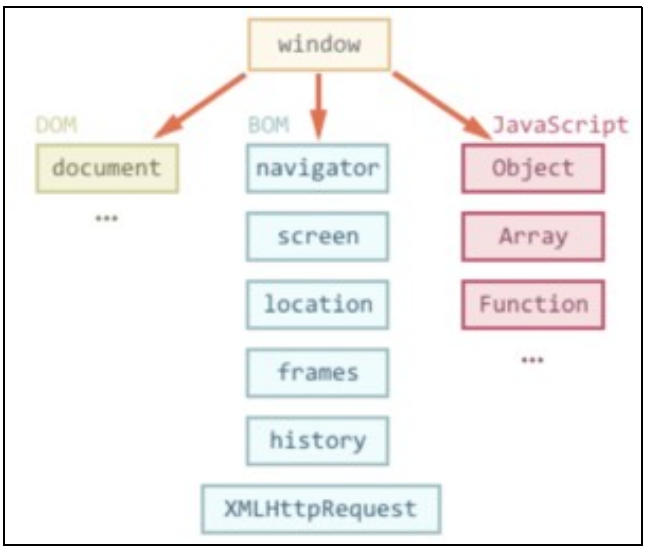
Objetos de más alto nivel en JavaScript

El código JavaScript ejecutado en un navegador tiene acceso a un número alto de objetos. Estos objetos pueden estar divididos en varios tipos:

- **Objetos del núcleo ECMAScript:** Todos los objetos mencionados hasta este momento.
- **DOM (*Document Object Model*):** Objetos que nos permiten hacer algo con la página cargada en estos momentos, objeto que se denomina document y todos los que cuelgan de él.
- **BOM (*Browser Object Model*):** Objetos externos al documento web en sí, como la ventana del navegador y la pantalla del equipo.



Objetos alto nivel JavaScript

El **DOM** es un estándar definido por el W3C y tiene diferentes versiones, llamadas niveles, como son DOM Level1, DOM Level 2, etc. Los navegadores actuales implementan en cierto grado el estándar DOM pero, en general, todos implementan completamente el DOM Level 1, el denominado DOM Level 0 (aunque no existe ese estándar en sí).

El Modelo de Objetos del Documento (DOM), permite ver el mismo documento de otra manera, describiendo el contenido del documento como un conjunto de objetos, sobre los que un programa de Javascript puede interactuar.

Según el W3C, "el DOM es una interfaz de programación de aplicaciones (API), para documentos válidos HTML y bien construidos XML". Define así la estructura lógica de los documentos, y el modo en el que se acceden y se manipulan.

Pero, antes de adentrarnos en el DOM, nos centraremos en los objetos que forman el **BOM**. Históricamente, el BOM no es parte de ningún estándar. El estándar HTML5 implementa el comportamiento común de los navegadores e incluye objetos BOM comunes. Además, los dispositivos móviles vienen con sus objetos específicos (y HTML5 tiene como objetivo estandarizar esos objetos también), que tradicionalmente no eran necesarios para los equipos de escritorio, pero tienen sentido en un mundo móvil, como geolocalización, acceso a la cámara, vibración, eventos táctiles, telefonía y SMS.

Referencias interesantes para aprender más sobre BOM y DOM de lo que veremos aquí:

- Mozilla DOM.
- Wiki de Mozilla HTML5.
- Documentación de Microsoft para Internet Explorer.
- https://www.w3.org/DOM/DOMTR.

Sumario

El BOM

El **BOM** es una colección de objetos que dan acceso al navegador y a la pantalla del ordenador. Estos objetos veremos que son accesibles desde el objeto global **window**.

Así, ahora, nos centraremos en objetos de alto nivel, que se encontrarán frecuentemente en las aplicaciones de JavaScript: **window**, **location**, **navigator**, **screen** y **history**. El objetivo, no es solamente indicar las nociones básicas para que se puedan comenzar a realizar tareas sencillas, sino también, el prepararse para profundizar en las propiedades y métodos, gestores de eventos, etc. que se utilizarán posteriormente.

Objeto window

En la jerarquía de objetos tenemos, en la parte superior, el objeto **window**.

Este objeto está situado justamente ahí, porque es el contenedor principal de todo el contenido que se visualiza en el navegador. Tan pronto como se abre una ventana (**window**) en el navegador, incluso aunque no se cargue ningún documento en ella, este objeto **window** ya estará definido en memoria.

Además de la sección de contenido del objeto **window**, que es justamente dónde se cargarán los documentos, el campo de influencia de este objeto, abarca también las dimensiones de la ventana, así como todo lo que rodea al área de contenido: las barras de desplazamiento, barra de herramientas, barra de estado, etc.

Como se ve en el gráfico anterior de la jerarquía de objetos, debajo del objeto **window** tenemos otros objetos como el **navigator**, **screen**, **history**, **location** y el objeto **document**. Este objeto **document** será el que contendrá todos los objetos existentes dentro de nuestra página HTML.

