ENTORNO DEL NAVEGADOR (DOM)

2º DAW - Desarrollo Web en Entorno Cliente

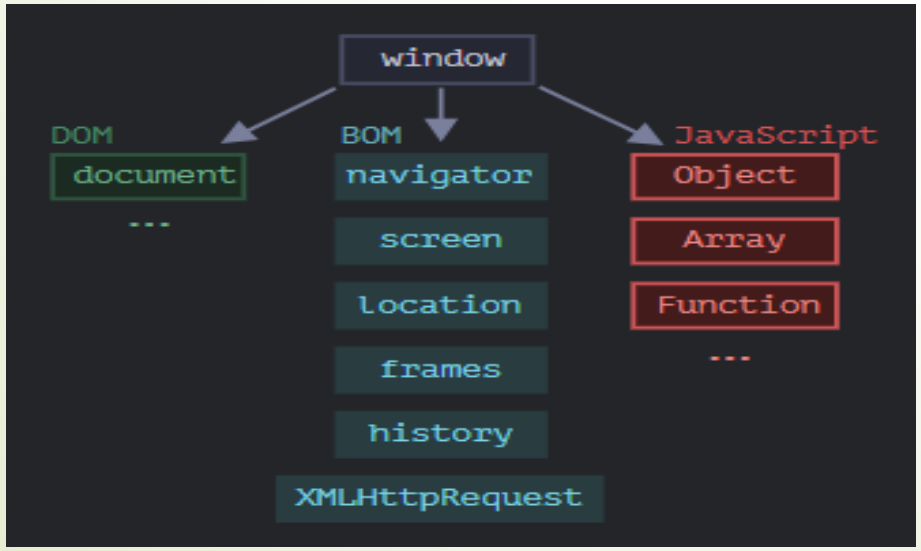
Entorno del navegador, especificaciones

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:

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



Hay un objeto “raíz” llamado window. Tiene dos roles:

- Primero, es un objeto global para el código JavaScript, como se describe en el capítulo Objeto Global.
- Segundo, representa la “ventana del navegador” y proporciona métodos para controlarla.

Por ejemplo, podemos usarlo como objeto global:

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

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

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 (https://dom.spec.whatwg.org/).**

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 URL
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.

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/.
  - 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.

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, pero existen muchas otras propiedades, tales como:

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

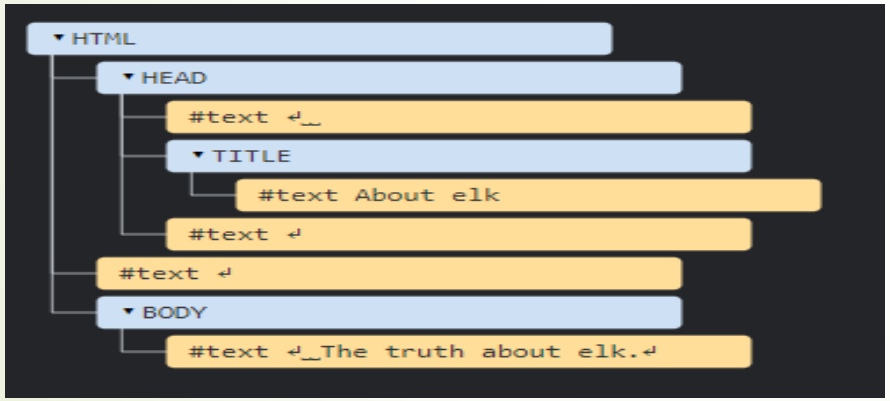
Más adelante, aprenderemos otras formas de manipular el DOM, pero primero necesitamos conocer su estructura.

Un ejemplo del DOM

Comencemos con un documento simple:

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

El DOM representa el HTML como una estructura de árbol de etiquetas. A continuación podemos ver cómo 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 simplemente “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, y son etiquetados como #text. Un nodo de texto puede contener únicamente una cadena y no puede tener hijos, siempre es una hoja del árbol.

