body.previousSibling ); // HTMLHeadElement
```

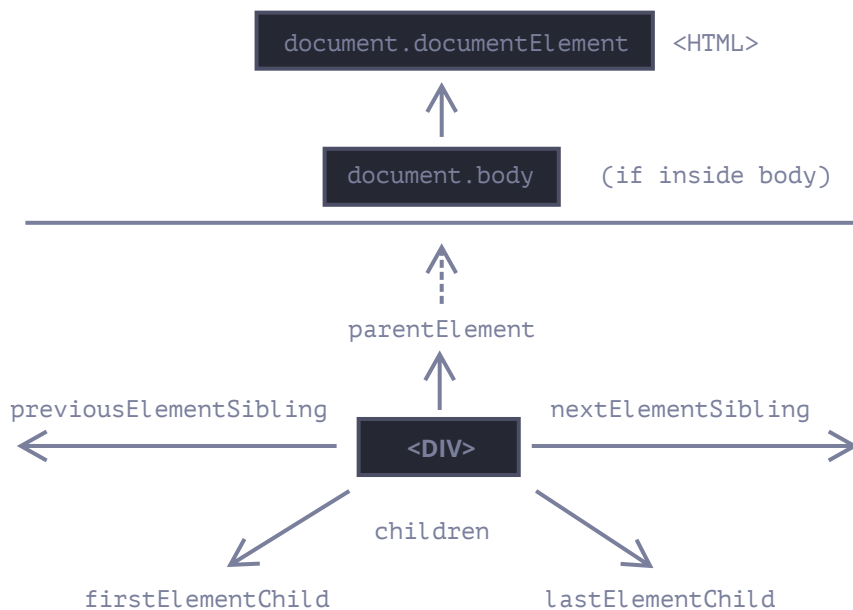


## Navegación solo por elementos

Las propiedades de navegación enumeradas abajo se refieren a *todos* los nodos. Por ejemplo, en `childNodes` podemos ver nodos de texto, nodos elementos; y si existen, incluso los nodos de comentarios.

Pero para muchas tareas no queremos los nodos de texto o comentarios. Queremos manipular el nodo que representa las etiquetas y formularios de la estructura de la página.

Así que vamos a ver más enlaces de navegación que solo tienen en cuenta los *elementos nodos*:



Los enlaces son similares a los de arriba, solo que tienen dentro la palabra **Element** :

- `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.

### **i** ¿Por qué `parentElement` ? ¿Puede el padre *no* ser un elemento?

La propiedad `parentElement` devuelve el "elemento" padre, mientras `parentNode` devuelve "cualquier nodo" padre. Estas propiedades son normalmente las mismas: ambas seleccionan el padre.

Con la excepción de `document.documentElement` :

```
1 alert( document.documentElement.parentNode ); // documento
2 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.

Este detalle puede ser útil cuando queramos navegar hacia arriba desde cualquier elemento `elem` al `<html>` , pero no hacia el `document` :

```
1 while(elem = elem.parentElement) { // sube hasta <html>
2   alert( elem );
3 }
```

Vamos a modificar uno de los ejemplos de arriba: reemplaza `childNodes` por `children` . Ahora enseña solo elementos:

```
1 <html>
2 <body>
3   <div>Begin</div>
4
5   <ul>
6     <li>Information</li>
7   </ul>
8
9   <div>End</div>
10
11  <script>
12    for (let elem of document.body.children) {
13      alert(elem); // DIV, UL, DIV, SCRIPT
14    }
15  </script>
16  ...
17 </body>
18 </html>
```

## Más enlaces: tablas

Hasta ahora hemos descrito las propiedades de navegación básicas.

Ciertos tipos de elementos del DOM pueden tener propiedades adicionales, específicas de su tipo, por conveniencia.

Las tablas son un gran ejemplo de ello, y representan un particular caso importante:

**El elemento `<table>`** soporta estas propiedades (añadidas a las que hemos dado anteriormente):

- `table.rows` – la colección de elementos `<tr>` de la tabla.
- `table.caption/tHead/tFoot` – referencias a los elementos `<caption>`, `<thead>`, `<tfoot>`.
- `table.tBodies` – la colección de elementos `<tbody>` (pueden ser muchos según el estándar, pero siempre habrá al menos uno, aunque no esté en el HTML el navegador lo pondrá en el DOM).

**`<thead>`, `<tfoot>`, `<tbody>`** estos elementos proporcionan las propiedades de las **filas**.

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

**`<tr>`:**

- `tr.cells` – la colección de celdas `<td>` y `<th>` dentro del `<tr>` dado.
- `tr.sectionRowIndex` – la posición (índice) del `<tr>` dado dentro del `<thead>/<tbody>/<tfoot>` adjunto.
- `tr.rowIndex` – el número de `<tr>` en la tabla en su conjunto (incluyendo todas las filas de una tabla).

**`<td>` and `<th>`:**

- `td.cellIndex` – el número de celdas dentro del adjunto `<tr>`.

Un ejemplo de uso:

```
1 <table id="table">
2   <tr>
3     <td>one</td><td>two</td>
4   </tr>
5   <tr>
6     <td>three</td><td>four</td>
7   </tr>
8 </table>
9
10 <script>
11   // seleccionar td con "dos" (primera fila, segunda columna)
12   let td = table.rows[0].cells[1];
13   td.style.backgroundColor = "red"; // destacarlo
14 </script>
```



La especificación: [tabular data](#).

También hay propiedades de navegación adicionales para los formularios HTML. Las veremos más adelante cuando empecemos a trabajar con los formularios.

## Resumen

Dado un nodo del DOM, podemos ir a sus inmediatos vecinos utilizando las propiedades de navegación.

Hay dos conjuntos principales de ellas:

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

Algunos tipos de elementos del DOM, por ejemplo las tablas, proveen propiedades adicionales y colecciones para acceder a su contenido.

## ✓ Tareas

---

### DOM children

importancia: 5

Mira esta página:

```
1 <html>
2 <body>
3   <div>Users:</div>
4   <ul>
5     <li>John</li>
6     <li>Pete</li>
7   </ul>
8 </body>
9 </html>
```

Para cada una de las siguientes preguntas, da al menos una forma de cómo acceder a ellos:

- ¿El nodo `<div>` del DOM?
- ¿El nodo `<ul>` del DOM?
- El segundo `<li>` (con Pete)?

solución

---

### La pregunta de los hermanos

importancia: 5

Si `elem` – es un elemento nodo arbitrario del DOM...

- ¿Es cierto que `elem.lastChild.nextSibling` siempre es `null`?
- ¿Es cierto que `elem.children[0].previousSibling` siempre es `null`?

solución



## Seleccionar todas las celdas diagonales

importancia: 5

Escribe el código para pintar todas las celdas diagonales de rojo.

Necesitarás obtener todas las `<td>` de la `<table>` y pintarlas usando el código:

```
1 // td debe ser la referencia a la celda de la tabla
2 td.style.backgroundColor = 'red';
```

El resultado debe ser:

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

[Abrir un entorno controlado para la tarea.](#)

[solución](#)



Lección anterior

Próxima lección



Compartir



[Mapa del Tutorial](#)

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))



🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 24 de octubre de 2022

# Buscar: getElement\*, querySelector\*

Las propiedades de navegación del DOM son ideales cuando los elementos están cerca unos de otros. Pero, ¿y si no lo están? ¿Cómo obtener un elemento arbitrario de la página?

Para estos casos existen métodos de búsqueda adicionales.

## document.getElementById o sólo id

Si un elemento tiene el atributo `id`, podemos obtener el elemento usando el método `document.getElementById(id)`, sin importar dónde se encuentre.

Por ejemplo:

```
1 <div id="elem">
2   <div id="elem-content">Elemento</div>
3 </div>
4
5 <script>
6   // obtener el elemento
7   let elem = document.getElementById('elem');
8
9   // hacer que su fondo sea rojo
10  elem.style.background = 'red';
11 </script>
```



Existe además una variable global nombrada por el `id` que hace referencia al elemento:

```
1 <div id="elem">
2   <div id="elem-content">Elemento</div>
3 </div>
4
5 <script>
6   // elem es una referencia al elemento del DOM con id="elem"
7   elem.style.background = 'red';
8
9   // id="elem-content" tiene un guion en su interior, por lo que no puede ser u
10  // ...pero podemos acceder a él usando corchetes: window['elem-content']
11 </script>
```



...Esto es a menos que declaremos una variable de JavaScript con el mismo nombre, entonces ésta tiene prioridad:

```
1 <div id="elem"></div>
2
3 <script>
4   let elem = 5; // ahora elem es 5, no una referencia a <div id="elem">
5
6   alert(elem); // 5
7 </script>
```

### Por favor, no utilice variables globales nombradas por id para acceder a los elementos

Este comportamiento se encuentra descrito [en la especificación](#), pero está soportado principalmente para compatibilidad.

El navegador intenta ayudarnos mezclando espacios de nombres (*namespaces*) de JS y DOM. Esto está bien para los scripts simples, incrustados en HTML, pero generalmente no es una buena 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.

Aquí en el tutorial usamos `id` para referirnos directamente a un elemento por brevedad, cuando es obvio de dónde viene el elemento.

En la vida real `document.getElementById` es el método preferente.

### El `id` debe ser único

El `id` debe ser único. Sólo puede haber en todo el documento un elemento con un `id` determinado.

Si hay múltiples elementos con el mismo `id`, entonces el comportamiento de los métodos que lo usan es impredecible, por ejemplo `document.getElementById` puede devolver cualquiera de esos elementos al azar. Así que, por favor, sigan la regla y mantengan el `id` único.

### Sólo `document.getElementById`, no `anyElem.getElementById`

El método `getElementById` sólo puede ser llamado en el objeto `document`. Busca el `id` dado en todo el documento.

## querySelectorAll

Sin duda el método más versátil, `elem.querySelectorAll(css)` 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:

```
1 <ul>
```

```

2   <li>La</li>
3   <li>prueba</li>
4 </ul>
5 <ul>
6   <li>ha</li>
7   <li>pasado</li>
8 </ul>
9 <script>
10  let elements = document.querySelectorAll('ul > li:last-child');
11
12  for (let elem of elements) {
13    alert(elem.innerHTML); // "prueba", "pasado"
14  }
15 </script>

```

Este método es muy poderoso, porque se puede utilizar cualquier selector de CSS.

### **i** También se pueden usar pseudoclases

Las pseudoclases como `:hover` (cuando el cursor sobrevuela el elemento) y `:active` (cuando hace clic con el botón principal) también son soportadas. Por ejemplo, `document.querySelectorAll(':hover')` devolverá una colección de elementos sobre los que el puntero hace hover en ese momento (en orden de anidación: desde el más exterior `<html>` hasta el más anidado).