Acceso a propiedades y métodos

Para acceder a las propiedades y métodos del objeto **window**, lo podremos hacer de diferentes formas, dependiendo más de nuestro estilo, que de requerimientos sintácticos.

Así, la forma más lógica y común de realizar esa referencia, incluiría el objeto **window** tal y como se muestra en este ejemplo:

```
window.nombrePropiedad
window.nombreMétodo( [parámetros] )
```

Como se puede ver, como siempre, los parámetros van entre corchetes, indicando que son opcionales y que dependerán del método al que estemos llamando.

Un objeto **window** también se podrá referenciar mediante la palabra especial **self**, cuando estamos haciendo la referencia desde el propio documento contenido en esa ventana:

```
self.nombrePropiedad
self.nombreMétodo( [parámetros] )
```

Podremos usar cualquiera de las dos referencias anteriores, pero intentaremos dejar la palabra reservada **self**, para *scripts* más complejos en los que tengamos múltiples ventanas.

Debido a que el objeto **window** siempre estará presente cuando ejecutemos nuestros *scripts*, podremos omitirlo, en referencias a los objetos dentro de esa ventana. Así que, si escribimos:

```
nombrePropiedad
nombreMétodo( [parámetros] )
```

También funcionaría sin ningún problema, porque se asume que esas propiedades o métodos, son del objeto de mayor jerarquía (el objeto **window**) en el cuál nos encontramos.

Gestión de ventanas. Métodos window.open() / close()

Un *script* no creará nunca la ventana principal de un navegador. Es el usuario, quien realiza esa tarea abriendo una URL en el navegador o un archivo desde el menú de abrir. Pero, sin embargo, un *script* que esté ejecutándose en una de las ventanas principales del navegador, sí que podrá crear o abrir nuevas sub-ventanas.

El método que genera una nueva ventana es **window.open()**. Este método contiene hasta tres parámetros, que definen las características de la nueva ventana: la URL del documento a abrir, el nombre de esa ventana y su apariencia física (tamaño, color,etc.).

Por ejemplo, podemos ver la siguiente instrucción que abre una nueva ventana de un tamaño determinado y con el contenido de un documento HTML:

```
var subVentana = window.open("nueva.html", "nueva", "height=800, width=600");
```

Lo importante de esa instrucción, es la asignación que hemos hecho en la variable **subVentana**. De esta forma podremos, a lo largo de nuestro código, referenciar a la nueva ventana desde el *script* original de la ventana principal. Por ejemplo, si quisiéramos cerrar la nueva ventana desde nuestro *script*, simplemente tendríamos que hacer:

```
subVentana.close();
```

Aquí sí que es necesario especificar **subVentana**, ya que si escribiéramos **window.close()**, **self.close()** o **close()** estaríamos intentando cerrar nuestra propia ventana (previa confirmación), pero no la **subVentana** que creamos en los pasos anteriores.

Métodos window.moveTo() y window.resizeTo()

Se trata de dos métodos que nos permiten mover y cambiar el tamaño de las ventanas:

```
//Mueve la ventana a la posición de la pantalla x= 100 e y = 100, con origen en la esquina superior-izquierda.
window.moveTo(100, 100)
//Mueve la ventana 10 pixels a la derecha y 10 pixels para arriba.
window.moveBy(10, -10)
//También, con dos parámetros que cambian el tamaño de la ventana.
window.resizeTo(x, y) y window.resizeBy(x, y)
```

Ejemplo donde se abre y se cierra una página desde otra principal

- Código HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>JavaScript</title>

</head>
```

```
<h1>Abrimos y cerramos ventanas</h1>
<form>
<p>
<input type="button" id="crear-ventana" value="Crear Nueva Ventana">
<input type="button" id="cerrar-ventana" value="Cerrar Nueva Ventana">
</p>
</form>


<script src="prueba1.js"></script>
</body>
</html>
```

- Código JavaScript:

```
//Variable ventana definida para que sea accesible desde las dos funciones.
var nuevaVentana;
//Función que crea una Nueva Ventana
function crearNueva() {
nuevaVentana = window.open(
"https://www.iessanclemente.net",
"",
"height=400,width=800"
);
}
//Función que cierra una ventana si esta existe
function cerrarNueva() {
if (nuevaVentana) {
nuevaVentana.close();
nuevaVentana = null;
}
}

