TEMA 9

EL DOM

INTRODUCCIÓN

Las siglas **DOM** significan **Document Object Model**, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM.

En JavaScript, cuando nos referimos al DOM nos referimos a esta estructura, que podemos modificar de forma dinámica desde JavaScript, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc.

Al tener debajo un lenguaje de programación, todas estas tareas se pueden automatizar, incluso indicando que se realicen cuando el usuario haga acciones determinadas, como: pulsar un botón, mover el ratón, hacer click en una parte del documento, escribir un texto, etc…

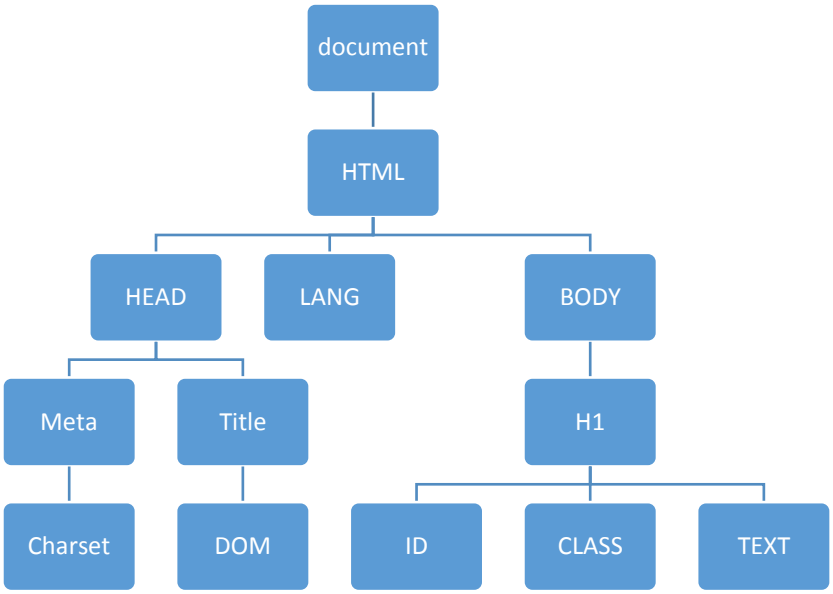
1. EL OBJETO DOCUMENT

La forma de acceder al DOM es a través de un objeto llamado document, que representa el árbol DOM de la página o pestaña del navegador donde nos encontramos. En su interior pueden existir varios tipos de elementos, pero principalmente serán ELEMENT o NODE.

Todos los elementos HTML, dependiendo del elemento que sea, tendrá un tipo de dato específico. Algunos ejemplos son: asociado a HTMLDivElement, asociado a HTMLSpanElement, etc. Todo son elementos que nosotros podemos añadir a través de etiquetas en una página web.

Debemos tener en cuenta que el DOM no es JavaScript, es una API (Aplication Programing Interface), que se puede modificar a través de JavaScript. Veamos cómo representar en el DOM una simple página web.

Como vemos esta página es muy simple, y solo nos muestra un título en la página web. Veamos cómo sería el DOM asociado.



Como vemos el número de nodos (cada uno de los elementos del esquema) es grande, y eso que la página es extremadamente sencilla. Cuando trabajemos con páginas más complicadas, podemos suponer que el número de nodos sería muy grande, y el gestionarlo podría llegar a ser muy complicado.

Los tipos de nodos que nos podemos encontrar en el DOM son tres:

1. Elemento nodo, asociado al valor 1, que sería cualquier etiqueta HTML
2. Text nodo, asociado al valor 3, que sería el contenido de la etiqueta
3. Comentario nodo, asociado al valor 8, que sería cualquier comentario en HTML.

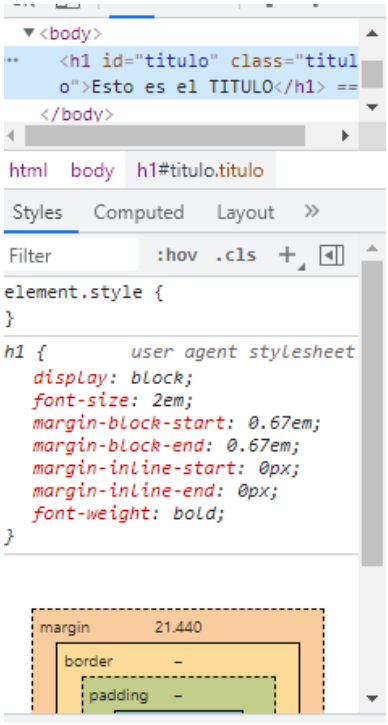
Cuando preguntamos sobre el tipo de nodo del DOM, el valor que se devuelve realmente es el número asociado que hemos incluido en el esquema.

2. UNIR HTML CON JAVASCRIPT

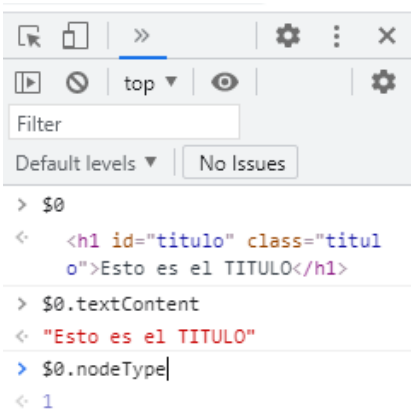
Hasta ahora, cuando queríamos añadir un fichero javascript con un fichero html, lo que hacíamos era añadir en la cabecera <head>, la instrucción <script src="fichero.js">. Es verdad, que cuando solo utilizamos funciones, la forma de hacerlo puede ser esta, pero cuando lo que vamos a hacer es manejar elementos del DOM, necesitamos que estos elementos estén creados para poder modificarlos.

En la actualidad, la forma de ejecución cuando insertamos ficheros javascript, no es exactamente secuencial como explicábamos al principio, así que el sitio donde nos encontraremos esta instrucción será al final del cuerpo, es decir, justo antes de la etiqueta </body>.

Desde la consola del navegador, podemos seleccionar el elemento sobre el que nos pueda interesar recabar información. Y desde la consola con el comando $0 podemos solicitar información del elemento correspondiente, que se corresponderá con un nodo en el DOM.



Luego desde la consola, podemos ir solicitando el valor de los atributos que nos interesa a nosotros. En la imagen posterior podemos ver alguna de las opciones. Como vemos el tipo de nodo es 1, ya que h1 es un elemento de HTML.



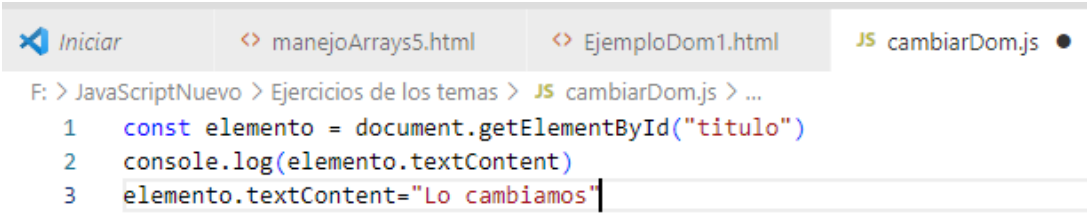
3. SELECCIONAR ELEMENTOS DEL DOM

Si nos encontramos en nuestro código JavaScript y queremos hacer modificaciones en un elemento de la página HTML, lo primero que debemos hacer es buscar dicho elemento. Para ello, se suele intentar identificar el elemento a través de alguno de sus atributos más utilizados, generalmente el **id** o la **clase**.