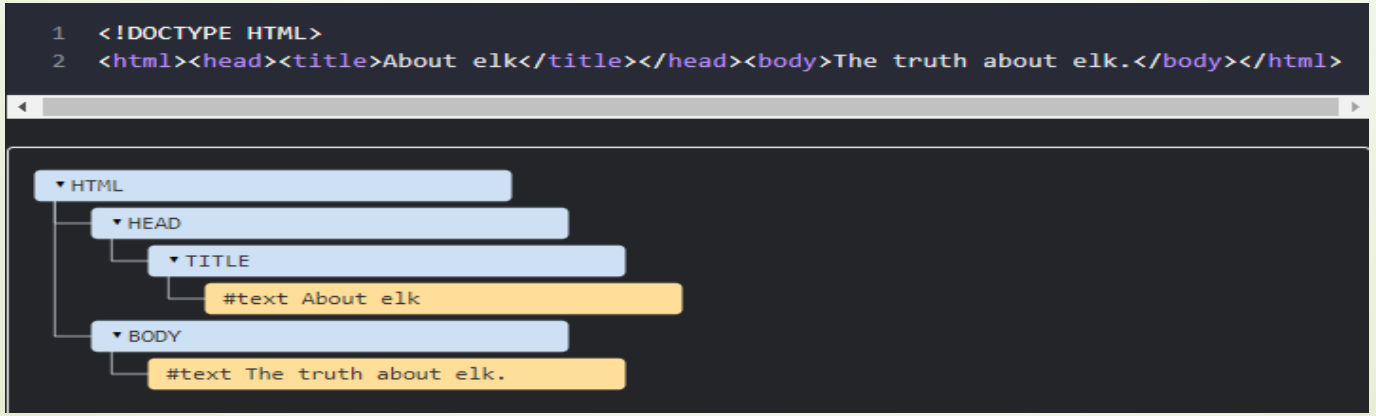
Por ejemplo, la etiqueta <title> tiene el texto "About elk". Hay que tener en cuenta los caracteres especiales en nodos de texto:

- una línea nueva: ↵ (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 ubicados antes de la etiqueta <head> son ignorados por razones históricas.
2. Si colocamos algo después de la etiqueta </body>, automáticamente se situará dentro de body, en el final, ya que la especificación HTML necesita que todo el contenido esté dentro de la etiqueta <body>. No puede haber espacios después de esta.

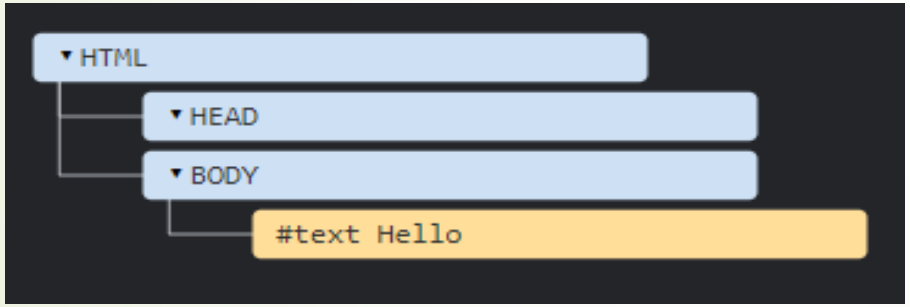
En otros casos todo es sencillo: si hay espacios (como cualquier carácter) en el documento, se convierten en nodos de texto en el DOM; y si los eliminamos, entonces no habrá nodo. En el siguiente ejemplo, no hay nodos de texto con espacios en blanco:



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>.

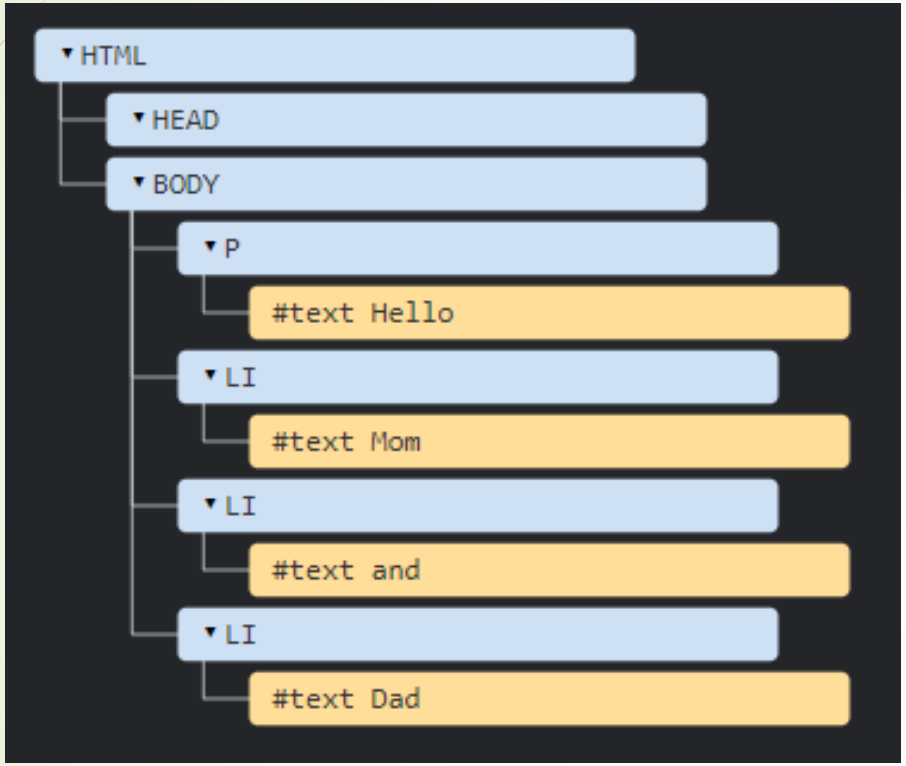
Como ejemplo de esto, si el archivo HTML es la palabra "Hello", el navegador lo envolverá con las etiquetas <html> y <body>, y añadirá la etiqueta <head> la cual es requerida. Basado en esto, el DOM resultante será:



