Contenido

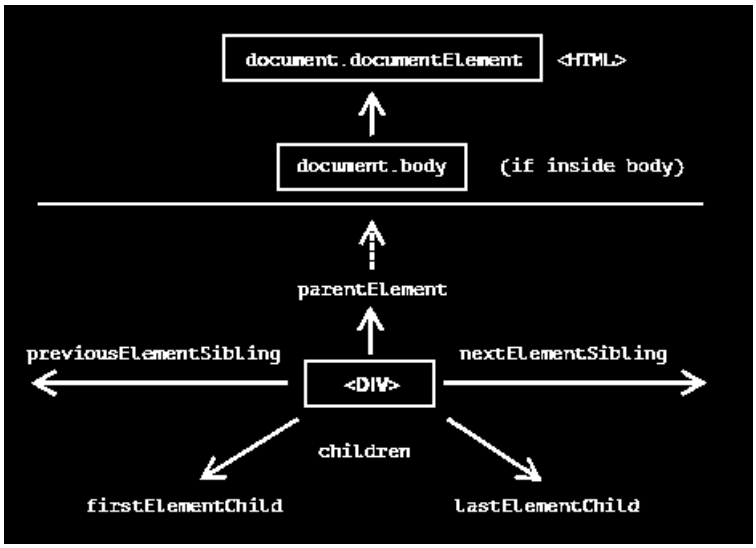
1 NAVEGAR A TRAVÉS DE ELEMENTOS 1
1.1.1 Navegar a través de nodos 3

NAVEGAR POR EL DOM

En algunas ocasiones en las que conocemos y controlamos perfectamente la estructura del código HTML de la página, nos puede resultar más cómodo tener a nuestra disposición una serie de propiedades para **navegar por la jerarquía** de elementos HTML relacionados.
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.

1 NAVEGAR A TRAVÉS DE ELEMENTOS

Las propiedades de navegación enumeradas en el próximo apartado (punto 2), 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 los enlaces de navegación que solo tienen en cuenta los elementos:



Las propiedades que veremos a continuación devuelven información de otros elementos relacionados con el elemento en cuestión.

Propiedades de elementos HTML	Descripción
ARRAY children	Devuelve una lista de elementos HTML hijos.
ELEMENT parentElement	Devuelve el padre del elemento o NULL si no tiene.
ELEMENT firstElementChild	Devuelve el primer elemento hijo.
ELEMENT lastElementChild	Devuelve el último elemento hijo.
ELEMENT previousElementSibling	Devuelve el elemento hermano anterior o NULL si no tiene.
ELEMENT nextElementSibling	Devuelve el elemento hermano siguiente o NULL si no tiene.

En primer lugar, tenemos la propiedad **children** que nos ofrece un ARRAY con una lista de elementos HTML hijos. Podríamos acceder a cualquier hijo utilizando los corchetes de array y seguir utilizando otras propiedades en el hijo seleccionado.

- La propiedad **firstElementChild** sería un acceso rápido a **children[0]**
- La propiedad **lastElementChild** sería un acceso rápido al último elemento hijo.

Por último, tenemos las propiedades **previousElementSibling** y **nextElementSibling** que nos devuelven los elementos hermanos anteriores o posteriores, respectivamente. La propiedad **parentElement** nos devolvería el padre del elemento en cuestión. En el caso de no existir alguno de estos elementos, nos devolvería NULL.

Consideremos el siguiente documento HTML:

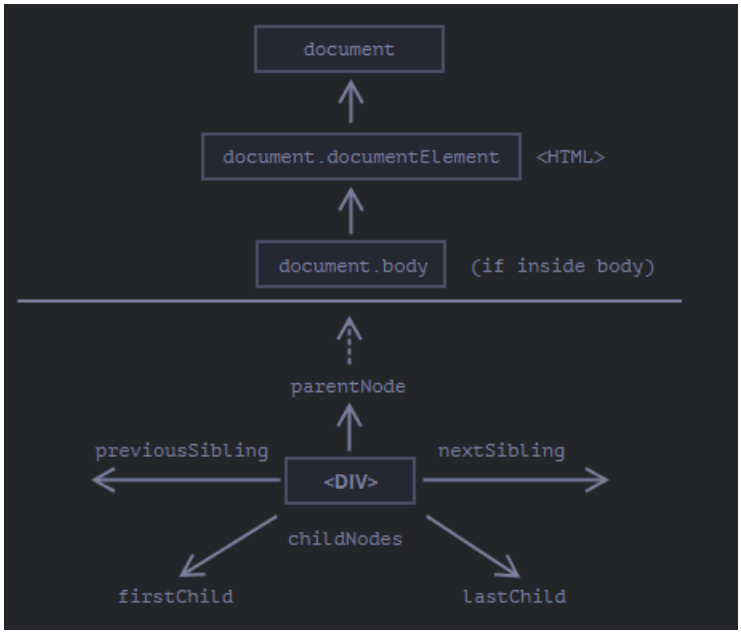
```
<html>
 <body>
  <div id="app">
   <div class="header">
    <h1>Titular</h1>
   </div>
   <p>Párrafo de descripción</p>
   <a href="/">Enlace</a>
  </div>
 </body>
</html>
```

Si trabajamos bajo este documento HTML, y utilizamos el siguiente código Javascript, podremos «navegar» por la jerarquía de elementos, **moviéndonos entre elementos** padre, hijo o hermanos:

```
document.body.children.length;  // 1
document.body.children;       // <div id="app">
document.body.parentElement;   // <html>

const app = document.querySelector("#app");

app.children;               // [div.header, p, a]
app.firstElementChild;      // <div class="header">
app.lastElementChild;       // <a href="/">

const a = app.querySelector("a");

a.previousElementSibling;     // <p>
a.nextElementSibling;        // null
```

Estas son las propiedades más habituales para navegar entre elementos HTML, sin embargo, tenemos otra modalidad un poco más detallada.