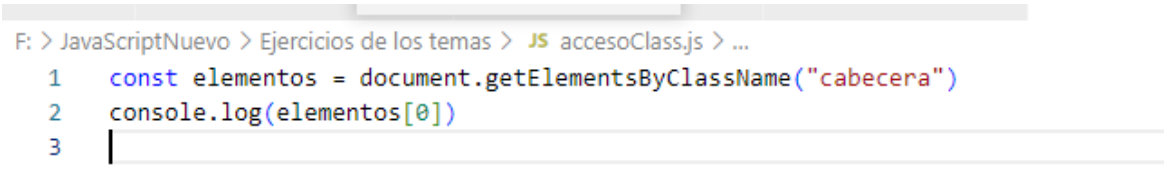
Existen varios métodos, los más clásicos y tradicionales para realizar búsquedas de elementos en el documento son:

- **getElementById(id):** este método busca un elemento HTML con el id especificado en el id por parámetro. En principio, un documento HTML bien construido no debería tener más de un elemento id, por tanto, este método devolverá solo un elemento. En caso de no encontrar el elemento indicado devolverá NULL.

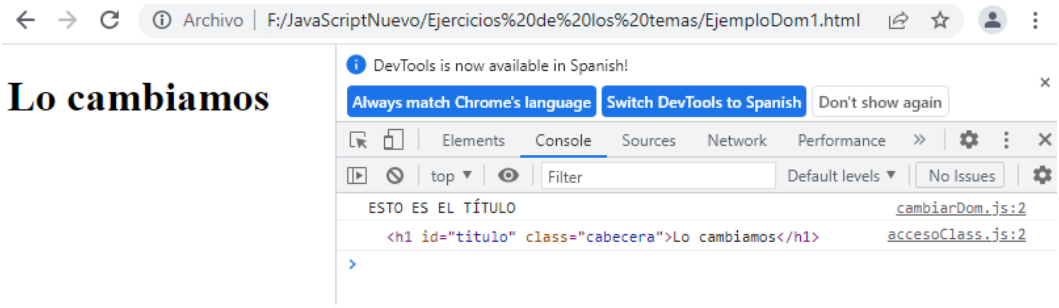
Veamos un ejemplo de cómo usar este método, para ello incluiremos un fichero JavaScript donde localizaremos un elemento y luego modificaremos uno de sus atributos.



- **getElementsByClassName():** Este método permite buscar los elementos con la clase especificada en class. Es importante darse cuenta del matiz de que el método tienen getElements en plural, y esto es porque al devolver clases se pueden repetir, y por tanto, devolvernos varios elementos, no solo uno.



Este es el resultado obtenido:



Existen varios métodos más modernos que podemos utilizar para el manejo de elementos del DOM:

- **querySelector(selector)**: devuelve el primer elemento que encuentra que encaja con el selector CSS suministrado en selector. Igual que su equivalente getElementById(), devuelve un solo elemento y en caso de no coincidir con ninguno devuelve NULL.
  Veamos un ejemplo de uso:

  **const page = document.querySelector("#page"); // **

  **const info = document.querySelector(".main .info"); // **

  Lo interesante de este método, es que al permitir suministrarle un selector CSS básico o incluso un selector CSS avanzado, se vuelve un sistema mucho más potente.

  El primer ejemplo es equivalente a utilizar un **.getElementById()**, solo que en la versión **.querySelector()** indicamos por parámetro un **selector**, y en el primero le pasamos un simple **String**. Observa que estamos indicando un # porque se trata de un id.

  En el segundo ejemplo, estamos recuperando el primer elemento con clase **info** que se encuentre dentro de otro elemento con clase **main**. Eso podría realizarse con los métodos tradicionales, pero sería menos directo ya que tendríamos que realizar varias llamadas con **.querySelector()** se hace directamente con solo una.

Como hemos dicho este método nos encontraría el primer elemento de la página asociado a la información pasada como argumento. Si quisiéramos acceder al que ocupa una posición determinada podemos utilizar el condicionador **nth-child**, al que le pasaremos qué sitio ocupa el elemento que queremos recuperar. Veamos un ejemplo:

const page = document.querySelector(".parrafo:nth-child(2)");

- **querySelectorAll(selector)**: Este método permite acceder a todos los elementos que coinciden con el selector CSS, devuelve un nodeList, que podemos manejar de forma similar a un array, pero debemos tener cuidado porque no es un array, por lo que no se pueden utilizar los métodos que vimos asociados a los arrays. Si no encuentra ningún elemento de ese tipo lo que hace es devolver una lista vacía.
  Si queremos acceder a uno de los elementos específicamente, bastará con acceder a él a través del índice asociado, como en los arrays empezaremos a contar en el índice cero.

  page[2].style.color =”yelow”

4. MODIFICAR ATRIBUTOS EN EL DOM

Vamos a trabajar ahora sobre una página donde nos aparece un título, asociado a un id, una etiqueta y un cuadro de texto.

C: > Ejercicios javascript code > atributosDOM.html > html > body > input#nombre

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=<device-width>, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <h1 id="titulo">Cambiando los atributos del DOM</h1>
11     <label for="name">Nombre</label>
12     <input type="text" id="nombre">
13 </body>
14 </html>
```

4.1 Atributos del DOM

Vamos a ver dos métodos que podemos utilizar para trabajar con los atributos en el DOM, getAttribute que nos devuelve el atributo, y setAttribute que lo que hace es asignar un valor al atributo. Veremos a continuación un ejemplo.

La sintaxis de getAttribute es la siguiente: **element.getAttribute(“atributo”)**. A este método le pasamos el nombre del atributo del que queremos recuperar la información.

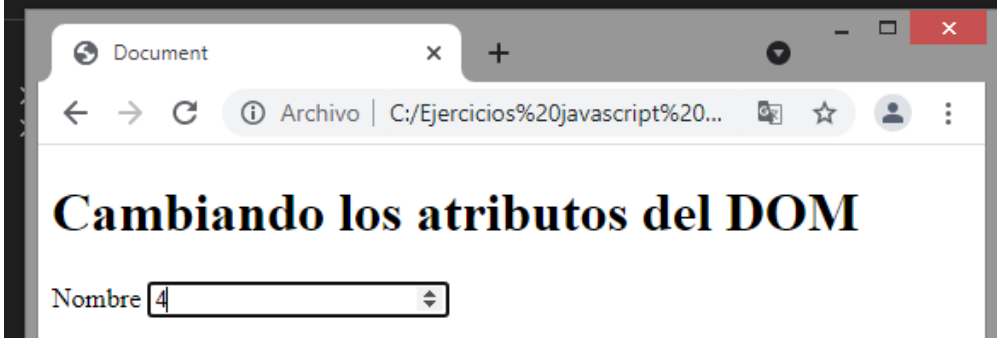
```
1  const title=document.getElementById("titulo");
2  const nombre=document.getElementById("nombre");
3
4  console.log(title);
5  console.log(nombre);
6
7  console.log(nombre.getAttribute("type"));
```