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:



NOTA: Las tablas siempre tienen la etiqueta <tbody>. 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.

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: 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.

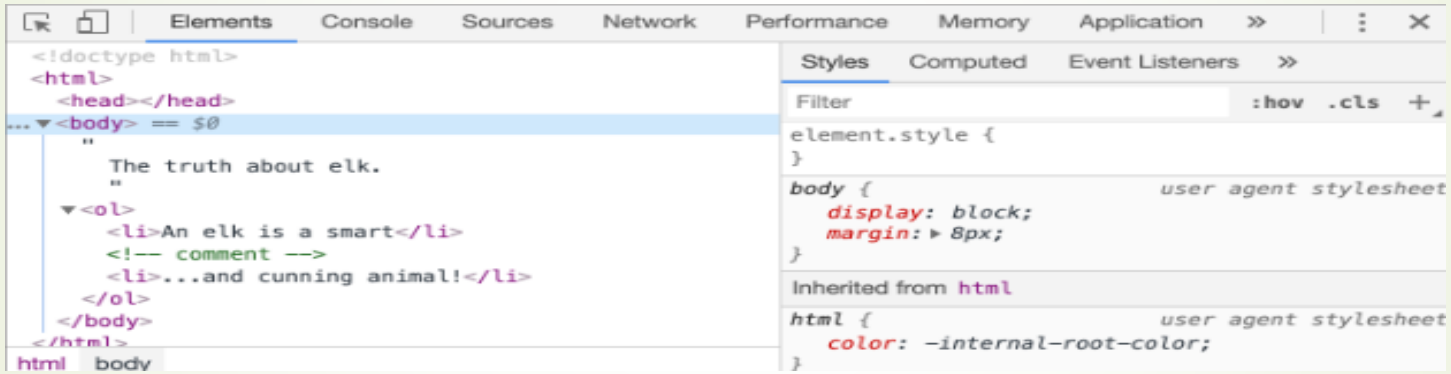
El objeto document que representa todo el documento es también, formalmente, un nodo DOM. Hay 12 tipos de nodos. En la práctica generalmente trabajamos con 4 de ellos:

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 – Podríamos colocar información allí. No se mostrará, pero JS puede leerla desde el DOM.

Revisalo tu mismo

Para ver la estructura del DOM en tiempo real, intente **Live DOM Viewer (https://software.hixie.ch/utilities/js/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 desarroladores del navegador y cambie la pestaña a elementos. Debe verse así:

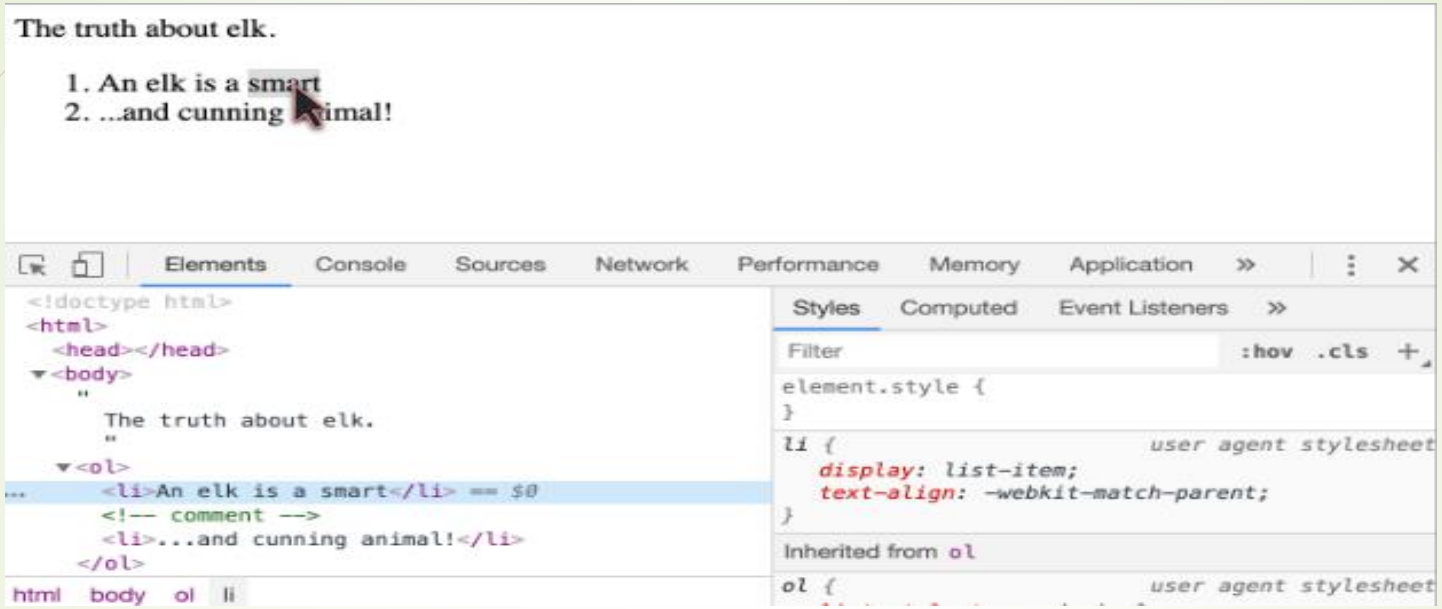


Puedes ver el DOM, hacer clic sobre los elementos, ver sus detalles, etc.

Tenga en cuenta que la estructura DOM en la herramienta para desarrolladores está simplificada. Los nodos de texto se muestran como texto. Y no hay nodos de texto con espacios en blanco en absoluto. Esto es aceptable, 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 “escuchadores de eventos” adosados a elementos del DOM (los cubriremos en la siguiente parte del tutorial).

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

Interacción con la consola

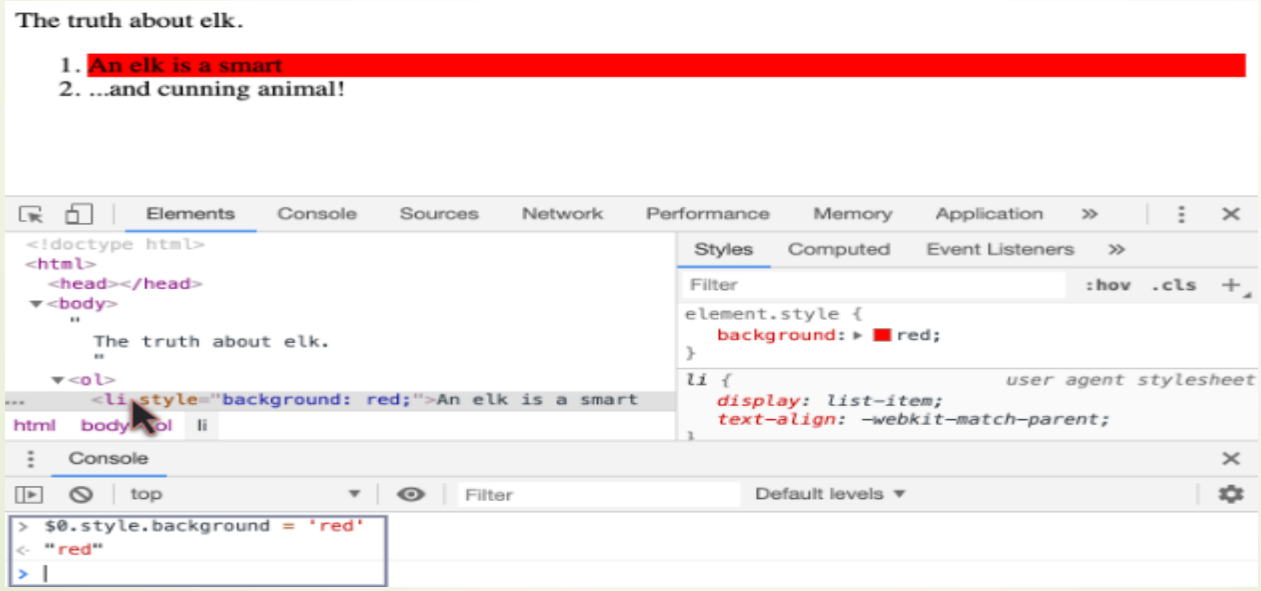
A medida que trabajamos con el DOM, también podemos querer aplicarle JavaScript. Por ejemplo, obtener un nodo y ejecutar algún código para modificarlo y ver el resultado. Aquí hay algunos consejos para desplazarse entre la pestaña elementos y la consola.