2 Navegar a través de nodos



La primera tabla que hemos visto nos muestra una serie de propiedades cuando trabajamos con ELEMENT . Sin embargo, si queremos hilar más fino y trabajar a nivel de NODE, podemos utilizar las siguientes propiedades, que son equivalentes a las anteriores:

Propiedades de nodos HTML	Descripción
ARRAY childNodes	Devuelve una lista de nodos hijos. Incluye nodos de texto y comentarios.
NODE parentNode	Devuelve el nodo padre del nodo o NULL si no tiene.
NODE firstChild	Devuelve el primer nodo hijo.
NODE lastChild	Devuelve el último nodo hijo.
NODE previousSibling	Devuelve el nodo hermano anterior o NULL si no tiene.
NODE nextSibling	Devuelve el nodo hermano siguiente o NULL si no tiene.

Estas propiedades suelen ser más interesantes cuando queremos trabajar sobre nodos de texto, ya que incluso los espacios en blanco entre elementos HTML influyen. Volvamos a trabajar sobre el documento HTML anterior, pero ahora utilizando este grupo de propiedades basadas en **NODE**:

```
document.body.childNodes.length;  // 8
document.body.childNodes;       // elementos html + contenido
document.body.parentNode;       // <html>

const app = document.querySelector("#app");

app.childNodes;               // [div.header, h1, text, p, text, a, text]
app.firstChild.textContent;     // "                    "
app.lastChild.textContent;      // "                    "

const a = app.querySelector("a");

a.previousSibling;            // #text
a.nextSibling;              // #text
```

Con todo esto, ya tenemos suficientes herramientas para trabajar a bajo nivel con las etiquetas y nodos HTML de un documento HTML desde Javascript.

Colecciones del DOM

Como podemos ver, childNodes o children parece devolver 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
```

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 hemos visto anteriormente

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:

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

3 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:

```
<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.

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

4 TEMPORIZADORES

4.1 ¿QUÉ SON LOS TEMPORIZADORES?

El objeto window posee varios métodos relacionados con temporizadores. El manejo del tiempo es un elemento fundamental de la programación. Es fundamental en la programación de juegos, en la creación de animaciones, en publicidad y en otras muchas áreas.

Todas estas capacidades tienen que ver con dos métodos que permiten ejecutar un determinado código (normalmente por medio de una función callback) cuando pase cierto tiempo.

El hecho de que se ejecute un código cuando ocurra un tiempo concreto nos aproxima a la gestión de eventos que es la base del tema siguiente. No obstante, la buena noticia es que el manejo de temporizadores en JavaScript es francamente fácil.

4.2 SETTIMEOUT

El método setTimeout de window tiene dos parámetros: el primero es la función que se ejecutará cuando se cumpla el tiempo, el segundo es un valor numérico que indica el tiempo a cuplir en milisegundos. Ejemplo:

```
setTimeout ( ( )=>alert( "Hola"), 5000);
```

El primer parámetro del código anterior es la función flecha ()=>alert("Hola") esa función simplemente lanza el mensaje "Hola" en un cuadro de diálogo. El segundo parámetro indica que se esperará 5 segundos antes de ejecutar el código de la función.

El método setTimeout devuelve un número que identifica al temporizador en sí. Es decir, a cada temporizador lanzado con setTimeout se le asigna un número o identificador. Ese número se puede usar para cancelar el temporizador mediante **clearTimeout**.Ejemplo:

```
var temp1=setTimeout ( ()=>alert ( "Hola"),5000);
clearTimeOut (temp1) ;
```

Tal cual está escrito el código anterior, jamás aparecerá el mensaje Hola porque se cancela el temporizador inmediatamente (no habrán pasado los 5 segundos).

4.3 SETINTERVAL

El método **setlnterval** es muy similar al anterior. Tiene los mismos parámetros y devuelve también un identificador de temporizador que puede ser almacenado en una variable para poder cancelar el temporizador con un método llamado **clearlnterval**.

La diferencia es que, en este caso, el código asignado al temporizador se invoca cada vez que pase el tiempo indicado. Es decir, setTimeout invoca al código una sola vez y setlnterval lo invoca constantemente:

```
var temp2=setInterval (()=>alert("Hola"), 5000);
```

El código es casi idéntico al que vimos en el apartado anterior. Pero ahora el cuadro de mensaje diciendo Hola aparecerá cada 5 segundos y no solo una vez como antes. Jamás dejará de aparecer a no ser que lo cancelemos:

```
var cont=0;
var temp2=setInterval ( function () {
        alert("Hola ") ;
        cont++;
        if(cont) =10) clear Interval (temp2 );
        } ,5000);
```

Este código muestra la palabra Hola en un mensaje cada 5 segundos. Lo hará 10 veces y luego ya no lo hará, ya que cuando el contador valga diez, se habrá eliminado el termporizador mediante clearInterval.