```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> h1#titulo
> input#nombre
  text
```

Como vemos, en este caso lo que hemos incluido en getAttribute es type, con lo que nos muestra es el tipo del elemento asociado al nombre, que es de tipo text.

En el caso de setAttribute, la sintaxis es la siguiente: element.setAttribute("atribute",valor). Como vemos en este caso pasamos dos parámetros, ya que hay que especificar a qué atributo queremos hacerle la modificación, y qué valor le queremos dar. Veamos un ejemplo:

```
1  const title=document.getElementById("titulo");
2  const nombre=document.getElementById("nombre");
3
4  console.log(title);
5  console.log(nombre);
6
7  console.log(nombre.setAttribute("type","number"));
```



Como vemos, algo que debería ser un texto, se convierte en un cuadro de texto donde solo se pueden incluir números. En el segundo valor podríamos poner cualquiera de los otros elementos que podemos encontrar en una página html: date, radio, etc.

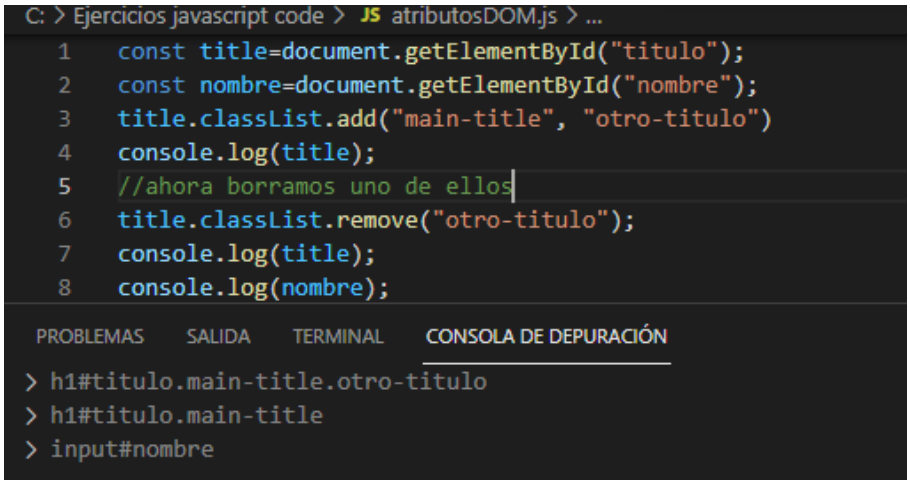
4.2 Clases del DOM

Vamos a ver varios métodos asociados a classList que nos va a permitir añadir clases, las clases irán entre comillas y separadas por comas, para ello usaremos el método **add**. Podemos borrar clases con el método **remove**, podemos eliminar tantas clases como queramos. Podemos también utilizar el método **togglet** que lo que hace es si una clase no existe se añade, y si existe se elimina. Esto se utiliza mucho en los menús de las páginas web, como veremos. Hay otro método, el **contains** que devolverá true o false, en función de si la clase que pasamos como argumento está o no lo está. Y el método **replace** que lo que hace es sustituir una clase por otra,

con lo que necesitaremos pasar dos argumentos. Vamos a trabajar con estos métodos en los siguientes ejemplos.

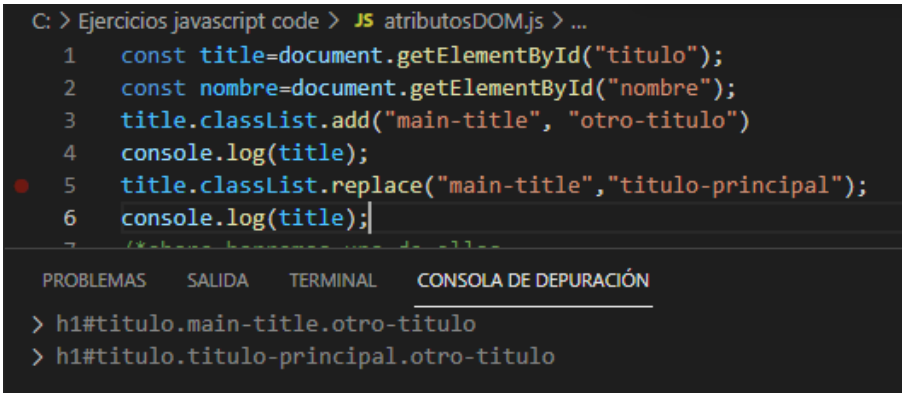
En este caso, como podemos ver en la consola hemos añadido los atributos main-title y otro-titulo a la cabecera h1.

Veamos ahora como eliminar una de las clases que hemos añadido, simplemente utilizando el método remove.

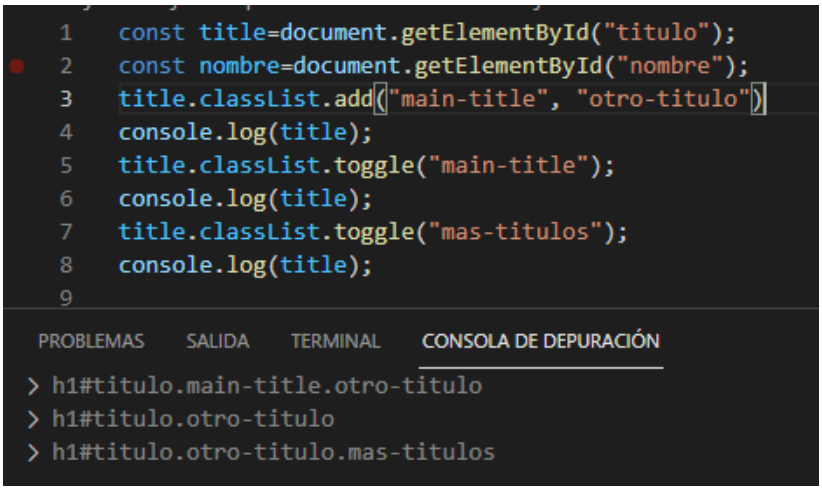


Veamos ahora, como contains, lo que hace es comprobar si una clase existe, devolviendo true si es así, y false si la clase no existe.

Y para el caso de sustituir una clase por otra, usaremos replace de la siguiente forma:



Veamos ahora como toggle, añade si no existe y elimina si existe.



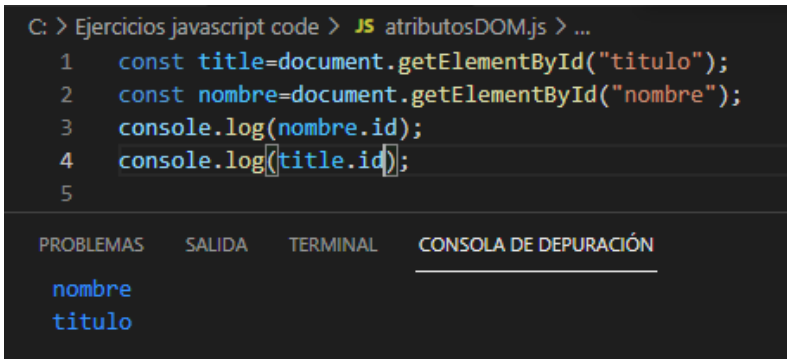
4.3 Modificar los atributos o propiedades de un objeto

Para saber cuáles son estos elementos, bastará con desplegar en la consola un objeto, y observar cuáles de los elementos no se pueden desplegar, estos se podrían modificar o mostrar directamente. Veámoslo con dos de los atributos más utilizados: id, value, innerHTML, textContent.

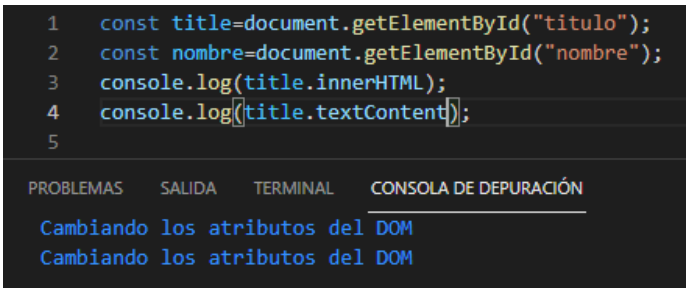
Como podemos ver en el ejemplo anterior, al modificar el value asociado al cuadro de texto, lo que sucede es que el String elegido para incluir en el atributo, aparece en el cuadro de texto. Este

atributo es el que necesitaremos comprobar, cuando queramos validar los elementos de un formulario, como veremos posteriormente.

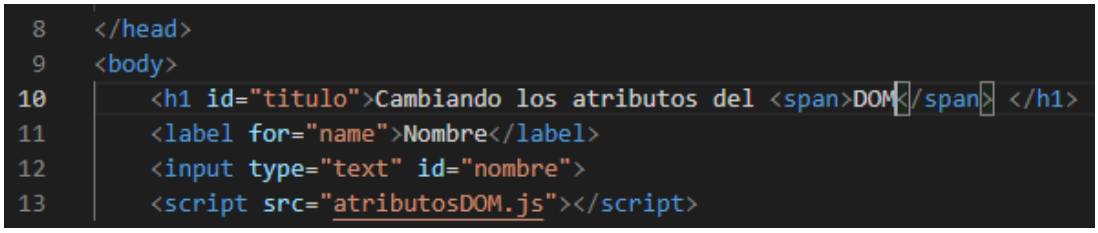
En el caso del atributo id, lo que nos muestra cuál es el identificador asociado al elemento que nosotros solicitamos.



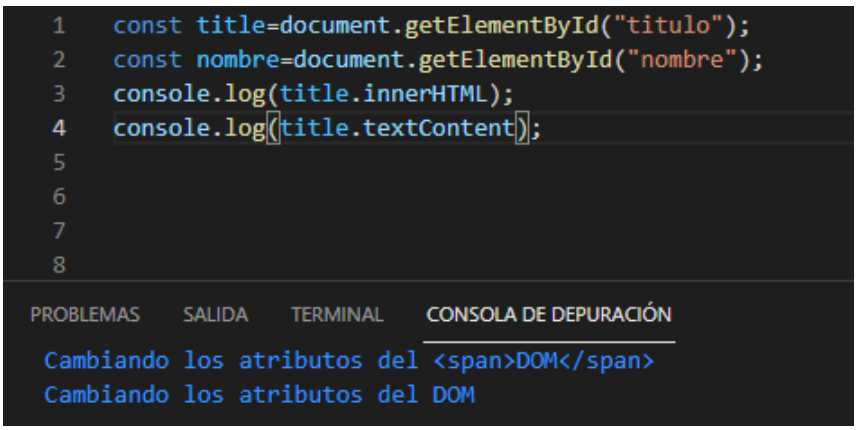