Para empezar:

1. Seleccione el primer elemento <li> en la pestaña elementos.
2. Presiona 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.

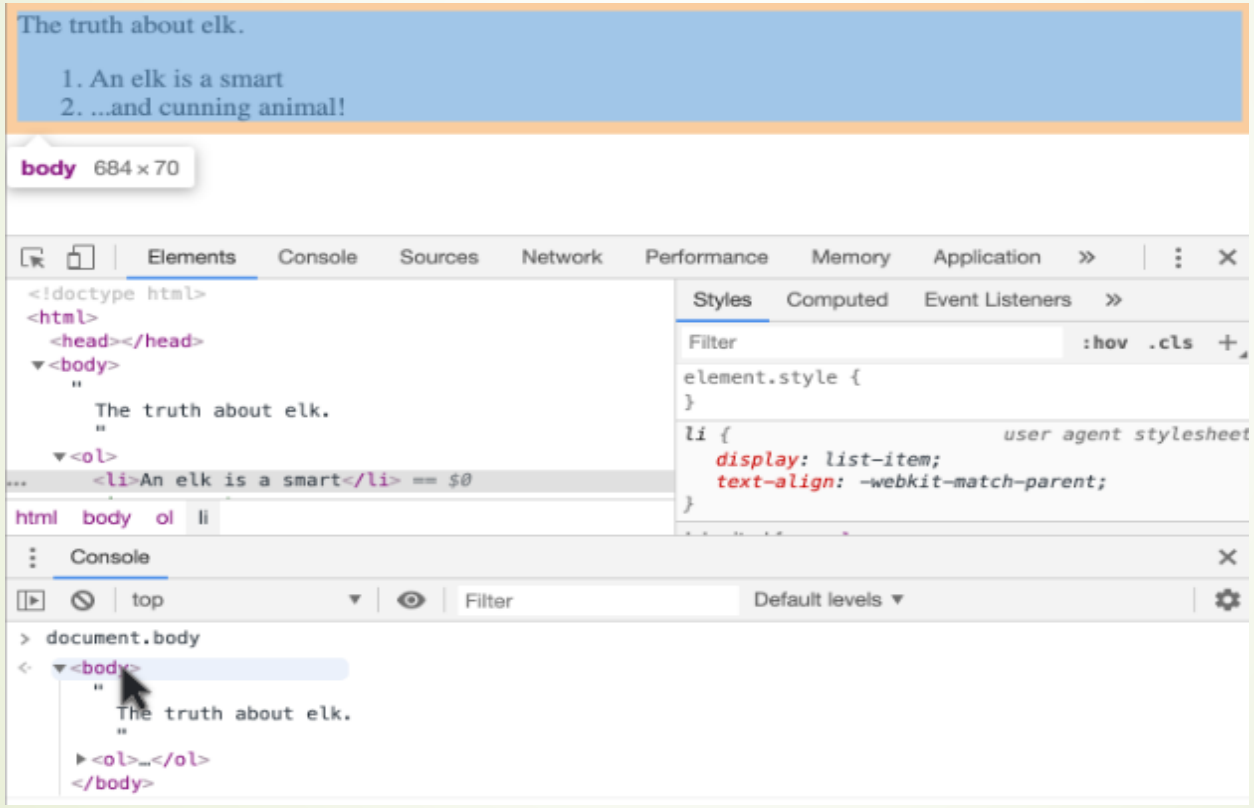
Podemos ejecutar comandos en ellos. Por ejemplo, ***$0.style.background = 'red'*** hace que el elemento de la lista seleccionado sea rojo, algo así:



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

También está el camino inverso. 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 imprimir el nodo del DOM en la consola y explorarlo en el lugar, tal como document.body a continuación:



Desde luego, eso es para propósitos de depuración. Las herramientas para desarrolladores del navegador son de mucha ayuda en el desarrollo: podemos explorar el DOM, probar cosas y ver qué sale mal.