## querySelector

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

En otras palabras, el resultado es el mismo que `elem.querySelectorAll(css)[0]`, pero este último busca *todos* los elementos y elige uno, mientras que `elem.querySelector` sólo busca uno. Así que es más rápido y también más corto de escribir.

## matches

Los métodos anteriores consistían en buscar en el DOM.

El `elem.matches(css)` no busca nada, sólo comprueba si el `elem` coincide con el selector CSS dado. Devuelve `true` o `false`.

Este método es útil cuando estamos iterando sobre elementos (como en un array) y tratando de filtrar los que nos interesan.

Por ejemplo:



```

1 <a href="http://example.com/file.zip">...</a>
2 <a href="http://ya.ru">...</a>
3
4 <script>
5   // puede ser cualquier colección en lugar de document.body.children
6   for (let elem of document.body.children) {

```

```

7     if (elem.matches('a[href$="zip"]')) {
8         alert("La referencia del archivo: " + elem.href );
9     }
10 }
11 </script>

```

## closest

Los *ancestros* de un elemento son: el padre, el padre del padre, su padre y así sucesivamente. Todos los ancestros juntos forman la cadena de padres desde el elemento hasta la cima.

El método `elem.closest(css)` busca el ancestro más cercano que coincide con el selector CSS. El propio `elem` también se incluye en la búsqueda.

En otras palabras, el método `closest` sube del elemento y comprueba cada uno de los padres. Si coincide con el selector, entonces la búsqueda se detiene y devuelve dicho ancestro.

Por ejemplo:

```

1  <h1>Contenido</h1>
2
3  <div class="contents">
4      <ul class="book">
5          <li class="chapter">Capítulo 1</li>
6          <li class="chapter">Capítulo 2</li>
7      </ul>
8  </div>
9
10 <script>
11     let chapter = document.querySelector('.chapter'); // LI
12
13     alert(chapter.closest('.book')); // UL
14     alert(chapter.closest('.contents')); // DIV
15
16     alert(chapter.closest('h1')); // null (porque h1 no es un ancestro)
17 </script>

```



## getElementsBy\*

También hay otros métodos que permiten buscar nodos por una etiqueta, una clase, etc.

Hoy en día, son en su mayoría historia, ya que `querySelector` es más poderoso y corto de escribir.

Aquí los cubrimos principalmente por completar el temario, aunque todavía se pueden encontrar en scripts antiguos.

- `elem.getElementsByTagName(tag)` busca elementos con la etiqueta dada y devuelve una colección con ellos. 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.

Por ejemplo:

```
1 // obtener todos los divs del documento
2 let divs = document.getElementsByTagName('div');
```

Para encontrar todas las etiquetas `input` dentro de una tabla:

```
1 <table id="table">
2   <tr>
3     <td>Su edad:</td>
4
5     <td>
6       <label>
7         <input type="radio" name="age" value="young" checked> menos de 18
8       </label>
9       <label>
10        <input type="radio" name="age" value="mature"> de 18 a 50
11      </label>
12      <label>
13        <input type="radio" name="age" value="senior"> más de 60
14      </label>
15    </td>
16  </tr>
17 </table>
18
19 <script>
20   let inputs = table.getElementsByTagName('input');
21
22   for (let input of inputs) {
23     alert( input.value + ': ' + input.checked );
24   }
25 </script>
```

### ⚠ ¡No olvides la letra "s" !

Los desarrolladores novatos a veces olvidan la letra `"s"` . Esto es, intentan llamar a `getElementByTagName` en vez de a `getElementsByTagName` .

La letra `"s"` no se encuentra en `getElementById` porque devuelve sólo un elemento. But `getElementsByTagName` devuelve una colección de elementos, de ahí que tenga la `"s"` .

## ⚠ ¡Devuelve una colección, no un elemento!

Otro error muy extendido entre los desarrolladores novatos es escribir:

```
1 // no funciona
2 document.getElementsByTagName('input').value = 5;
```

Esto no funcionará, porque toma una *colección* de inputs y le asigna el valor a ella en lugar de a los elementos dentro de ella.

En dicho caso, deberíamos iterar sobre la colección o conseguir un elemento por su índice y luego asignarlo así:

```
1 // debería funcionar (si hay un input)
2 document.getElementsByTagName('input')[0].value = 5;
```

Buscando elementos `.article`:

```
1 <form name="my-form">
2   <div class="article">Artículo</div>
3   <div class="long article">Artículo largo</div>
4 </form>
5
6 <script>
7   // encontrar por atributo de nombre
8   let form = document.getElementsByName('my-form')[0];
9
10  // encontrar por clase dentro del formulario
11  let articles = form.getElementsByClassName('article');
12  alert(articles.length); // 2, encontró dos elementos con la clase "article"
13 </script>
```

## Colecciones vivas

Todos los métodos `"getElementsBy*"` devuelven una colección *viva* (*live collection*). Tales colecciones siempre reflejan el estado actual del documento y se "auto-actualizan" cuando cambia.

En el siguiente ejemplo, hay dos scripts.

1. El primero crea una referencia a la colección de `<div>`. Por ahora, su longitud es `1`.
2. El segundo script se ejecuta después de que el navegador se encuentre con otro `<div>`, por lo que su longitud es de `2`.

```
1 <div>Primer div</div>
2
```



```

3  <script>
4    let divs = document.getElementsByTagName('div');
5    alert(divs.length); // 1
6  </script>
7
8  <div>Segundo div</div>
9
10 <script>
11   alert(divs.length); // 2
12 </script>

```

Por el contrario, `querySelectorAll` devuelve una colección *estática*. Es como un array de elementos fijos.

Si lo utilizamos en lugar de `getElementsByTagName`, entonces ambos scripts dan como resultado **1**:

```

1  <div>Primer div</div>
2
3  <script>
4    let divs = document.querySelectorAll('div');
5    alert(divs.length); // 1
6  </script>
7
8  <div>Segundo div</div>
9
10 <script>
11   alert(divs.length); // 1
12 </script>

```

Ahora podemos ver fácilmente la diferencia. La colección estática no aumentó después de la aparición de un nuevo `div` en el documento.

## Resumen

Hay 6 métodos principales para buscar nodos en el DOM:

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

Los más utilizados son `querySelector` y `querySelectorAll`, pero `getElementBy*` puede ser de ayuda esporádicamente o encontrarse en scripts antiguos.

Aparte de eso:

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

Y mencionemos un método más para comprobar la relación hijo-padre, ya que a veces es útil:

- `elemA.contains(elemB)` devuelve true si `elemB` está dentro de `elemA` (un descendiente de `elemA`) o cuando `elemA==elemB`.

## ✓ Tareas

### Buscar elementos

importancia: 4

Aquí está el documento con la tabla y el formulario.

¿Cómo encontrar?...

1. La tabla con `id="age-table"`.
2. Todos los elementos `label` dentro de la tabla (debería haber 3).
3. El primer `td` en la tabla (con la palabra "Age").
4. El `form` con `name="search"`.
5. El primer `input` en ese formulario.
6. El último `input` en ese formulario.

Abra la página [table.html](#) en una ventana separada y haga uso de las herramientas del navegador.

**solución**



Lección anterior

Próxima lección



Compartir



Mapa del Tutorial

## 💬 Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.

- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 25 de junio de 2022

# Propiedades del nodo: tipo, etiqueta y contenido

Echemos un mirada más en profundidad a los nodos DOM.

En este capítulo veremos más sobre cuáles son y aprenderemos sus propiedades más utilizadas.

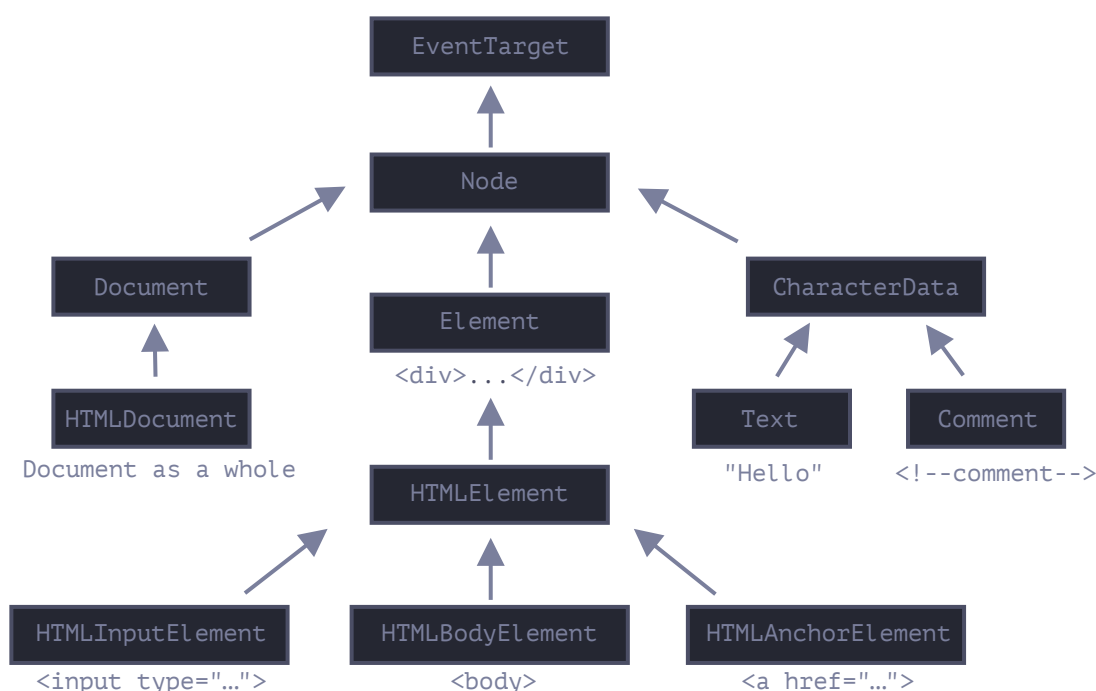
## Clases de nodo DOM

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, y el correspondiente a `<input>` tiene propiedades relacionadas con la entrada y así sucesivamente. Los nodos de texto no son lo mismo que los nodos de elementos. Pero también hay propiedades y métodos comunes entre todos ellos, porque todas las clases de nodos DOM forman una única jerarquía.

Cada nodo DOM pertenece a la clase nativa correspondiente.

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

Aquí está la imagen, con las explicaciones a continuación:



Las clases son:

- **EventTarget** – es la clase raíz “abstracta”.