Las propiedades innerHTML, aunque en ocasiones puede parecer que devuelven lo mismo no es así. El textContent lo que devuelve es el texto incluido en un elemento; mientras que innerHTML lo que devuelve son las etiquetas incluidas en un elemento. Cuando no existen etiquetas, el resultado sí que es el mismo. Veámoslo:



Sin embargo, si incluimos una etiqueta en el encabezado h1, veamos cuál es el resultado.



El resultado aplicando las mismas sentencias que en el caso anterior, como podemos ver no es el mismo.



Hay muchos más atributos que podemos cambiar, que irán apareciendo a lo largo del curso, y que podemos estudiar en HTML.

EVENTOS

Un evento es cualquier cosa que sucede en nuestro documento. Existen muchos tipos de eventos, y no siempre los ejecuta el usuario, aunque nos parezca eso. Por ejemplo, cuando se carga una página se ejecuta un evento, cuando se lee un contenido, que la ventana se cierre también es un evento; y como vemos en estos no interviene directamente el usuario. Además, existen eventos que sí que son directamente aplicados por los usuarios, como mover un ratón, pulsar una tecla, etc.

Como hemos visto los eventos se pueden controlar directamente en el elemento html, incluyendo en su etiqueta qué evento producirá una respuesta, asociada a una función. Pero la forma más recomendable en la actualidad es como ya explicamos añadiendo al elemento el método addEventListener, pero esto ya se hace directamente en javascript, lo que hace que el código sea más claro y más fácil de modificar.

La sintaxis de la función **addEventListener** es el siguiente:

element.addEventListener("event",callback);

donde callback, sería la función que queremos que se ejecute cuando se produce el evento. Veamos alguno ejemplo de los eventos que más se utilizan.

En primer lugar, tendremos una página con un solo botón que podemos pulsar, lo que vamos a hacer es utilizar el addEventListener para que cuando pulsemos el botón se ejecute una función.

C: > Ejercicios javascript code > eventos.html > html

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta http-equiv="X-UA-Compatible" content="IE=edge">
6       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7       <title>Document</title>
8   </head>
9   <body>
10
11      <button id="boton" >Click Me</button>
12      <script src="eventos.js"></script>
13  </body>
14  </html>
```

Cuando pulsamos el botón este es el resultado que obtenemos. Como vemos ahora el evento utilizado es click, que implica el que la función se ejecuta cuando hacemos un **click** sobre el botón.

```
C: > Ejercicios javascript code > JS eventos.js > button.addEventListener("click") callback
1   const button = document.getElementById("boton");
2   button.addEventListener("click", () => {
3       console.log("Hemos pulsado el botón");
4   });

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
Hemos pulsado el botón
```

Si lo que queremos es que se escriba cuando hacemos dobleclick sobre el botón, lo que tenemos que hacer es cambiar el evento **dblclick**, como vemos en la siguiente imagen.

```
C: > Ejercicios javascript code > JS eventos.js > ...
1   const button = document.getElementById("boton");
2   button.addEventListener("dblclick" , () => {
3       console.log("Hemos pulsado el botón");
4   });

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
Hemos pulsado el botón
```

Para comprobar los eventos de ratón, vamos a crear una caja en el documento html, en la que modificaremos las propiedades.