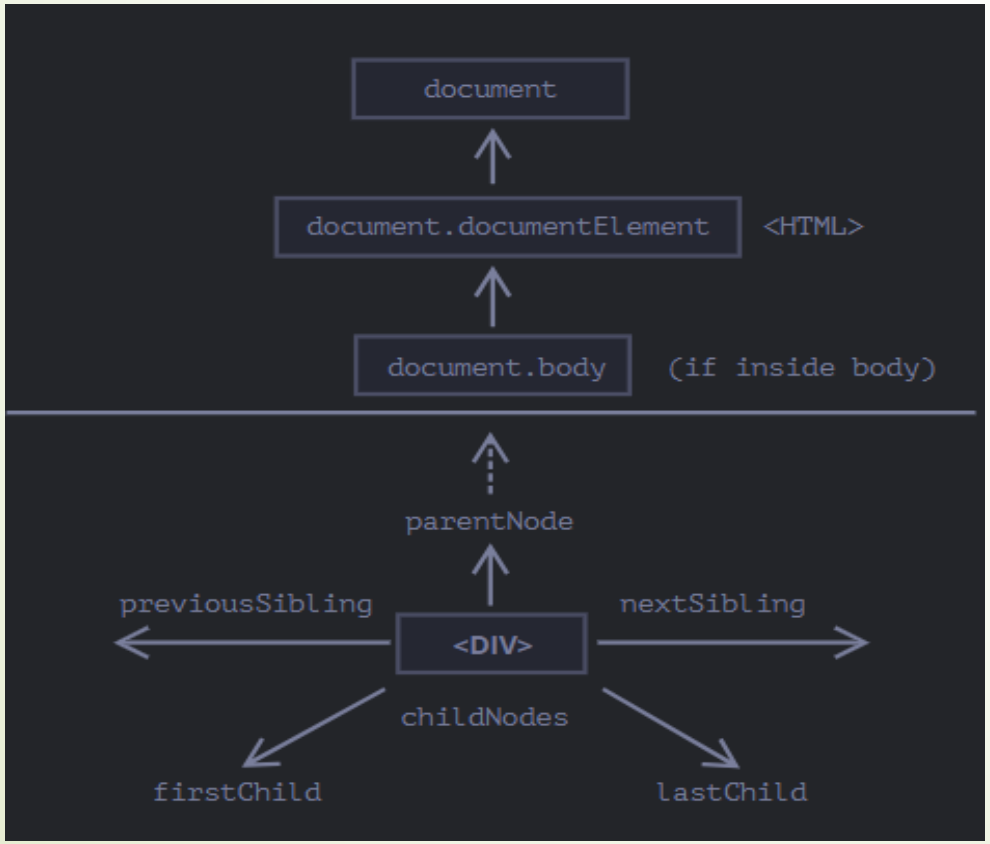
RESUMEN

- Un documento HTML/XML esta 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.
  - 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 hemos cubierto los conceptos básicos, las acciones más importantes y 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 la documentación y elija el resto.
- Los nodos del DOM tienen propiedades y métodos que nos permiten desplazarnos entre ellos, modificarlos, movernos por la página, y más.

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:



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.

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

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 y <ul> (y unos pocos nodos de texto en blanco):

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>
      <b>Information</b>
    </li>
  </ul>
</body>
</html>
```

...Y los descendientes de <body> no son solo los hijos , <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:

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Texto, DIV, Texto, UL, ..., SCRIPT
    }
  </script>
  ...más cosas...
</body>
</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:

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

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:

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

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:

```
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:

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

Hermanos y el padre

Los hermanos son nodos que son hijos del mismo padre. Por ejemplo, aquí <head> y <body> son hermanos:

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

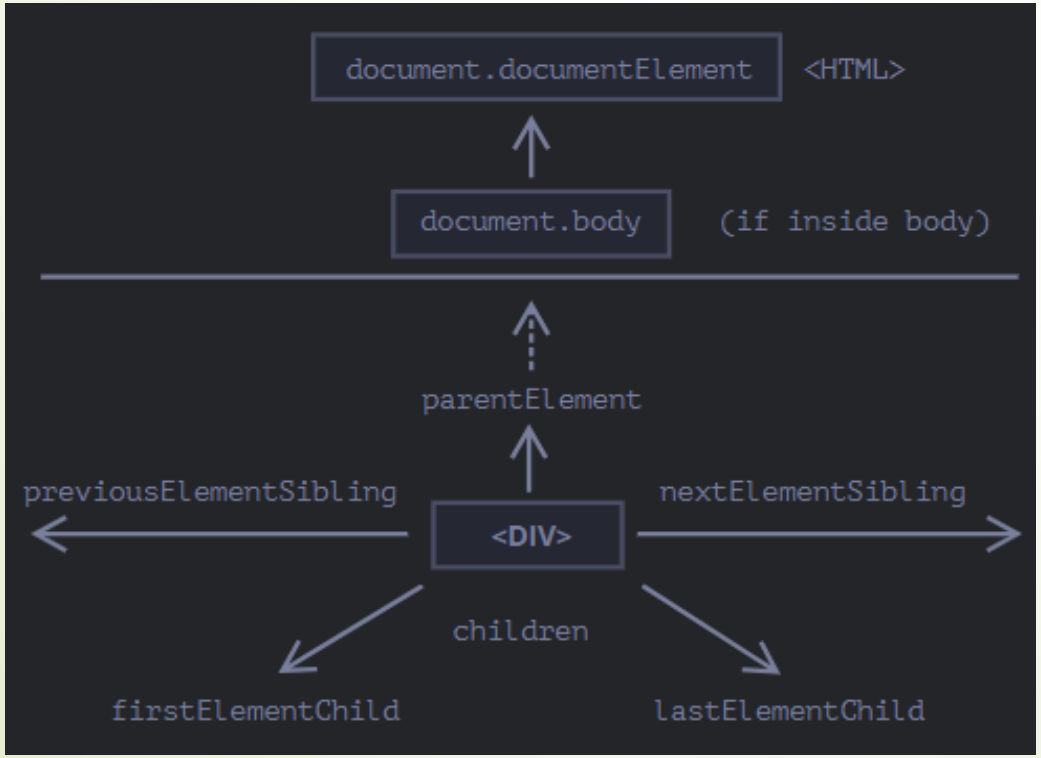
El hermano siguiente está en la propiedad nextSibling y el anterior – en previousSibling. El padre está disponible en parentNode. Por ejemplo:

```
// el padre de <body> es <html>
alert( document.body.parentNode === document.documentElement ); // verdadero

// después de <head> va <body>
alert( document.head.nextSibling ); // HTMLBodyElement

// antes de <body> va <head>
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.

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

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>
</html>
```

Más enlaces: tablas

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:

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

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

La especificación: **tabular data (https://html.spec.whatwg.org/multipage/tables.html)**.

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

Mira esta página:

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

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

¿El nodo del DOM?

¿El nodo <ul> del DOM?

El segundo <li> (con Pete)?

Seleccionar todas las celdas diagonales

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

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

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

El resultado debe ser:

Abrir un entorno controlado para la tarea.

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:

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // obtener el elemento
  let elem = document.getElementById('elem');

  // hacer que su fondo sea rojo
  elem.style.background = 'red';
</script>
```

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

```
<div id="elem">
  <div id="elem-content">Elemento</div>
</div>

<script>
  // elem es una referencia al elemento del DOM con id="elem"
  elem.style.background = 'red';

  // id="elem-content" tiene un guion en su interior, por lo que no puede ser un nombr
  // ...pero podemos acceder a él usando corchetes: window['elem-content']
</script>
```

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

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

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

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

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

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:

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

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

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

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 (https://dom.spec.whatwg.org/#dom-element-matches)** 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:

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

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

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:

```
<h1>Contenido</h1>

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

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

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

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

getElementsBy*

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

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:

```
<table id="table">
  <tr>
    <td>Su edad:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> menos de 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> de 18 a 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> más de 60
      </label>
    </td>
  </tr>
</table>

<script>
  let inputs = table.getElementsByTagName('input');

  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

Buscando elementos .article:

```
<form name="my-form">
  <div class="article">Artículo</div>
  <div class="long article">Artículo largo</div>
</form>

<script>
  // encontrar por atributo de nombre
  let form = document.getElementsByName('my-form')[0];

  // encontrar por clase dentro del formulario
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, encontró dos elementos con la clase "article"
</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.

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

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

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

<div>Segundo div</div>

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

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:

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

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

<div>Segundo div</div>

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

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?
`querySelector`	selector CSS	✓	-
`querySelectorAll`	selector CSS	✓	-
`getElementById`	`id`	-	-
`getElementsByName`	`name`	-	✓
`getElementsByTagName`	etiqueta o `'*'`	✓	✓
`getElementsByClassName`	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.