Los objetos de esta clase nunca se crean. Sirve como base, es por la que todos los nodos DOM soportan los llamados “eventos” que estudiaremos más adelante.

- **Node** – también es una clase “abstracta”, sirve como base para los nodos DOM.

Proporciona la funcionalidad del árbol principal: **parentNode**, **nextSibling**, **childNodes** y demás (son getters). Los objetos de la clase **Node** nunca se crean. Pero hay clases de nodos concretas que heredan de ella (y también heredan la funcionalidad de **Node**).

- **Document**, por razones históricas, heredado a menudo por **HTMLDocument** (aunque la última especificación no lo exige) – es el documento como un todo.

El objeto global **document** pertenece exactamente a esta clase. Sirve como punto de entrada al DOM.

- **CharacterData** – una clase “abstract” heredada por:

- **Text** – la clase correspondiente a texto dentro de los elementos, por ejemplo **Hello** en `<p>Hello</p>`.
- **Comment** – la clase para los “comentarios”. No se muestran, pero cada comentario se vuelve un miembro del DOM.

- **Element** – es una clase base para elementos DOM.

Proporciona navegación a nivel de elemento como **nextElementSibling**, **children** y métodos de búsqueda como **getElementsByTagName**, **querySelector**.

Un navegador admite no solo HTML, sino también XML y SVG. La clase **Element** sirve como base para clases más específicas: **SVGElement**, **XMLElement** (no las necesitamos aquí) y **HTMLElement**.

- Finalmente, **HTMLElement** – es la clase básica para todos los elementos HTML. Trabajaremos con ella la mayor parte del tiempo.

Es heredado por elementos HTML concretos:

- **HTMLInputElement** – la clase para elementos `<input>`,
- **HTMLBodyElement** – la clase para los elementos `<body>`,
- **HTMLAnchorElement** – la clase para elementos `<a>`,
- ...y así sucesivamente.

Hay muchas otras etiquetas con sus propias clases que pueden tener propiedades y métodos específicos, mientras que algunos elementos, tales como `<span>`, `<section>`, `<article>`, no tienen ninguna propiedad específica entonces derivan de la clase **HTMLElement**.

Entonces, el conjunto completo de propiedades y métodos de un nodo dado viene como resultado de la cadena de herencia.

Por ejemplo, consideremos el objeto DOM para un elemento `<input>`. Pertenece a la clase **HTMLInputElement**.

Obtiene propiedades y métodos como una superposición de (enumerados en orden de herencia):

- **HTMLInputElement** – esta clase proporciona propiedades específicas de entrada,
- **HTMLElement** – proporciona métodos de elementos HTML comunes (y getters/setters),
- **Element** – proporciona métodos de elementos genéricos,
- **Node** – proporciona propiedades comunes del nodo DOM,
- **EventTarget** – da el apoyo para eventos (a cubrir),

- ...y finalmente hereda de `Object` , por lo que también están disponibles métodos de “objeto simple” como `hasOwnProperty` .

Para ver el nombre de la clase del nodo DOM, podemos recordar que un objeto generalmente tiene la propiedad `constructor` . Hace referencia al constructor de la clase, y `constructor.name` es su nombre:

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

...O podemos simplemente usar `toString` :

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

También podemos usar `instanceof` para verificar la herencia:

```
1 alert( document.body instanceof HTMLBodyElement ); // true
2 alert( document.body instanceof HTMLElement ); // true
3 alert( document.body instanceof Element ); // true
4 alert( document.body instanceof Node ); // true
5 alert( document.body instanceof EventTarget ); // true
```

Como podemos ver, los nodos DOM son objetos regulares de JavaScript. Usan clases basadas en prototipos para la herencia.

Eso también es fácil de ver al generar un elemento con `console.dir(elem)` en un navegador. Allí, en la consola, puede ver `HTMLElement.prototype` , `Element.prototype` y así sucesivamente.

### **i** `console.dir(elem)` versus `console.log(elem)`

La mayoría de los navegadores admiten dos comandos en sus herramientas de desarrollo: `console.log` y `console.dir` . Envían sus argumentos a la consola. Para los objetos JavaScript, estos comandos suelen hacer lo mismo.

Pero para los elementos DOM son diferentes:

- `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.

Inténtalo en `document.body` .

## IDL en la especificación

En la especificación, las clases DOM no se describen mediante JavaScript, sino con un [Lenguaje de descripción de interfaz](#) (IDL) especial, que suele ser fácil de entender.

En IDL, todas las propiedades están precedidas por sus tipos. Por ejemplo, `DOMString`, `boolean` y así sucesivamente.

Aquí hay un extracto, con comentarios:

```
1 // Definir HTMLInputElement
2 // Los dos puntos ":" significan que HTMLInputElement hereda de HTMLElement
3 interface HTMLInputElement: HTMLElement {
4     // aquí van las propiedades y métodos de los elementos <input>
5
6     // "DOMString" significa que el valor de una propiedad es un string
7     attribute DOMString accept;
8     attribute DOMString alt;
9     attribute DOMString autocomplete;
10    attribute DOMString value;
11
12    // Propiedad de valor booleano (true/false)
13    attribute boolean autofocus;
14    ...
15    // ahora el método: "void" significa que el método no devuelve ningún val
16    void select();
17    ...
18 }
```

## La propiedad “nodeType”

La propiedad `nodeType` proporciona una forma “anticuada” más de obtener el “tipo” de un nodo DOM.

Tiene un valor numérico:

- `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:

```
1 <body>
2   <script>
3     let elem = document.body;
4
5     // vamos a examinar: ¿qué tipo de nodo es elem?
6     alert(elem.nodeType); // 1 => elemento
```



```

7
8 // Y el primer hijo es...
9 alert(elem.firstChild.nodeType); // 3 => texto
10
11 // para el objeto de tipo documento, el tipo es 9
12 alert( document.nodeType ); // 9
13 </script>
14 </body>

```

En los scripts modernos, podemos usar `instanceof` y otras pruebas basadas en clases para ver el tipo de nodo, pero a veces `nodeType` puede ser más simple. Solo podemos leer `nodeType`, no cambiarlo.

## Tag: nodeName y tagName

Dado un nodo DOM, podemos leer su nombre de etiqueta en las propiedades de `nodeName` o `tagName`:

Por ejemplo:

```

1 alert( document.body.nodeName ); // BODY
2 alert( document.body.tagName ); // BODY

```



¿Hay alguna diferencia entre `tagName` y `nodeName`?

Claro, la diferencia se refleja en sus nombres, pero de hecho es un poco sutil.

- 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.

Por ejemplo, comparemos `tagName` y `nodeName` para `document` y un nodo de comentario:

```

1 <body><!-- comentario -->
2
3 <script>
4 // para comentarios
5 alert( document.body.firstChild.tagName ); // undefined (no es un elemento)
6 alert( document.body.firstChild.nodeName ); // #comment
7
8 // para documentos
9 alert( document.tagName ); // undefined (no es un elemento)
10 alert( document.nodeName ); // #document
11 </script>
12 </body>

```





Si solo tratamos con elementos, entonces podemos usar tanto `tagName` como `nodeName` – no hay diferencia.

### **i El nombre de la etiqueta siempre está en mayúsculas, excepto en el modo XML**

El navegador tiene dos modos de procesar documentos: HTML y XML. Por lo general, el modo HTML se usa para páginas web. El modo XML está habilitado cuando el navegador recibe un documento XML con el encabezado: `Content-Type: application/xml+xhtml`.

En el modo HTML, `tagName/nodeName` siempre está en mayúsculas: es `BODY` ya sea para `<body>` o `<BoDy>`.

En el modo XML, el caso se mantiene “tal cual”. Hoy en día, el modo XML rara vez se usa.

## innerHTML: los contenidos

La propiedad `innerHTML` permite obtener el HTML dentro del elemento como un string.

También podemos modificarlo. Así que es una de las formas más poderosas de cambiar la página.

El ejemplo muestra el contenido de `document.body` y luego lo reemplaza por completo:

```
1 <body>
2   <p>Un párrafo</p>
3   <div>Un div</div>
4
5   <script>
6     alert( document.body.innerHTML ); // leer el contenido actual
7     document.body.innerHTML = 'El nuevo BODY!'; // reemplazar
8   </script>
9
10 </body>
```



Podemos intentar insertar HTML no válido, el navegador corregirá nuestros errores:

```
1 <body>
2
3   <script>
4     document.body.innerHTML = '<b>prueba'; // olvidé cerrar la etiqueta
5     alert( document.body.innerHTML ); // <b>prueba</b> (arreglado)
6   </script>
7
8 </body>
```



### **i Los scripts no se ejecutan**

Si `innerHTML` inserta una etiqueta `<script>` en el documento, se convierte en parte de HTML, pero no se ejecuta.

## Cuidado: “innerHTML+=” hace una sobrescritura completa

Podemos agregar HTML a un elemento usando `elem.innerHTML+="more html"`.

Así:

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

Pero debemos tener mucho cuidado al hacerlo, porque lo que está sucediendo *no* es una adición, sino una sobrescritura completa.

Técnicamente, estas dos líneas hacen lo mismo:

```
1 elem.innerHTML += "...";
2 // es una forma más corta de escribir:
3 elem.innerHTML = elem.innerHTML + "..."
```

En otras palabras, `innerHTML+=` hace esto:

1. Se elimina el contenido antiguo.
2. 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..**

En el ejemplo de `chatDiv` arriba, la línea `chatDiv.innerHTML+="¿Cómo va?"` recrea el contenido HTML y recarga `smile.gif` (con la esperanza de que esté en caché). Si `chatDiv` tiene muchos otros textos e imágenes, entonces la recarga se vuelve claramente visible.

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`. Y si había un `<input>` con un texto ingresado por el visitante, entonces el texto será eliminado. Y así.

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

## outerHTML: HTML completo del elemento

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

He aquí un ejemplo:

```
1 <div id="elem">Hola <b>Mundo</b></div>
2
3 <script>
4   alert(elem.outerHTML); // <div id="elem">Hola <b>Mundo</b></div>
5 </script>
```



**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.

Considera el ejemplo:



```
1 <div>¡Hola, mundo!</div>
2
3 <script>
4   let div = document.querySelector('div');
5
6   // reemplaza div.outerHTML con <p>...</p>
7   div.outerHTML = '<p>Un nuevo elemento</p>'; // (*)
8
9   // ¡Guauu! ¡'div' sigue siendo el mismo!
10  alert(div.outerHTML); // <div>¡Hola, mundo!</div> (**)
11 </script>
```

Parece realmente extraño, ¿verdad?