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta http-equiv="X-UA-Compatible" content="IE=edge">
6       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7       <title>Document</title>
8       <style>
9       .caja{
10          height: 100px;
11          width: 100px;
12          background-color: red;
13              }</style>
14  </head>
15  <body>
16      <div class="caja" id="box"></div>
17      <button id="boton" >Click Me</button>
18      <script src="eventos.js"></script>
19  </body>
20  </html>
```

Vamos a hacer que cuando entremos con el ratón en la caja, se modifique el color de la caja, y cuando salga vuelva a tener el mismo color inicial. Para ello utilizaremos los métodos **mouseenter** o **mouseleave**.

C: > Ejercicios javascript code > JS eventos.js > caja.addEventListener("mouseleave") callback

```
const caja = document.getElementById("box");
caja.addEventListener("mouseenter" , () => {
    caja.style.backgroundColor ="blue";
});
caja.addEventListener("mouseleave" , () => {
    caja.style.backgroundColor ="red";
});
```

Pero como hemos dicho que debemos intentar no hacer modificaciones desde javascript en el formato de nuestra página web, ¿cómo podemos hacer entonces estas modificaciones? Lo que podemos hacer es crear una página CSS e ir modificando las clases asociadas a los elementos, veamos cómo.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="eventos.css"/>
</head>
<body>
    <div class="cajared" id="box"></div>
    <button id="boton" >Click Me</button>
    <script src="eventos.js"></script>
</body>
```

Añadimos el archivo css a nuestra aplicación html, generando dos formatos distintos asociados a la caja, el rojo y el verde.

```
.cajared{
    height: 100px;
    width: 100px;
    background-color: red;
        }
.cajagreen{
        height: 100px;
        width: 100px;
        background-color: green;
            }
```

Lo que haremos ahora será reemplazar el código javascript, y cambiaremos la clase asociada al elemento cuando el ratón entra o sale de la caja, de la siguiente manera.

```
const caja = document.getElementById("box");
caja.addEventListener("mouseenter" , () => {
    caja.classList.replace("cajared","cajagreen");
});
caja.addEventListener("mouseleave" , () => {
    caja.classList.replace("cajagreen","cajared");
});
```

Ejercicio1. Crear una página donde se aplique los siguientes eventos a alguno de los elementos de una página web que creéis.

mousedown: Este evento se da cuando pulsamos el botón izquierdo del ratón, pero no lo soltamos.

mouseup: Este evento se produce cuando soltamos el botón izquierdo del ratón.

mousemove: Este evento se produce al mover el ratón.

Veamos algún ejemplo de eventos relacionados con el teclado, cuando pulsamos alguna tecla. Algunos de los eventos asociados al teclado son los siguientes:

keydown: evento que se produce cuando pulsamos una tecla.

keyup: evento que se produce cuando soltamos una tecla.

keypress: evento que se produce cuando pulsamos una tecla y no la soltamos.

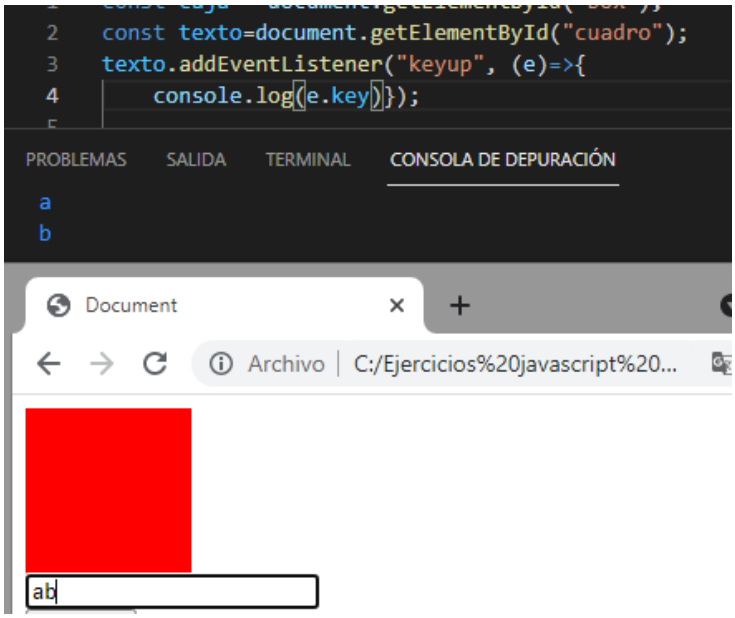
Veamos un ejemplo cuando pulsamos una tecla, para eso crearemos un cuadro de texto para que nos permita ver qué teclas son las que estamos pulsando.

```
10  <body>
11      <div class="cajared" id="box"></div>
12      <input type="text" id="cuadro"> <br>
13      <button id="boton" >Click Me</button>
14      <script src="eventos.js"></script>
15  </body>
```

Y esto es lo que vamos a escribir en el archivo js.

```
1  const caja = document.getElementById("box");
2  const texto=document.getElementById("cuadro");
3  texto.addEventListener("keydown", ()=>{
4  console.log("Hemos pulsado una tecla")});
5  texto.addEventListener("keyup", ()=>{
6  console.log("Hemos soltado la tecla")});
7  texto.addEventListener("keypress", ()=>{
8  console.log("Estamos pulsando una tecla")});
9
```

Sin embargo, en este caso no nos interesa solo saber que hemos pulsado una tecla, nos interesa además saber que tecla hemos pulsado. En este caso lo que haremos será añadir un evento a la función que asociamos al método **addEvenListener().** Además, como estamos trabajando con eventos de teclado, tenemos un atributo asociado al evento, llamado key, donde se guarda cuál es la tecla pulsada. Veámoslo en un ejemplo:



Aunque los atributos asociados pueden ser muchos, debemos prestar especial atención a target, que es uno de los que más se utilizan. Esta propiedad nos devuelve el elemento que hemos pulsado en la página web, y si lo asociados a target.textContent lo que nos devolverá será el contenido del elemento pulsado.