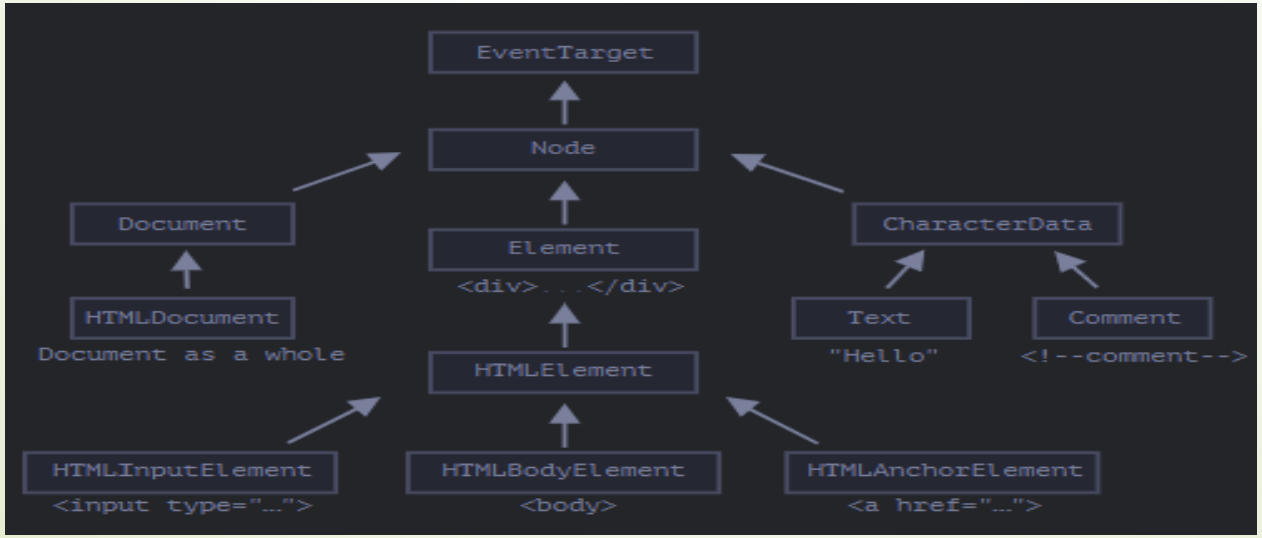
Propiedades del nodo: tipo, etiqueta y contenido

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 Hello .
    - 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>,

- Hay muchas otras etiquetas con sus propias clases que pueden tener propiedades y métodos específicos, mientras que algunos elementos, tales como , <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:

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

…O podemos simplemente usar toString:

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

También podemos usar instanceof para verificar la herencia:

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

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.

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:

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

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

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

  // para el objeto de tipo documento, el tipo es 9
  alert( document.nodeType ); // 9
  </script>
</body>
```

En los scripts modernos, podemos usar instanceof 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:

```
alert( document.body.nodeName ); // BODY
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:

```
<body><!-- comentario -->

  <script>
    // para comentarios
    alert( document.body.firstChild.tagName ); // undefined (no es un elemento)
    alert( document.body.firstChild.nodeName ); // #comment

    // para documentos
    alert( document.tagName ); // undefined (no es un elemento)
    alert( document.nodeName ); // #document
  </script>
</body>
```

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

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:

```
<body>
  <p>Un párrafo</p>
  <div>Un div</div>

  <script>
    alert( document.body.innerHTML ); // leer el contenido actual
    document.body.innerHTML = 'El nuevo BODY!'; // reemplazar
  </script>

</body>
```

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

```
<body>

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

</body>
```

NOTA: “innerHTML+=” hace una sobrescritura completa

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

```
chatDiv.innerHTML += "<div>Hola<img src='smile.gif'/> !</div>";
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:

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. 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:

```
<div id="elem">Hola <b>Mundo</b></div>

<script>
  alert(elem.outerHTML); // <div id="elem">Hola <b>Mundo</b></div>
</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:

```
<div>¡Hola, mundo!</div>

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

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

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

En la línea (*) reemplazamos div con Un nuevo elemento . En el documento externo (el DOM) podemos ver el nuevo contenido en lugar del . 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 Un nuevo elemento se insertó en su lugar.
  - div todavía tiene su antiguo valor. El nuevo HTML no se guardó en ninguna variable.
- Es muy fácil cometer un error aquí: modificar ***div.outerHTML*** y luego continuar trabajando con div como si tuviera el nuevo contenido. Pero no es así. Esto es correcto para innerHTML, pero no para outerHTML.
- 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:

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

    let comment = text.nextSibling;
    alert(comment.data); // Comentario
  </script>
</body>
```

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

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

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

textContent: texto puro

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

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

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

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

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:

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

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

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

1. El primer obtiene el nombre "como HTML": todas las etiquetas se convierten en etiquetas, por lo que vemos el nombre en negrita.
2. El segundo obtiene el nombre "como texto", así que literalmente vemos ¡Winnie-Pooh!.

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:

```
<div>Ambos divs a continuación están ocultos</div>

<div hidden>Con el atributo "hidden"</div>

<div id="elem">JavaScript asignó la propiedad "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Técnicamente, hidden funciona igual que style="display:none". Pero es más corto de escribir. Aquí hay un elemento parpadeante:

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

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

Por ejemplo:

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

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // value
</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.