En la línea `(*)` reemplazamos `div` con `<p>Un nuevo elemento</p>` . En el documento externo (el DOM) podemos ver el nuevo contenido en lugar del `<div>` . Pero, como podemos ver en la línea `(**)` , ¡el valor de la antigua variable `div` no ha cambiado!

La asignación `outerHTML` no modifica el elemento DOM (el objeto al que hace referencia, en este caso, la variable `'div'`), pero lo elimina del DOM e inserta el nuevo HTML en su lugar.

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

- `div` fue eliminado del documento.
- Otro fragmento de HTML `<p>Un nuevo elemento</p>` 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` .

Podemos escribir en `elem.outerHTML` , pero debemos tener en cuenta que no cambia el elemento en el que estamos escribiendo (`'elem'`). En su lugar, coloca el nuevo HTML en su lugar. Podemos obtener referencias a los nuevos elementos consultando el DOM.

## nodeValue/data: contenido del nodo de texto

La propiedad `innerHTML` solo es válida para los nodos de elementos.

Otros tipos de nodos, como los nodos de texto, tienen su contraparte: propiedades `nodeValue` y `data` . Estas dos son casi iguales para uso práctico, solo hay pequeñas diferencias de especificación. Entonces usaremos `data` , porque es más corto.

Un ejemplo de lectura del contenido de un nodo de texto y un comentario:

```

1 <body>
2   Hola
3   <!-- Comentario -->
4   <script>
5     let text = document.body.firstChild;
6     alert(text.data); // Hola
7
8     let comment = text.nextSibling;
9     alert(comment.data); // Comentario
10  </script>
11 </body>

```



Para los nodos de texto podemos imaginar una razón para leerlos o modificarlos, pero ¿por qué comentarios?

A veces, los desarrolladores incorporan información o instrucciones de plantilla en HTML, así:

```

1 <!-- if isAdmin -->
2   <div>¡Bienvenido, administrador!</div>
3 <!-- /if -->

```

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

## textContent: texto puro

El `textContent` proporciona acceso al *texto* dentro del elemento: solo texto, menos todas las `<tags>`.

Por ejemplo:

```

1 <div id="news">
2   <h1>¡Titular!</h1>
3   <p>¡Los marcianos atacan a la gente!</p>
4 </div>
5
6 <script>
7   // ¡Titular! ¡Los marcianos atacan a la gente!
8   alert(news.textContent);
9 </script>

```



Como podemos ver, solo se devuelve texto, como si todas las `<etiquetas>` fueran recortadas, pero el texto en ellas permaneció.

En la práctica, rara vez se necesita leer este tipo de texto.

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

Digamos que tenemos un string arbitrario, por ejemplo, ingresado por un usuario, y queremos mostrarlo.

- Con `innerHTML` lo tendremos insertado “como HTML”, con todas las etiquetas HTML.
- Con `textContent` lo tendremos insertado “como texto”, todos los símbolos se tratan literalmente.

Compara los dos:



```
1 <div id="elem1"></div>
2 <div id="elem2"></div>
3
4 <script>
5   let name = prompt("¿Cuál es tu nombre?", "<b>¡Winnie-Pooh!</b>");
6
7   elem1.innerHTML = name;
8   elem2.textContent = name;
9 </script>
```

1. El primer `<div>` obtiene el nombre “como HTML”: todas las etiquetas se convierten en etiquetas, por lo que vemos el nombre en negrita.
2. El segundo `<div>` obtiene el nombre “como texto”, así que literalmente vemos `<b>¡Winnie-Pooh!</b>` .

En la mayoría de los casos, esperamos el texto de un usuario y queremos tratarlo como texto. No queremos HTML inesperado en nuestro sitio. Una asignación a `textContent` hace exactamente eso.

## La propiedad “hidden”

El atributo “hidden” y la propiedad DOM especifican si el elemento es visible o no.

Podemos usarlo en HTML o asignarlo usando JavaScript, así:



```
1 <div>Ambos divs a continuación están ocultos</div>
2
3 <div hidden>Con el atributo "hidden"</div>
4
5 <div id="elem">JavaScript asignó la propiedad "hidden"</div>
6
7 <script>
8   elem.hidden = true;
9 </script>
```

Técnicamente, `hidden` funciona igual que `style="display:none"` . Pero es más corto de escribir.

Aquí hay un elemento parpadeante:



```
1 <div id="elem">Un elemento parpadeante</div>
2
3 <script>
4   setInterval(() => elem.hidden = !elem.hidden, 1000);
5 </script>
```

## 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`).
- ...y mucho más...

Por ejemplo:

```
1 <input type="text" id="elem" value="value">
2
3 <script>
4   alert(elem.type); // "text"
5   alert(elem.id); // "elem"
6   alert(elem.value); // value
7 </script>
```



La mayoría de los atributos HTML estándar tienen la propiedad DOM correspondiente, y podemos acceder a ella así.

Si queremos conocer la lista completa de propiedades admitidas para una clase determinada, podemos encontrarlas en la especificación. Por ejemplo, `HTMLInputElement` está documentado en <https://html.spec.whatwg.org/#htmlinputelement>.

O si nos gustaría obtenerlos rápidamente o estamos interesados en una especificación concreta del navegador, siempre podemos generar el elemento usando `console.dir(elem)` y leer las propiedades. O explora las “propiedades DOM” en la pestaña Elements de las herramientas de desarrollo del navegador.

## Resumen

Cada nodo DOM pertenece a una determinada clase. Las clases forman una jerarquía. El conjunto completo de propiedades y métodos proviene de la herencia.

Las propiedades principales del nodo DOM son:

### **nodeType**

Podemos usarla para ver si un nodo es un texto o un elemento. Tiene un valor numérico: **1** para elementos, **3** para nodos de texto y algunos otros para otros tipos de nodos. Solo lectura.

### **nodeName/tagName**

Para los elementos, nombre de la etiqueta (en mayúsculas a menos que esté en modo XML). Para los nodos que no son elementos, **nodeName** describe lo que es. Solo lectura.

### **innerHTML**

El contenido HTML del elemento. Puede modificarse.

### **outerHTML**

El HTML completo del elemento. 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/data**

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: HTML menos todas las `<tags>`. Escribir en él coloca el texto dentro del elemento, con todos los caracteres especiales y etiquetas tratados exactamente como texto. 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`.

Los nodos DOM también tienen otras propiedades dependiendo de su clase. Por ejemplo, los elementos `<input>` (`HTMLInputElement`) admiten `value`, `type`, mientras que los elementos `<a>` (`HTMLAnchorElement`) admiten `href`, etc. La mayoría de los atributos HTML estándar tienen una propiedad DOM correspondiente.

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

## Tareas

---

### Contar los descendientes

importancia: 5

Hay un árbol estructurado como `ul/li` anidado.

Escribe el código que para cada `<li>` muestra:

1. ¿Cuál es el texto dentro de él (sin el subárbol)?
2. El número de `<li>` anidados: todos los descendientes, incluidos los profundamente anidados.

[Demo en nueva ventana](#)

[Abrir un entorno controlado para la tarea.](#)

**solución**

---

### ¿Qué hay en nodeType?

importancia: 5

¿Qué muestra el script?

```
1 <html>
2
3 <body>
4   <script>
5     alert(document.body.lastChild.nodeType);
6   </script>
7 </body>
8
9 </html>
```

solución

---

## Etiqueta en comentario

importancia: 3

¿Qué muestra este código?

```
1 <script>
2   let body = document.body;
3
4   body.innerHTML = "<!--" + body.tagName + "-->";
5
6   alert( body.firstChild.data ); // ¿qué hay aquí?
7 </script>
```

solución

---

## ¿Dónde está el "document" en la jerarquía?

importancia: 4

¿A qué clase pertenece el `document` ?

¿Cuál es su lugar en la jerarquía DOM?

¿Hereda de `Node` o `Element` , o tal vez `HTMLElement` ?

solución



Lección anterior

Próxima lección







## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 1 de septiembre de 2022

# Atributos y propiedades

Cuando el navegador carga la página, "lee" (o "parser"(analiza en inglés)) el HTML y genera objetos DOM a partir de él. Para los nodos de elementos, la mayoría de los atributos HTML estándar se convierten automáticamente en propiedades de los objetos DOM.

Por ejemplo, si la etiqueta es `<body id="page">`, entonces el objeto DOM tiene `body.id="page"`.

¡Pero el mapeo de propiedades y atributos no es uno a uno! En este capítulo, prestaremos atención para separar estas dos nociones, para ver cómo trabajar con ellos, cuándo son iguales y cuándo son diferentes.

## Propiedades DOM

Ya hemos visto propiedades DOM integradas. Hay muchas. Pero técnicamente nadie nos limita, y si no hay suficientes, podemos agregar las nuestras.

Los nodos DOM son objetos JavaScript normales. Podemos alterarlos.

Por ejemplo, creemos una nueva propiedad en `document.body`:

```
1 document.body.myData = {  
2   name: 'Cesar',  
3   title: 'Emperador'  
4 };  
5  
6 alert(document.body.myData.title); // Emperador
```



También podemos agregar un método:

```
1 document.body.sayTagName = function() {  
2   alert(this.tagName);  
3 };  
4  
5 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:

```
1 Element.prototype.sayHi = function() {
```



```
2   alert(`Hola, yo soy ${this.tagName}`);
3   };
4
5   document.documentElement.sayHi(); // Hola, yo soy HTML
6   document.body.sayHi(); // Hola, yo soy BODY
```

Por lo tanto, las propiedades y métodos DOM se comportan igual que los objetos JavaScript normales:

- Pueden tener cualquier valor.
- Distingue entre mayúsculas y minúsculas (escribir `elem.nodeType`, no es lo mismo que `elem.NoDeTyPe`).

## Atributos HTML

En HTML, las etiquetas pueden tener atributos. Cuando el navegador analiza el HTML para crear objetos DOM para etiquetas, reconoce los atributos *estándar* y crea propiedades DOM a partir de ellos.

Entonces, cuando un elemento tiene `id` u otro atributo *estándar*, se crea la propiedad correspondiente. Pero eso no sucede si el atributo no es estándar.

Por ejemplo:

```
1 <body id="test" something="non-standard">
2   <script>
3     alert(document.body.id); // prueba
4     // el atributo no estándar no produce una propiedad
5     alert(document.body.something); // undefined
6   </script>
7 </body>
```



Tenga en cuenta que un atributo estándar para un elemento puede ser desconocido para otro. Por ejemplo, `"type"` es estándar para `<input>` (`HTMLInputElement`), pero no para `<body>` (`HTMLBodyElement`). Los atributos estándar se describen en la especificación para la clase del elemento correspondiente.

Aquí podemos ver esto:

```
1 <body id="body" type="...">
2   <input id="input" type="text">
3   <script>
4     alert(input.type); // text
5     alert(body.type); // undefined: Propiedad DOM no creada, porque no es estándar
6   </script>
7 </body>
```



Entonces, si un atributo no es estándar, no habrá una propiedad DOM para él. ¿Hay alguna manera de acceder a tales atributos?

Claro. Todos los atributos son accesibles usando los siguientes métodos:

- `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.

Estos métodos funcionan exactamente con lo que está escrito en HTML.

También se pueden leer todos los atributos usando `elem.attributes` : una colección de objetos que pertenecen a una clase integrada `Attr`, con propiedades `nombre` y `valor` .

Aquí hay una demostración de la lectura de una propiedad no estándar:



```
1 <body something="non-standard">
2   <script>
3     alert(document.body.getAttribute('something')); // no estándar
4   </script>
5 </body>
```

Los atributos HTML tienen las siguientes características:

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

Aquí hay una demostración extendida de cómo trabajar con atributos:



```
1 <body>
2   <div id="elem" about="Elephant"></div>
3
4   <script>
5     alert( elem.getAttribute('About') ); // (1) 'Elephant', leyendo
6
7     elem.setAttribute('Test', 123); // (2), escribiendo
8
9     alert( elem.outerHTML ); // (3), ver si el atributo está en HTML (sí)
10
11    for (let attr of elem.attributes) { // (4) listar todo
12      alert( `${attr.name} = ${attr.value}` );
13    }
14  </script>
15 </body>
```

Tenga en cuenta:

1. `getAttribute('About')` – la primera letra está en mayúscula aquí, y en HTML todo está en minúscula. Pero eso no importa: los nombres de los atributos no distinguen entre mayúsculas y minúsculas.
2. Podemos asignar cualquier cosa a un atributo, pero se convierte en un string. Así que aquí tenemos `"123"` como valor.
3. Todos los atributos, incluidos los que configuramos, son visibles en `outerHTML` .

4. La colección `attributes` es iterable y tiene todos los atributos del elemento (estándar y no estándar) como objetos con propiedades `name` y `value`.

## Sincronización de propiedad y atributo

Cuando cambia un atributo estándar, la propiedad correspondiente se actualiza automáticamente, y (con algunas excepciones) viceversa.

En el ejemplo a continuación, `id` se modifica como un atributo, y podemos ver que la propiedad también es cambiada. Y luego lo mismo al revés:

```
1  <input>
2
3  <script>
4    let input = document.querySelector('input');
5
6    // atributo -> propiedad
7    input.setAttribute('id', 'id');
8    alert(input.id); // id (actualizado)
9
10   // propiedad -> atributo
11   input.id = 'newId';
12   alert(input.getAttribute('id')); // newId (actualizado)
13 </script>
```

Pero hay exclusiones, por ejemplo, `input.value` se sincroniza solo del atributo a la propiedad (atributo → propiedad), pero no de regreso:

```
1  <input>
2
3  <script>
4    let input = document.querySelector('input');
5
6    // atributo -> propiedad
7    input.setAttribute('value', 'text');
8    alert(input.value); // text
9
10   // NO propiedad -> atributo
11   input.value = 'newValue';
12   alert(input.getAttribute('value')); // text (¡no actualizado!)
13 </script>
```

En el ejemplo anterior:

- Cambiar el atributo `value` actualiza la propiedad.
- Pero el cambio de propiedad no afecta al atributo.

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.

## 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:

```
1 <input id="input" type="checkbox" checked> checkbox
2
3 <script>
4   alert(input.getAttribute('checked')); // el valor del atributo es: string vac
5   alert(input.checked); // el valor de la propiedad es: true
6 </script>
```

Hay otros ejemplos. El atributo `style` es un string, pero la propiedad `style` es un objeto:

```
1 <div id="div" style="color:red;font-size:120%">Hola</div>
2
3 <script>
4   // string
5   alert(div.getAttribute('style')); // color:red;font-size:120%
6
7   // object
8   alert(div.style); // [object CSSStyleDeclaration]
9   alert(div.style.color); // red
10 </script>
```

La mayoría de las propiedades son strings.

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`.

Aquí hay un ejemplo:

```
1 <a id="a" href="#hola">link</a>
2 <script>
3   // atributo
4   alert(a.getAttribute('href')); // #hola
5
6   // propiedad
7   alert(a.href); // URL completa de http://site.com/page#hola
8 </script>
```

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

## Atributos no estándar, dataset

Cuando escribimos HTML, usamos muchos atributos estándar. Pero, ¿qué pasa con los no personalizados y personalizados? Primero, veamos si son útiles o no. ¿Para qué?

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

Como esto:

```
1  <!-- marque el div para mostrar "nombre" aquí -->
2  <div show-info="nombre"></div>
3  <!-- y "edad" aquí -->
4  <div show-info="edad"></div>
5
6  <script>
7    // el código encuentra un elemento con la marca y muestra lo que se solicita
8    let user = {
9      nombre: "Pete",
10     edad: 25
11   };
12
13   for(let div of document.querySelectorAll('[show-info]')) {
14     // inserta la información correspondiente en el campo
15     let field = div.getAttribute('show-info');
16     div.innerHTML = user[field]; // primero Pete en "nombre", luego 25 en "edad"
17   }
18 </script>
```

También se pueden usar para diseñar un elemento.

Por ejemplo, aquí para el estado del pedido se usa el atributo `order-state`:

```
1  <style>
2    /* los estilos se basan en el atributo personalizado "order-state" */
3    .order[order-state="nuevo"] {
4      color: green;
5    }
6
7    .order[order-state="pendiente"] {
8      color: blue;
9    }
10
11    .order[order-state="cancelado"] {
12      color: red;
13    }
```

```

14 </style>
15
16 <div class="order" order-state="nuevo">
17     Un nuevo pedido.
18 </div>
19
20 <div class="order" order-state="pendiente">
21     Un pedido pendiente.
22 </div>
23
24 <div class="order" order-state="cancelado">
25     Un pedido cancelado
26 </div>

```

¿Por qué sería preferible usar un atributo a tener clases como `.order-state-new`, `.order-state-pending`, `.order-state-canceled`?

Porque un atributo es más conveniente de administrar. El estado se puede cambiar tan fácil como:

```

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

```

Pero puede haber un posible problema con los atributos personalizados. ¿Qué sucede si usamos un atributo no estándar para nuestros propósitos y luego el estándar lo introduce y hace que haga algo? El lenguaje HTML está vivo, crece y cada vez hay más atributos que aparecen para satisfacer las necesidades de los desarrolladores. Puede haber efectos inesperados en tal caso.

Para evitar conflictos, existen atributos `data-*`.

**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`.

Como esto:

```

1 <body data-about="Elefante">
2 <script>
3     alert(document.body.dataset.about); // Elefante
4 </script>

```



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

Aquí hay un ejemplo reescrito de "estado del pedido":

```

1 <style>
2     .order[data-order-state="nuevo"] {
3         color: green;

```





```

4    }
5
6    .order[data-order-state="pendiente"] {
7        color: blue;
8    }
9
10   .order[data-order-state="cancelado"] {
11       color: red;
12   }
13 </style>
14
15 <div id="order" class="order" data-order-state="nuevo">
16     Una nueva orden.
17 </div>
18
19 <script>
20     // leer
21     alert(order.dataset.orderState); // nuevo
22
23     // modificar
24     order.dataset.orderState = "pendiente"; // (*)
25 </script>

```

El uso de los atributos `data-*` es una forma válida y segura de pasar datos personalizados.

Tenga en cuenta que no solo podemos leer, sino también modificar los atributos de datos. Luego, CSS actualiza la vista en consecuencia: en el ejemplo anterior, la última línea `(*)` cambia el color a azul.

## Resumen

- 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, cuando necesitamos exactamente atributos, 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".

## ✓ Tareas

---

### Obtén en atributo

importancia: 5

Escribe el código para obtener el atributo **data-widget-name** del documento y leer su valor.



```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5   <div data-widget-name="menu">Elige el genero</div>
6
7   <script>
8     /* Tu código */
9   </script>
10 </body>
11 </html>
```

solución

---

### Haz los enlaces externos naranjas

importancia: 3

Haz todos los enlaces externos de color orange alterando su propiedad **style**.

Un link es externo si:

- Su **href** tiene **://**
- Pero no comienza con **http://internal.com**.

Ejemplo:



```
1 <a name="list">the list</a>
2 <ul>
3   <li><a href="http://google.com">http://google.com</a></li>
4   <li><a href="/tutorial">/tutorial.html</a></li>
5   <li><a href="local/path">local/path</a></li>
```

```
6 <li><a href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
7 <li><a href="http://nodejs.org">http://nodejs.org</a></li>
8 <li><a href="http://internal.com/test">http://internal.com/test</a></li>
9 </ul>
10
11 <script>
12 // establecer un estilo para un enlace
13 let link = document.querySelector('a');
14 link.style.color = 'orange';
15 </script>
```

El resultado podría ser:

La lista:

- <http://google.com>
- </tutorial.html>
- <local/path>
- <ftp://ftp.com/my.zip>
- <http://nodejs.org>
- <http://internal.com/test>

Abrir un entorno controlado para la tarea.

solución



Lección anterior

Próxima lección



Compartir



Mapa del Tutorial

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen](#)...)



🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 3 de julio de 2022

# 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.

## Ejemplo: mostrar un mensaje

Hagamos una demostración usando un ejemplo. Añadiremos un mensaje que se vea más agradable que un `alert`.

Así es como se verá:

```
1 <style>
2 .alert {
3   padding: 15px;
4   border: 1px solid #d6e9c6;
5   border-radius: 4px;
6   color: #3c763d;
7   background-color: #dff0d8;
8 }
9 </style>
10
11 <div class="alert">
12   <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
13 </div>
```

**¡Hola!** Usted ha leído un importante mensaje.

Eso fue el ejemplo HTML. Ahora creemos el mismo `div` con JavaScript (asumiendo que los estilos ya están en HTML/CSS).

## Creando un elemento

Para crear nodos DOM, hay dos métodos:

**`document.createElement(tag)`**

Creará un nuevo *nodo elemento* con la etiqueta HTML dada:

```
1 let div = document.createElement('div');
```

### `document.createTextNode(text)`

Crea un nuevo *nodo texto* con el texto dado:

```
1 let textNode = document.createTextNode('Aquí estoy');
```

La mayor parte del tiempo necesitamos crear nodos de elemento, como el `div` para el mensaje.

## Creando el mensaje

Crear el `div` de mensaje toma 3 pasos:

```
1 // 1. Crear elemento <div>
2 let div = document.createElement('div');
3
4 // 2. Establecer su clase a "alert"
5 div.className = "alert";
6
7 // 3. Agregar el contenido
8 div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje."
```

Hemos creado el elemento. Pero hasta ahora solamente está en una variable llamada `div`, no aún en la página, y no la podemos ver.

## Métodos de inserción

Para hacer que el `div` aparezca, necesitamos insertarlo en algún lado dentro de `document`. Por ejemplo, en el elemento `<body>`, referenciado por `document.body`.

Hay un método especial `append` para ello: `document.body.append(div)`.

El código completo:

```
1 <style>
2 .alert {
3   padding: 15px;
4   border: 1px solid #d6e9c6;
5   border-radius: 4px;
6   color: #3c763d;
7   background-color: #dff0d8;
8 }
9 </style>
10
```



```

11 <script>
12   let div = document.createElement('div');
13   div.className = "alert";
14   div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje";
15
16   document.body.append(div);
17 </script>

```

Aquí usamos el método `append` sobre `document.body`, pero podemos llamar `append` sobre cualquier elemento para poner otro elemento dentro de él. Por ejemplo, podemos añadir algo a `<div>` llamando `div.append(anotherElement)`.

Aquí hay más métodos de inserción, ellos especifican diferentes lugares donde insertar:

- `node.append(...nodos o strings)` – agrega nodos o strings *al final* de `node`,
- `node.prepend(...nodos o strings)` – inserta nodos o strings *al principio* de `node`,
- `node.before(...nodos o strings)` – inserta nodos o strings *antes* de `node`,
- `node.after(...nodos o strings)` – inserta nodos o strings *después* de `node`,
- `node.replaceWith(...nodos o strings)` – reemplaza `node` con los nodos o strings dados.

Los argumentos de estos métodos son una lista arbitraria de lo que se va a insertar: nodos DOM o strings de texto (estos se vuelven nodos de texto automáticamente).

Veámoslo en acción.

Aquí tenemos un ejemplo del uso de estos métodos para agregar items a una lista y el texto antes/después de él:

```

1 <ol id="ol">
2   <li>0</li>
3   <li>1</li>
4   <li>2</li>
5 </ol>
6
7 <script>
8   ol.before('before'); // inserta el string "before" antes de <ol>
9   ol.after('after'); // inserta el string "after" después de <ol>
10
11   let liFirst = document.createElement('li');
12   liFirst.innerHTML = 'prepend';
13   ol.prepend(liFirst); // inserta liFirst al principio de <ol>
14
15   let liLast = document.createElement('li');
16   liLast.innerHTML = 'append';
17   ol.append(liLast); // inserta liLast al final de <ol>
18 </script>

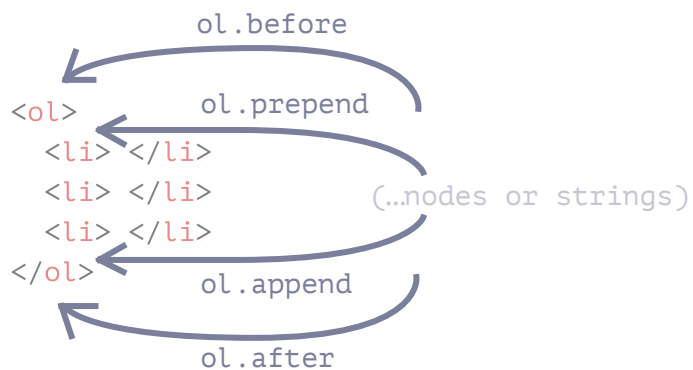
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Aquí la representación visual de lo que hacen los métodos:



Entonces la lista final será:

```
1 before
2 <ol id="ol">
3   <li>prepend</li>
4   <li>0</li>
5   <li>1</li>
6   <li>2</li>
7   <li>append</li>
8 </ol>
9 after
```

Como dijimos antes, estos métodos pueden insertar múltiples nodos y piezas de texto en un simple llamado.

Por ejemplo, aquí se insertan un string y un elemento:

```
1 <div id="div"></div>
2 <script>
3   div.before('<p>Hola</p>', document.createElement('hr'));
4 </script>
```

Nota que el texto es insertado "como texto" y no "como HTML", escapando apropiadamente los caracteres como `<`, `>`.

Entonces el HTML final es:



```

1 <p>&gt;Hola</p>;
2 <hr>
3 <div id="div"></div>

```



En otras palabras, los strings son insertados en una manera segura, tal como lo hace `elem.textContent`.

Entonces, estos métodos solo pueden usarse para insertar nodos DOM como piezas de texto.

Pero ¿y si queremos insertar un string HTML “como html”, con todas las etiquetas y demás funcionando, de la misma manera que lo hace `elem.innerHTML`?

## insertAdjacentHTML/Text/Element

Para ello podemos usar otro métodos, muy versátil: `elem.insertAdjacentHTML(where, html)`.

El primer parámetro es un palabra código que especifica dónde insertar relativo a `elem`. Debe ser uno de los siguientes:

- `"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`

El segundo parámetro es un string HTML, que es insertado “como HTML”.

Por ejemplo:

```

1 <div id="div"></div>
2 <script>
3   div.insertAdjacentHTML('beforebegin', '<p>Hola</p>');
4   div.insertAdjacentHTML('afterend', '<p>Adiós</p>');
5 </script>

```



...resulta en:

```

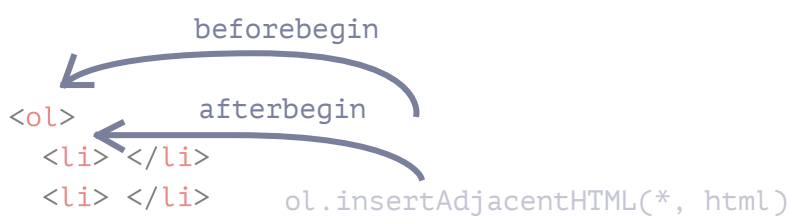
1 <p>Hola</p>
2 <div id="div"></div>
3 <p>Adiós</p>

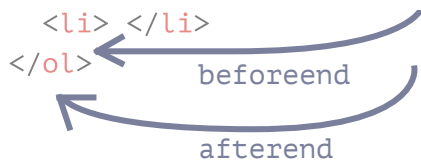
```



Así es como podemos añadir HTML arbitrario a la página.

Aquí abajo, la imagen de las variantes de inserción:





Fácilmente podemos notar similitudes entre esta imagen y la anterior. Los puntos de inserción son los mismos, pero este método inserta HTML.

El método tiene dos hermanos:

- `elem.insertAdjacentText(where, text)` – la misma sintaxis, pero un string de `texto` es insertado “como texto” en vez de HTML,
- `elem.insertAdjacentElement(where, elem)` – la misma sintaxis, pero inserta un elemento.

Ellos existen principalmente para hacer la sintaxis “uniforme”. En la práctica, solo `insertAdjacentHTML` es usado la mayor parte del tiempo. Porque para elementos y texto, tenemos los métodos `append/prepend/before/after` : son más cortos para escribir y pueden insertar piezas de texto y nodos.

Entonces tenemos una alternativa para mostrar un mensaje:

```
1 <style>
2 .alert {
3   padding: 15px;
4   border: 1px solid #d6e9c6;
5   border-radius: 4px;
6   color: #3c763d;
7   background-color: #dff0d8;
8 }
9 </style>
10
11 <script>
12   document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
13     <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
14   </div>`);
15 </script>
```

## Eliminación de nodos

Para quitar un nodo, tenemos el método `node.remove()` .

Hagamos que nuestro mensaje desaparezca después de un segundo:

```
1 <style>
2 .alert {
3   padding: 15px;
4   border: 1px solid #d6e9c6;
5   border-radius: 4px;
6   color: #3c763d;
7   background-color: #dff0d8;
8 }
```

```

9   </style>
10
11  <script>
12    let div = document.createElement('div');
13    div.className = "alert";
14    div.innerHTML = "<strong>¡Hola!</strong> Usted ha leído un importante mensaje";
15
16    document.body.append(div);
17    setTimeout(() => div.remove(), 1000);
18  </script>

```

Nota que si queremos *mover* un elemento a un nuevo lugar, no hay necesidad de quitarlo del viejo.

**Todos los métodos de inserción automáticamente quitan el nodo del lugar viejo.**

Por ejemplo, intercambiamos elementos:

```

1  <div id="first">Primero</div>
2  <div id="second">Segundo</div>
3  <script>
4    // no hay necesidad de llamar "remove"
5    second.after(first); // toma #second y después inserta #first
6  </script>

```

## Clonando nodos: cloneNode

¿Cómo insertar un mensaje similar más?

Podríamos hacer una función y poner el código allí. Pero la alternativa es *clonar* el `div` existente, y modificar el texto dentro si es necesario.

A veces, cuando tenemos un elemento grande, esto es más simple y rápido.

- La llamada `elem.cloneNode(true)` crea una clonación "profunda" del elemento, con todos los atributos y subelementos. Si llamamos `elem.cloneNode(false)`, la clonación se hace sin sus elementos hijos.

Un ejemplo de copia del mensaje:

```

1  <style>
2  .alert {
3    padding: 15px;
4    border: 1px solid #d6e9c6;
5    border-radius: 4px;
6    color: #3c763d;
7    background-color: #dff0d8;
8  }
9  </style>
10
11  <div class="alert" id="div">

```

```

12   <strong>¡Hola!</strong> Usted ha leído un importante mensaje.
13 </div>
14
15 <script>
16   let div2 = div.cloneNode(true); // clona el mensaje
17   div2.querySelector('strong').innerHTML = '¡Adiós!'; // altera el clon
18
19   div.after(div2); // muestra el clon después del div existente
20 </script>

```

## DocumentFragment

**DocumentFragment** es un nodo DOM especial que sirve como contenedor para trasladar listas de nodos.

Podemos agregarle nodos, pero cuando lo insertamos en algún lugar, lo que se inserta es su contenido.