Cuando queremos controlar la tecla pulsada desde cualquier punto de la aplicación, es decir, desde la ventana, podemos utilizar **window.addEventListener("evento",callback).**

CREAR E INSERTAR ELEMENTOS

Como veremos hay muchas opciones de incluir elementos en el DOM, partiremos en este caso de un título, una lista vacía de momento y un cuadro de selección donde de momento no tenemos ninguna opción. Veremos cómo podemos añadir elementos desde javascript en html. Una de las formas más fáciles es la siguiente:

```
9   <body>
10      <h1 id="titulo">Crear elementos en el DOM</h1>
11      <ul id="dias"> </ul>
12          <select id="seleccionardias"></select>
13      <script src="CrearElementos.js"></script>
14  </body>
15  </html>
```

Para crear un elemento a la página html, lo que hacemos es utilizar el método **createElement(element)**, siempre debe ir precedido de **document**, ya que los elementos tienen que crearse en la página web. Y para añadir un elemento al DOM, usaremos **parent.appendChild(element).**

Para escribir texto en un elemento que ya existe utilizaremos el atributo textContent asociado al documento. Y como ya hemos hablado en otras ocasiones, para escribir HTML en un elemento utilizaremos element.innerHTML, dentro de ellas incluiremos el código HTML. Veamos cómo podemos hacerlo en el documento que ya tenemos creado, crearemos un cuadro de lista e incluiremos contenido a los elementos que ya existen.

```
const diasem=["lunes","martes","miércoles","jueves","viernes","sábado","domingo"];
const titulo=document.getElementById("titulo");
const dia=document.getElementById("dias");
const diaselec=document.getElementById("seleccionardias");
const listadias=document.createElement("li");
for(const diar of diasem){
    //Para acumular todos los días
    dia.innerHTML +=`<li> ${diar} </li>`;
}
```

Esta sería una forma de añadir todos los elementos a la lista, y como podemos ver en la imagen coincide. Sin embargo, cada vez que ejecutamos el bucle, se sobrescribe el árbol del DOM, lo que significa que se utilizan demasiados recursos del navegador. Para evitar esto se utilizará el método createDocumenFragment(), esto evita que se sobrescriba, y el gasto de recursos. Este método, utiliza una variable que se va actualizando, y cuando terminamos de escribir todo el código que queremos insertar es cuando se inserta. Veamos cómo podemos utilizar este método, y el resultado obtenido.

```
const diasem=["lunes","martes","miércoles","jueves","viernes","sábado","domingo"];
const titulo=document.getElementById("titulo");
const dia=document.getElementById("dias");
const diaselec=document.getElementById("seleccionardias");
const listadias=document.createElement("li");
const fragment=document.createDocumentFragment();
//variable que guardará el códgio a insertar
for(const diar of diasem){
    //Para acumular todos los días
    const lista=document.createElement('li');
   lista.textContent = diar;
   fragment.appendChild(lista);
}
dia.appendChild(fragment);
```

PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN

MOVERNOS POR EL DOM

Como hemos visto cuando creamos un elemento, debemos decir donde queremos incluir esos elementos, por lo que es muy importante que podamos identificar cada uno de los elementos que hay en el DOM. Vamos a explicar las funciones que existen para podemos utilizar para movernos por el DOM.

- **parentNode**: Devuelve el nodo padre del elemento que le pasamos, que no tiene porqué ser un elemento.
- **parentElement**: Devuelve el nodo elemento padre del que le pasemos como argumento. Los nodos del tipo **Document** y **DocumentFragment** no tienen un elemento padre, por lo que el resultado devuelto será null.
- **childNodes**: Devuelve todos los nodos hijos del elemento que pasamos como argumento.
- **children**: Devuelve todos los nodos elementos hijos
- **firstChild**: Devuelve el primer nodo hijo
- **firstElementChild**: devuelve el primer nodo elemento hijo

- **lastChild**: devuelve el último nodo hijo
- **lastElementChild**: devuelve el último nodo elemento hijo
- **hasChildNodes()**: devuelve true si el nodo tiene hijos y false si no tiene
- **nextSibling**: Devuelve el siguiente nodo hermano
- **nextElementSibling:** Devuelve el siguiente elemento hermano
- **previsousSibling:** Devuelve el anterior nodo hermano
- **previousElementSibling:** Devuelve el nodo anterior hermano

Todas estas funciones se pueden ir acumulando para ir moviéndonos por todo el DOM. Veremos algún ejemplo, para ello partimos de una página web donde tenemos un título y varias listas.

```
<body>
    <h1 id="titulo">Veamos como podemos recorrer el DOM</h1>
    <nav>
    <ul id="primeralista">
        <li>Primer elemento de la lista</li>
        <li>Segundo elemento de la lista</li>
        <li>Tercer elemento de la lista</li>
        <ul>
            <li>Subprimer elemento</li>
            <li>Subsegundo elemento</li>
        </ul>
    </ul>
    </nav>
    <script src="RecorrerDOM.js"></script>
</body>
```

Y veamos que el nodo padre de la lista “primeralista” es nav; y sin embargo el padre de document es null porque no existe un nodo padre de este mismo.

```
const titular=document.getElementById("titulo");
const listaprimera=document.getElementById("primeralista");

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

```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> nav
> nav
  null
```

Si queremos subir varios niveles, bastará con aplicar la propiedad parentElement tantas veces como necesitemos. Veamos que sucede en este caso.

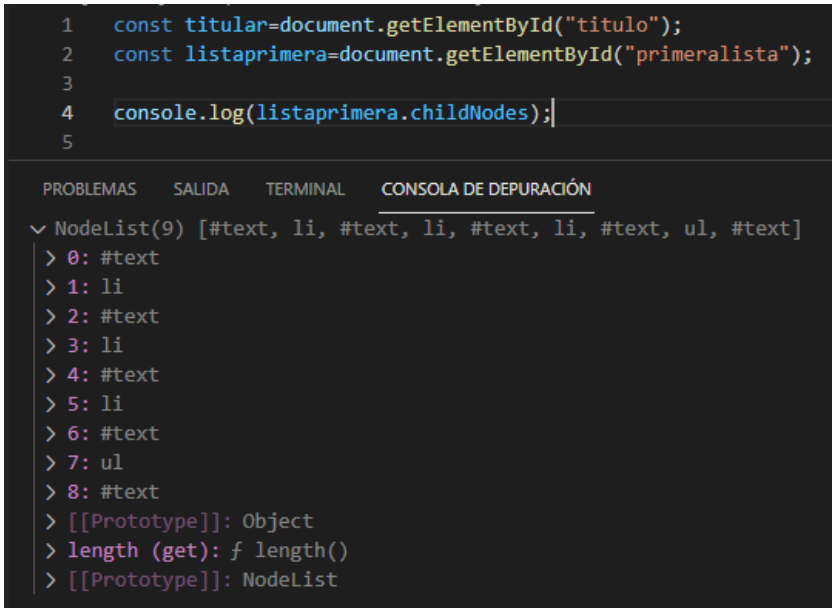
```
const titular=document.getElementById("titulo");
const listaprimera=document.getElementById("primeralista");