Por ejemplo, `getListContent` de abajo genera un fragmento con ítems `<li>`, que luego son insertados en `<ul>`:

```

1  <ul id="ul"></ul>
2
3  <script>
4  function getListContent() {
5    let fragment = new DocumentFragment();
6
7    for(let i=1; i<=3; i++) {
8      let li = document.createElement('li');
9      li.append(i);
10     fragment.append(li);
11   }
12
13   return fragment;
14 }
15
16 ul.append(getListContent()); // (*)
17 </script>

```

Nota que a la última línea (\*) añadimos **DocumentFragment**, pero este despliega su contenido. Entonces la estructura resultante será:

```

1  <ul>
2    <li>1</li>
3    <li>2</li>
4    <li>3</li>
5  </ul>

```

Es raro que **DocumentFragment** se use explícitamente. ¿Por qué añadir un tipo especial de nodo si en su lugar podemos devolver un array de nodos? El ejemplo reescrito:



```
1 <ul id="ul"></ul>
2
3 <script>
4 function getListContent() {
5   let result = [];
6
7   for(let i=1; i<=3; i++) {
8     let li = document.createElement('li');
9     li.append(i);
10    result.push(li);
11  }
12
13  return result;
14 }
15
16 ul.append(...getListContent()); // append + el operador "..." = ¡amigos!
17 </script>
```

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

## Métodos de la vieja escuela para insertar/quitar



### Vieja escuela

Esta información ayuda a entender los viejos scripts, pero no es necesaria para nuevos desarrollos.

Hay también métodos de manipulación de DOM de “vieja escuela”, existentes por razones históricas.

Estos métodos vienen de realmente viejos tiempos. No hay razón para usarlos estos días, ya que los métodos modernos como `append`, `prepend`, `before`, `after`, `remove`, `replaceWith`, son más flexibles.

La única razón por la que los listamos aquí es porque podrías encontrarlos en viejos scripts:

#### `parentElem.appendChild(node)`

Añade `node` como último hijo de `parentElem`.

El siguiente ejemplo agrega un nuevo `<li>` al final de `<ol>`:



```
1 <ol id="list">
2   <li>0</li>
3   <li>1</li>
4   <li>2</li>
5 </ol>
6
7 <script>
8   let newLi = document.createElement('li');
9   newLi.innerHTML = '¡Hola, mundo!';
10
```

```
11 list.appendChild(newLi);
12 </script>
```

### **parentElem.insertBefore(node, nextSibling)**

Inserta `node` antes de `nextSibling` dentro de `parentElem`.

El siguiente código inserta un nuevo ítem de lista antes del segundo `<li>`:

```
1 <ol id="list">
2   <li>0</li>
3   <li>1</li>
4   <li>2</li>
5 </ol>
6 <script>
7   let newLi = document.createElement('li');
8   newLi.innerHTML = '¡Hola, mundo!';
9
10  list.insertBefore(newLi, list.children[1]);
11 </script>
```



Para insertar `newLi` como primer elemento, podemos hacerlo así:

```
1 list.insertBefore(newLi, list.firstChild);
```

### **parentElem.replaceChild(node, oldChild)**

Reemplaza `oldChild` con `node` entre los hijos de `parentElem`.

### **parentElem.removeChild(node)**

Quita `node` de `parentElem` (asumiendo que `node` es su hijo).

El siguiente ejemplo quita el primer `<li>` de `<ol>`:

```
1 <ol id="list">
2   <li>0</li>
3   <li>1</li>
4   <li>2</li>
5 </ol>
6
7 <script>
8   let li = list.firstChild;
9   list.removeChild(li);
10 </script>
```



Todos estos métodos devuelven el nodo insertado/quitado. En otras palabras, `parentElem.appendChild(node)` devuelve `node`. Pero lo usual es que el valor no se use y solo ejecutemos el

método.

## Una palabra acerca de “document.write”

Hay uno más, un método muy antiguo para agregar algo a una página web: `document.write`.

La sintaxis:

```
1 <p>En algún lugar de la página...</p>
2 <script>
3   document.write('<b>Saludos de JS</b>');
4 </script>
5 <p>Fin</p>
```



El llamado a `document.write(html)` escribe el `html` en la página “aquí y ahora”. El string `html` puede ser generado dinámicamente, así que es muy flexible. Podemos usar JavaScript para crear una página completa al vuelo y escribirla.

El método viene de tiempos en que no había DOM ni estándares... Realmente viejos tiempos. Todavía vive, porque hay scripts que lo usan.

En scripts modernos rara vez lo vemos, por una importante limitación:

**El llamado a `document.write` solo funciona mientras la página está cargando.**

Si la llamamos después, el contenido existente del documento es borrado.

Por ejemplo:

```
1 <p>Después de un segundo el contenido de esta página será reemplazado...</p>
2 <script>
3   // document.write después de 1 segundo
4   // eso es después de que la página cargó, entonces borra el contenido existente
5   setTimeout(() => document.write('<b>...Por esto.</b>'), 1000);
6 </script>
```



Así que es bastante inusable en el estado “after loaded” (después de cargado), al contrario de los otros métodos DOM que cubrimos antes.

Ese es el punto en contra.

También tiene un punto a favor. Técnicamente, 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” involucrada. Escribe directamente en el texto de la página mientras el DOM ni siquiera está construido.

Entonces: si necesitamos agregar un montón de texto en HTML dinámicamente, estamos en la fase de carga de página, y la velocidad es importante, esto puede ayudar. Pero en la práctica estos requerimientos raramente vienen juntos. Así que si vemos este método en scripts, probablemente sea solo porque son viejos.

# Resumen

- Métodos para crear nuevos nodos:
  - `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.
- Inserción y eliminación:
  - `node.append(...nodes or strings)` – inserta en `node`, al final
  - `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`
  - `node.replaceWith(...nodes or strings)` – reemplaza `node`
  - `node.remove()` – quita el `node`.

Los strings de texto son insertados "como texto".

- También hay métodos "de vieja escuela":
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

Todos estos métodos devuelven `node`.

- Dado cierto HTML en `html`, `elem.insertAdjacentHTML(where, html)` lo inserta dependiendo del valor `where`:
  - "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.

- Para agregar HTML a la página antes de que haya terminado de cargar:
  - `document.write(html)`

Después de que la página fue cargada tal llamada borra el documento. Puede verse principalmente en scripts viejos.

## ✓ Tareas

---

### createTextNode vs innerHTML vs textContent

importancia: 5



Tenemos un elemento DOM vacío `elem` y un string `text`.

¿Cuáles de estos 3 comandos harán exactamente lo mismo?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`
3. `elem.textContent = text`

solución

---

## Limpiar el elemento

importancia: 5

Crea una función `clear(elem)` que remueva todo del elemento.

```
1 <ol id="elem">
2   <li>Hola</li>
3   <li>mundo</li>
4 </ol>
5
6 <script>
7   function clear(elem) { /* tu código */ }
8
9   clear(elem); // borra la lista
10 </script>
```



solución

---

## Por qué "aaa" permanece?

importancia: 1

En el ejemplo de abajo, la llamada `table.remove()` quita la tabla del documento.

Pero si la ejecutas, puedes ver que el texto "aaa" es aún visible.

¿Por qué ocurre esto?

```
1 <table id="table">
2   aaa
3   <tr>
4     <td>Test</td>
5   </tr>
6 </table>
7
8 <script>
9   alert(table); // la tabla, tal como debería ser
```



```
10
11   table.remove();
12   // ¿Por qué aún está "aaa" en el documento?
13 </script>
```

solución

---

## Crear una lista

importancia: 4

Escribir una interfaz para crear una lista de lo que ingresa un usuario.

Para cada item de la lista:

1. Preguntar al usuario acerca del contenido usando `prompt` .
2. Crear el `<li>` con ello y agregarlo a `<ul>` .
3. Continuar hasta que el usuario cancela el ingreso (presionando `Esc` o con un ingreso vacío).

Todos los elementos deben ser creados dinámicamente.

Si el usuario ingresa etiquetas HTML, deben ser tratadas como texto.

[Demo en nueva ventana](#)

solución

---

## Crea un árbol desde el objeto

importancia: 5

Escribe una función `createTree` que crea una lista ramificada `ul/li` desde un objeto ramificado.

Por ejemplo:

```
1  let data = {
2    "Fish": {
3      "trout": {},
4      "salmon": {}
5    },
6
7    "Tree": {
8      "Huge": {
9        "sequoia": {},
10       "oak": {}
11     },
12     "Flowering": {
13       "apple tree": {},
14       "magnolia": {}
15     }
16   }
```

```
15     }  
16   }  
17 };
```

La sintaxis:

```
1 let container = document.getElementById('container');  
2 createTree(container, data); // crea el árbol en el contenedor
```

El árbol resultante debe verse así:

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

Elige una de estas dos formas para resolver esta tarea:

1. Crear el HTML para el árbol y entonces asignarlo a `container.innerHTML` .
2. Crear los nodos del árbol y añadirlos con métodos DOM.

Sería muy bueno que hicieras ambas soluciones.

P.S. El árbol no debe tener elementos “extras” como `<ul></ul>` vacíos para las hojas.

[Abrir un entorno controlado para la tarea.](#)

**solución**

---

## Mostrar descendientes en un árbol

importancia: 5

Hay un árbol organizado como ramas `ul/li` .

Escribe el código que agrega a cada `<li>` el número de su descendientes. No cuentes las hojas (nodos sin hijos).

El resultado:

- Animals [9]
  - Mammals [4]
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other [3]
    - Snakes
    - Birds
    - Lizards
- Fishes [5]
  - Aquarium [2]
    - Guppy
    - Angelfish
  - Sea [1]
    - Sea trout

Abrir un entorno controlado para la tarea.

solución

## Crea un calendario

importancia: 4

Escribe una función `createCalendar(elem, year, month)` .

Su llamado debe crear un calendario para el año y mes dados y ponerlo dentro de `elem` .

El calendario debe ser una tabla, donde una semana es `<tr>` , y un día es `<td>` . Los encabezados de la tabla deben ser `<th>` con los nombres de los días de la semana: el primer día debe ser "lunes" y así hasta "domingo".

Por ejemplo, `createCalendar(cal, 2012, 9)` debe generar en el elemento `cal` el siguiente calendario:

MO	TU	WE	TH	FR	SA	SU
					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

P.S. Para esta tarea es suficiente generar el calendario, no necesita aún ser cliqueable.

Abrir un entorno controlado para la tarea.

solución

## Reloj coloreado con setInterval

importancia: 4

Crea un reloj coloreado como aquí:

hh:mm:ss

Start

Stop

Usa HTML/CSS para el estilo, JavaScript solamente actualiza la hora en elements.

[Abrir un entorno controlado para la tarea.](#)

solución

## Inserta el HTML en la lista

importancia: 5

Escribe el código para insertar `<li>2</li><li>3</li>` entre dos `<li>` aquí:

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

solución

## Ordena la tabla

importancia: 5

Tenemos una tabla:

```
1 <table>
2 <thead>
3   <tr>
4     <th>Name</th><th>Surname</th><th>Age</th>
5   </tr>
6 </thead>
7 <tbody>
8   <tr>
9     <td>John</td><td>Smith</td><td>10</td>
10  </tr>
```



```
11 <tr>
12   <td>Pete</td><td>Brown</td><td>15</td>
13 </tr>
14 <tr>
15   <td>Ann</td><td>Lee</td><td>5</td>
16 </tr>
17 <tr>
18   <td>...</td><td>...</td><td>...</td>
19 </tr>
20 </tbody>
21 </table>
```

Puede haber más filas en ella.