console.log(listaprimera.parentElement.parentElement);
console.log(document.parentElement);
```

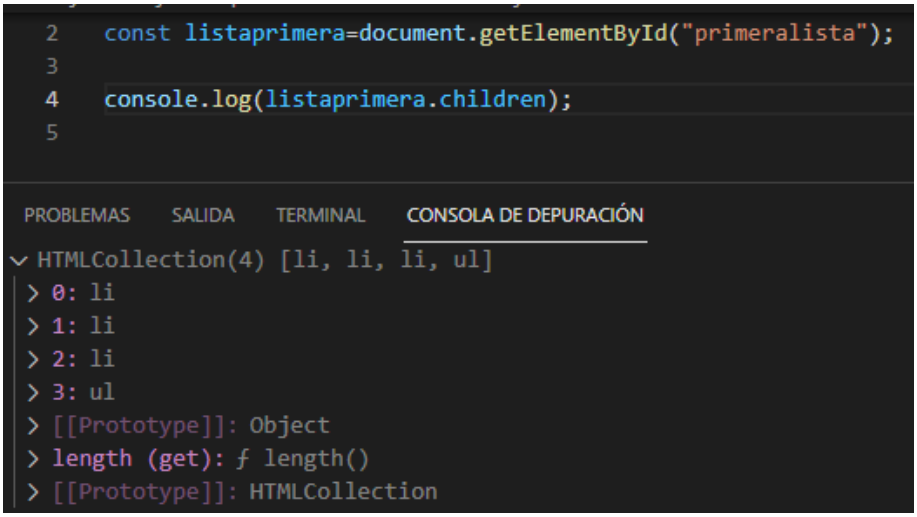
```
PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> body
  null
```

Como vemos en todas las propiedades tenemos la función donde nos aparece Element y donde no nos aparece esa palabra. La diferencia básica entre uno y otros es que cuando nos encontramos la palabra Element, lo que devuelve es el primer nodo de tipo elemento, así si tenemos un nodo que es de tipo salto de línea, o de tipo texto no se tendrá en cuenta.

Como vemos en el ejemplo siguiente, nos aparecen nodos de tipo texto que no representan elementos dentro de la página web.



Sin embargo, en este caso si utilizamos la opción children, solo nos aparecerán los nodos de tipo elemento.



Si añadimos un identificador al body y pedimos que nos muestre el primer nodo de tipo elemento nos aparecerá h1, si no lo ponemos lo que aparece es un nodo de tipo texto, como podemos ver a continuación.

```
1   const cuerpo=document.getElementById("cuerpo");
2   const listaprimera=document.getElementById("primeralista");
3   console.log(cuerpo.firstChild);
4   console.log(cuerpo.firstElementChild);
5

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> #text
> h1#titulo
```

Lo mismo nos pasará con el último hijo, y con las funciones que hemos nombrado anteriormente. En el siguiente ejemplo, vemos como podemos buscar elementos que estén al mismo nivel, es decir, elementos hermanos.

```
1   const titulo=document.getElementById("titulo");
2   const listaprimera=document.getElementById("primeralista");
3   console.log(listaprimera.parentElement.previousElementSibling);
4   console.log(titulo.nextElementSibling);
5

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN
> h1#titulo
> nav
```

Si no existiera el nodo hermano previo o siguiente, simplemente nos devolvería null.

INSERTAR, ELIMINAR Y CLONAR ELEMENTOS

Anteriormente hemos visto que para añadir elementos en un HTML podíamos utilizar la función apendChild, pero en este caso, el elemento siempre se añade detrás de los elementos que ya tenemos. En ocasiones nos puede interesar que el elemento nuevo sea insertado en una posición determinada, en este caso lo primero que debemos hacer es localizarnos dentro del DOM.

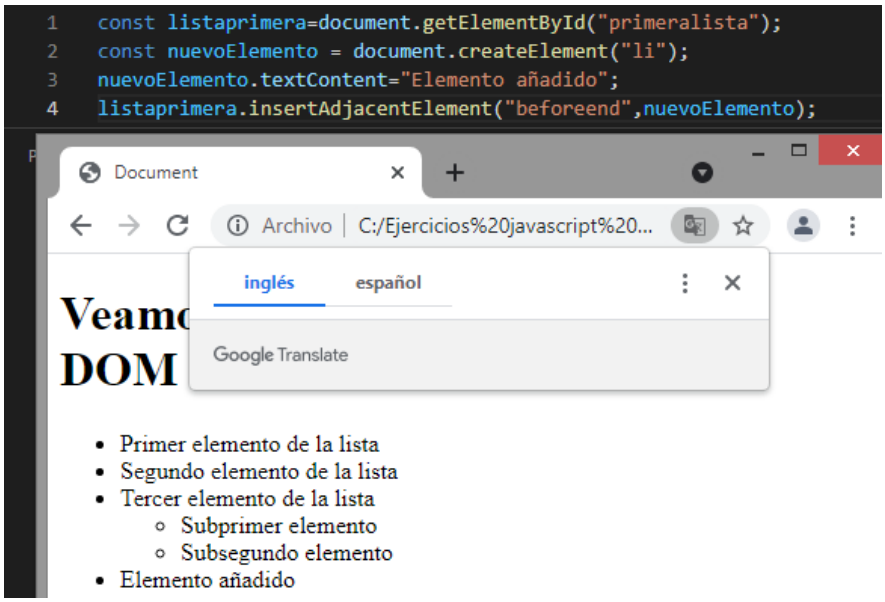
Alguna de las funciones que nos van a permitir hacer esto es **insertbefore**, para insertar antes de un elemento que indicamos. Veamos en el ejemplo anterior, cómo añadir un elemento a la lista que esté en la segunda posición.

Si lo que queremos añadir es un nodo hermano, utilizaremos la **función insertAdjacentElement**, como primer argumento a esta función le pasamos la posición donde queremos insertarlo, y las opciones son las siguientes:

- **beforebegin**: antes de que empiece (hermano del anterior).
- **afterbegin**: después de que empiece (primer hijo).
- **beforeend**: antes de que acabe (como último hijo).
- **afterend**: después de que acabe (como hermano siguiente).

Como vemos en el siguiente ejemplo, la nueva lista se añade como hermana de la anterior

Como elemento final de la lista, con beforeend.

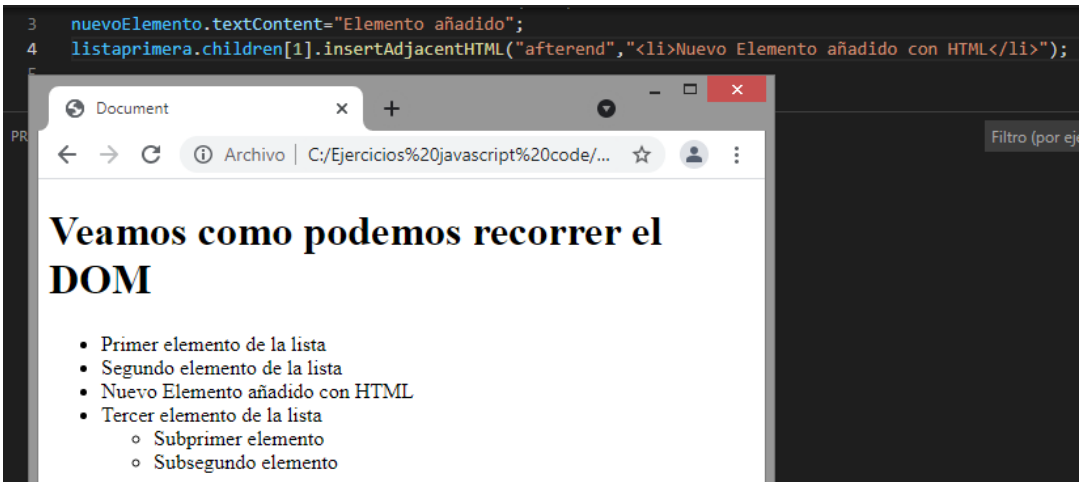


En ocasiones, podemos tener problemas si intentamos añadir un elemento en una posición que no exista, en este caso lo que haremos será asegurarnos que el elemento existe, para ello añadiremos el elemento que vamos a tener en cuenta cuando queremos añadir el elemento, de la siguiente forma:

listaprimera.children[2].insertAdjacentElement("afterend",nuevoElemento).

Ejercicio 2. Utiliza el resto de opciones de inserción para comprobar donde se añade el nuevo elemento creado. Si es necesario utiliza la posición donde crearlo.

Otra forma de insertar elementos es utilizar la **función insertAdjacentHTML,** en este caso como segundo parámetro añadiremos el código HTML del elemento que queremos añadir. En el primer parámetro se pueden incluir los mismos parámetros que para la función anterior. Veamos un ejemplo:



Ejercicio 3. Utiliza el resto de opciones de inserción para comprobar donde se añade el nuevo elemento creado.

1. REEMPLAZAR ELEMENTOS EN EL DOM

Otra de las acciones que podemos realizar en el DOM es reemplazar un elemento que existe en el DOM por otro. En este caso se le pasarán a la función que utilizamos para ello replaceChild dos parámetros, el primero que será el nuevo elemento que queremos añadir, y el segundo que será el elemento que queremos eliminar.

En este caso cambiamos el primer elemento de la lista por el nuevo elemento creado. También se puede utilizar **replacewith**(nuevo elemento). En este caso esta función se aplica directamente al elemento que queremos reemplazar **listaprimera.children[0].replacewith(elemento).**

2. AÑADIR ELEMENTOS EN EL DOM

Otras funciones que podemos utilizar para añadir elementos son las siguientes, que son similares a los parámetros asociados a las funciones Adjent.

- **parent.before():** Se utiliza para añadir antes de que empiece el elemento, es decir, será un hermano anterior.
- **parent.prepend():** después de que empiece, luego se añadirá como primer hijo.
- **parent.applend():** se insertará antes de que acabe, es decir, como último hijo.
- **parent.after():** se añadirá después de que acabe, es decir, como siguiente hermano.

Veamos un ejemplo con el archivo html anterior.

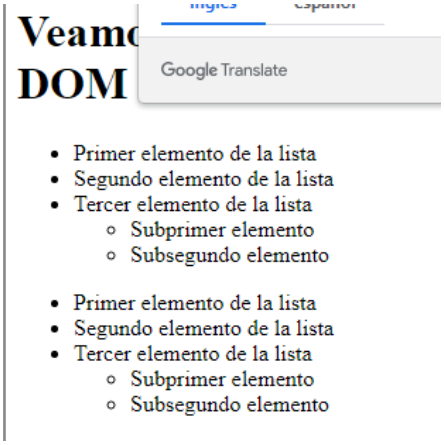
Podríamos utilizar de forma similar el resto de funciones que nombramos arriba, en función del la posición donde queremos posicionar el nuevo elemento.

3. CLONAR ELEMENTOS

La función que utilizaremos en este caso para clonar elementos es cloneNode. Veamos cómo podemos utilizar esta función, a la que le pasamos el argumento true si queremos que se clone el elemento.

```
const listaprimera=document.getElementById("primeralista");
const nuevoElemento = document.createElement("li");
nuevoElemento.textContent="Elemento añadido";
listaprimera.after(listaprimera.cloneNode(true));
```

Esta sería la forma de usar el función y en la siguiente página veremos el resultado obtenido, como vemos se repite después de la lista que teníamos creada de nuevo la lista.

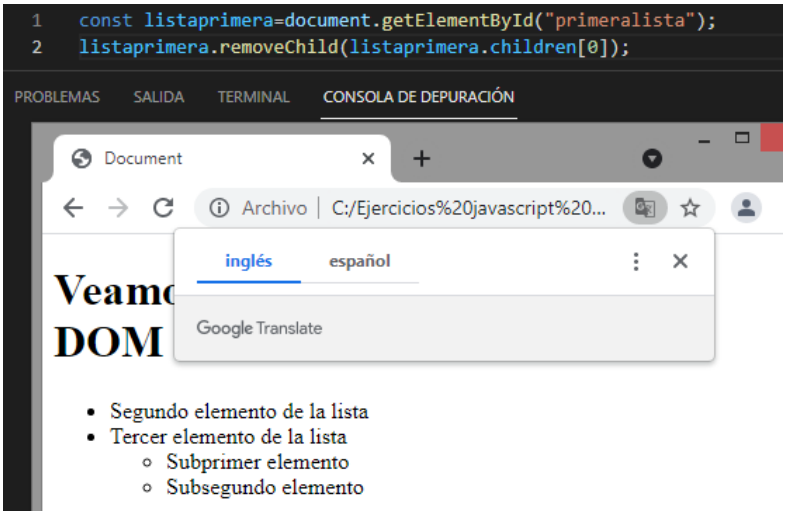


4. ELIMINAR ELEMENTOS

La función que utilizaremos en este caso para eliminar elementos es remove, si queremos eliminar todo el nodo, y removeChild si lo que queremos es eliminar uno de los hijos del elemento. Veamos un ejemplo.

```
2  const nuevoElemento = document.createElement("li");
3  nuevoElemento.textContent="Elemento añadido";
4  listaprimera.remove();
```

En este caso solo nos aparecerá la lista. Y en el caso anterior lo que pasará es que eliminará el nodo hijo que especificamos.



5. USANDO FRAGMENTOS

En algunas ocasiones, nos puede resultar muy interesante utilizar fragmentos. Los fragmentos son una especie de documento paralelo, aislado de la página con la que estamos trabajando, que tiene varias características:

- No tiene elemento padre. Está aislado de la página o documento.
- Es mucho más simple y ligero (mejor rendimiento)
- Si necesitamos hacer cambios consecutivos, no afecta al reflow (repintado del documento).

Es una estrategia muy útil para usarlo de documento temporal y no realizar cambios consecutivos, con su impacto de rendimiento. Para crearlos, necesitaremos usar la función document.createDocumentFragment().

La función createDocumentFragment() es un fragmento que podremos usar para almacenar en su interior un pequeño DOM temporal, que luego añadiremos en nuestro DOM principal.

Como podemos ver en el siguiente ejemplo, usamos el fragmento fragment generado como ubicación temporal donde hacer todos los cambios del DOM que necesitemos. Una vez terminemos nuestra lógica y tengamos el DOM definitivo, lo insertamos como hemos visto con un appendChild.

```
const fragment = document.createDocumentFragment();

for (let i = 0; i < 5000; i++) {
  const div = document.createElement("div");
  div.textContent = `Item número ${i}`;
  fragment.appendChild(div);
}

document.body.appendChild(fragment);
```

.