Escribe el código para ordenarla por la columna "name" .

[Abrir un entorno controlado para la tarea.](#)

solución



Lección anterior

Próxima lección



Compartir



Mapa del Tutorial

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))

🏠 → El navegador: Documentos, Eventos e Interfaces → Documento

📅 24 de octubre de 2022

## Estilos y clases

Antes de profundizar en cómo JavaScript maneja las clases y los estilos, hay una regla importante. Aunque es lo suficientemente obvio, aún tenemos que mencionarlo.

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

1. Crear una clase `css` y agregarla: `<div class="...">`
2. Escribir las propiedades directamente en `style`: `<div style="...">`.

JavaScript puede modificar ambos, clases y las propiedades de `style`.

Nosotros deberíamos preferir las clases `css` en lugar de `style`. Este último solo debe usarse si las clases “no pueden manejarlo”.

Por ejemplo, `style` es aceptable si nosotros calculamos las coordenadas de un elemento dinámicamente y queremos establecer estas desde JavaScript, así:

```
1 let top = /* cálculos complejos */;  
2 let left = /* cálculos complejos */;  
3  
4 elem.style.left = left; // ej. '123px', calculado en tiempo de ejecución  
5 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.

## className y classList

Cambiar una clase es una de las acciones más utilizadas.

En la antigüedad, había una limitación en JavaScript: una palabra reservada como `"class"` no podía ser una propiedad de un objeto. Esa limitación no existe ahora, pero en ese momento era imposible tener una propiedad `"class"`, como `elem.class`.

Entonces para clases de similares propiedades, `"className"` fue introducido: el `elem.className` corresponde al atributo `"class"`.

Por ejemplo:

```
1 <body class="main page">  
2   <script>
```

```
3     alert(document.body.className); // página principal
4   </script>
5 </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`.

El `elem.classList` es un objeto especial con métodos para agregar, eliminar y alternar ( `add/remove/toggle` ) una sola clase.

Por ejemplo:

```
1 <body class="main page">
2   <script>
3     // agregar una clase
4     document.body.classList.add('article');
5
6     alert(document.body.className); // clase "article" de la página principal
7   </script>
8 </body>
```

Entonces podemos trabajar con ambos: todas las clases como una cadena usando `className` o con clases individuales usando `classList`. Lo que elijamos depende de nuestras necesidades.

Métodos de `classList`:

- `elem.classList.add/remove("class")` – agrega o remueve la clase.
- `elem.classList.toggle("class")` – agrega la clase si no existe, si no la remueve.
- `elem.classList.contains("class")` – verifica si tiene la clase dada, devuelve `true/false`.

Además, `classList` es iterable, entonces podemos listar todas las clases con `for..of`, así:

```
1 <body class="main page">
2   <script>
3     for (let name of document.body.classList) {
4       alert(name); // main y luego page
5     }
6   </script>
7 </body>
```

## 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"` funciona igual que si tuviéramos en el atributo `style` una cadena con `width:100px`.



Para propiedades de varias palabras se usa `camelCase` :

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

Por ejemplo:

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

### **i** Propiedades prefijadas

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

Por ejemplo:

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

## Reseteando la propiedad `style`

A veces queremos asignar una propiedad de estilo y luego removerla.

Por ejemplo, para ocultar un elemento, podemos establecer `elem.style.display = "none"` .

Luego, más tarde, es posible que queramos remover `style.display` como si no estuviera establecido. En lugar de `delete elem.style.display` deberíamos asignarle una cadena vacía: `elem.style.display = ""` .

```
1 // si ejecutamos este código, el <body> parpadeará
2 document.body.style.display = "none"; // ocultar
3
4 setTimeout(() => document.body.style.display = "", 1000); // volverá a lo normal
```

Si establecemos `style.display` como una cadena vacía, entonces el navegador aplica clases y estilos CSS incorporados normalmente por el navegador, como si no existiera tal `style.display` .

También hay un método especial para eso, `elem.style.removeProperty('style property')` . Así, podemos quitar una propiedad:

```
1 document.body.style.background = 'red'; //establece background a rojo
2
3 setTimeout(() => document.body.style.removeProperty('background'), 1000); // quita
```

### **i** Reescribir todo usando `style.cssText`

Normalmente, podemos usar `style.*` para asignar propiedades de estilo individuales. No podemos establecer todo el estilo como `div.style="color: red; width: 100px"`, porque `div.style` es un objeto y es solo de lectura.

Para establecer todo el estilo como una cadena, hay una propiedad especial: `style.cssText` :

```
1 <div id="div">Button</div>
2
3 <script>
4   // podemos establecer estilos especiales con banderas como "important"
5   div.style.cssText=`color: red !important;
6     background-color: yellow;
7     width: 100px;
8     text-align: center;
9   `;
10
11   alert(div.style.cssText);
12 </script>
```

Esta propiedad es rara vez usada, porque tal asignación remueve todo los estilos: no agrega estilos sino que los reemplaza en su totalidad. Ocasionalmente podría eliminar algo necesario. Pero podemos usarlo de manera segura para nuevos elementos, cuando sabemos que no vamos a eliminar un estilo existente.

Lo mismo se puede lograr estableciendo un atributo: `div.setAttribute('style', 'color: red...')`.

## 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:

```
1 <body>
2   <script>
3     // ¡no funciona!
4     document.body.style.margin = 20;
5     alert(document.body.style.margin); // '' (cadena vacía, la asignación es incorrecta)
6
7     // ahora agregamos la unidad CSS (px) y esta sí funciona
8     document.body.style.margin = '20px';
9     alert(document.body.style.margin); // 20px
10
11     alert(document.body.style.marginTop); // 20px
12     alert(document.body.style.marginLeft); // 20px
13
```

```
14 </script>
    </body>
```

Tenga en cuenta: el navegador “desempaqueta” la propiedad `style.margin` en las últimas líneas e infiere `style.marginLeft` y `style.marginTop` de eso.

## Estilos calculados: `getComputedStyle`

Entonces, modificar un estilo es fácil. ¿Pero cómo *leerlo*?

Por ejemplo, queremos saber el tamaño, los márgenes, el color de un elemento. ¿Cómo hacerlo?

**La propiedad `style` solo opera en el valor del atributo `"style"`, sin ninguna cascada de `css`.**

Entonces no podemos leer ninguna clase CSS usando `elem.style`.

Por ejemplo, aquí `style` no ve el margen:

```
1 <head>
2   <style> body { color: red; margin: 5px } </style>
3 </head>
4 <body>
5
6   El texto en rojo
7   <script>
8     alert(document.body.style.color); // vacío
9     alert(document.body.style.marginTop); // vacío
10  </script>
11 </body>
```

Pero si necesitamos incrementar el margen a `20px` ? vamos el querer el valor de la misma.

Hay otro método para eso: `getComputedStyle`.

La sintaxis es:

```
1 getComputedStyle(element, [pseudo])
```

### **element**

Elemento del cual se va a leer el valor.

### **pseudo**

Un pseudo-elemento es requerido, por ejemplo `::before`. Una cadena vacía o sin argumento significa el elemento mismo.

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

Por ejemplo:



```
1 <head>
2   <style> body { color: red; margin: 5px } </style>
3 </head>
4 <body>
5
6   <script>
7     let computedStyle = getComputedStyle(document.body);
8
9     // ahora podemos leer los márgenes y el color de ahí
10
11     alert( computedStyle.marginTop ); // 5px
12     alert( computedStyle.color ); // rgb(255, 0, 0)
13   </script>
14
15 </body>
```

### Valores calculado y resueltos

Hay dos conceptos en `CSS`:

1. Un estilo *calculado* es el valor final de aplicar todas las reglas y herencias CSS, como resultado de la cascada CSS. Puede parecer `height:1em` o `font-size:125%`.
2. Un estilo *resuelto* 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`. Para las propiedades de geometría los valores resueltos pueden tener un punto flotante, como `width:50.5px`.

Hace mucho tiempo `getComputedStyle` fue creado para obtener los valores calculados, pero los valores resueltos son muchos más convenientes, y el estándar cambió.

Así que hoy en día `getComputedStyle` en realidad devuelve el valor resuelto de la propiedad, usualmente en `px` para geometría.

### 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`. De lo contrario, no se garantiza el resultado correcto.

Por ejemplo, si hay propiedades `paddingLeft/paddingTop`, entonces ¿qué deberíamos obtener de `getComputedStyle(elem).padding`? ¿Nada, o tal vez un valor "generado" de los paddings? No hay una regla estándar aquí.

## ¡Los estilos aplicados a los enlaces `:visited` están ocultos!

Los enlaces visitados pueden ser coloreados usando la pseudo-clase `:visited` de CSS.

Pero `getComputedStyle` no da 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.

JavaScript no puede ver los estilos aplicados por `:visited`. También hay una limitación 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.

## Resumen

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).

## Tareas

### Crear una notificación

importancia: 5

Escribir una función `showNotification(options)` que cree una notificación: `<div class="notification">` con el contenido dado. La notificación debería desaparecer automáticamente después de 1.5 segundos.

Las opciones son:

```
1 // muestra un elemento con el texto "Hello" cerca de la parte superior de la v
2 showNotification({
3   top: 10, // 10px desde la parte superior de la ventana (por defecto es 0px)
4   right: 10, // 10px desde el borde derecho de la ventana (por defecto es 0px)
5   html: "Hello!", // el HTML de la notificación
6 }
```

```
7   className: "welcome" // una clase adicional para el "div" (opcional)
   });
```

[Demo en nueva ventana](#)

Usar posicionamiento CSS para mostrar el elemento en las coordenadas (top/right) dadas. El documento tiene los estilos necesarios.

[Abrir un entorno controlado para la tarea.](#)

**solución**



[Lección anterior](#)

[Próxima lección](#)



Compartir



[Mapa del Tutorial](#)

## Comentarios

- Si tiene sugerencias sobre qué mejorar, por favor [enviar una propuesta de GitHub](#) o una solicitud de extracción en lugar de comentar.
- Si no puede entender algo en el artículo, por favor explique.
- Para insertar algunas palabras de código, use la etiqueta `<code>` , para varias líneas – envolverlas en la etiqueta `<pre>` , para más de 10 líneas – utilice un entorno controlado (sandbox) ([plnkr](#), [jsbin](#), [codepen...](#))