//Escuchadores para los botones de crear y cerrar nueva ventana:
document.getElementById("crear-ventana").addEventListener("click", crearNueva);
document.getElementById("cerrar-ventana").addEventListener("click", cerrarNueva);
```

Propiedades y métodos de los objetos window

- **Propiedades del objeto *window***

Propiedad	Descripción
closed	Devuelve un valor Boolean indicando cuando una ventana ha sido cerrada o no.
defaultStatus	Ajusta o devuelve el valor por defecto de la barra de estado de una ventana.
document	Devuelve el objeto document para la ventana.
history	Devuelve el objeto history de la ventana.
location	Devuelve la Localización del objeto ventana (URL del fichero).
name	Ajusta o devuelve el nombre de una ventana.
navigator	Devuelve el objeto navigator de una ventana.
opener	Devuelve la referencia a la ventana que abrió la ventana actual.
parent	Devuelve la ventana padre de la ventana actual.
self	Devuelve la ventana actual.
status	Ajusta el texto de la barra de estado de una ventana.

- **Métodos del objeto *window***

Propiedad	Descripción
Método	Descripción
alert()	Muestra una ventana emergente de alerta y un botón de aceptar.
blur()	Elimina el foco de la ventana actual.
clearInterval()	Resetea el cronómetro ajustado con setInterval().

Propiedad	Descripción
setInterval()	Llama a una función o evalúa una expresión en un intervalo especificado (en milisegundos).
close()	Cierra la ventana actual.
confirm()	Muestra una ventana emergente con un mensaje, un botón de aceptar y un botón de cancelar.
focus()	Coloca el foco en la ventana actual.
open()	Abre una nueva ventana de navegación.
prompt()	Muestra una ventana de diálogo para introducir datos.

Utilizando la propiedad window.navigator

navigator es un objeto que tiene información del navegador y sus capacidades.

Podemos crear una función que nos aporte información del navegador del siguiente modo:

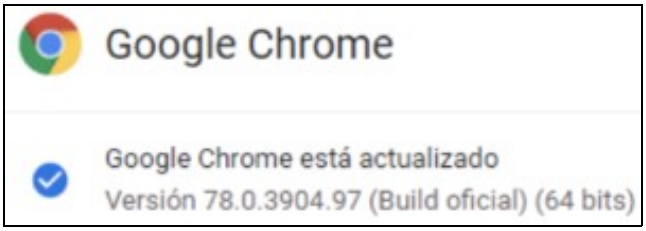
```
function leerIdentidadNavegador() {
return navigator.appName + " " + navigator.appVersion;
} //Fin leerIdentidadNavegador

console.log(leerIdentidadNavegador());
```

Que nos devolverá:

Netscape 5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.97 Safari/537.36

Siendo el Navegador un Google Chrome en este caso:



navigator
Pero, cuidado, pues muchos navegadores permiten al usuario modificar esa propiedad y, por lo tanto, no nos podremos fiar de ella para verificar el navegador que estamos utilizando.

Para ver todas las propiedades y métodos del objeto navigator podemos, simplemente, abrir la consola del navegador, escribir navigator y pulsar *Enter*.

navigator

- **Propiedades del objeto navigator**

Propiedad	Descripción
Propiedad	Descripción
appCodeName	Cadena que contiene el nombre en código del navegador.

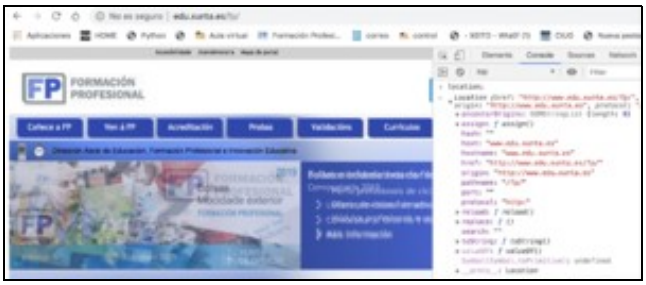
Propiedad	Descripción
appName	Cadena que contiene el nombre del cliente.
appVersion	Cadena que contiene información sobre la versión del cliente.
cookieEnabled	Determina si las cookies están o no habilitadas en el navegador.
platform	Cadena con la plataforma sobre la que se está ejecutando el programa cliente.
userAgent	Cadena que contiene la cabecera completa del agente enviada en una petición HTTP. Contiene la información de las propiedades appCodeName y appVersion.

- **Métodos del objeto navitagor**

Propiedad	Descripción
Método	Descripción
javaEnabled()	Devuelve true si el cliente permite la utilización de Java, en caso contrario, devuelve false.

Utilizando la propiedad window.location

La propiedad **location** es un objeto que contiene información sobre la URL de la página actualmente cargada. Por ejemplo, **location.href** es la URL completa y **location.hostname** es solamente el dominio. Con un simple bucle, podemos ver todas las propiedades del objeto **location**.



location
Podríamos realizar una función recursiva que nos muestre todas las propiedades del objeto location de la web cargada en esos momentos:

```
function verLocation() {
let windowLocation = [];
for (let i in window.location) {
if (typeof window.location[i] === "string") {
windowLocation.push(window.location[i]);
}
}
return windowLocation;
}

//Luego podemos llamarla
console.log(verLocation());
```

Propiedades del objeto location

Propiedad	Descripción
Propiedad	Descripción
hash	Cadena que contiene el nombre del enlace, dentro de la URL.
host	Cadena que contiene el nombre del servidor y el número del puerto, dentro de la URL.
hostname	Cadena que contiene el nombre de dominio del servidor (o la dirección IP), dentro de la URL.
href	Cadena que contiene la URL completa.
pathname	Cadena que contiene el camino al recurso, dentro de la URL.
port	Cadena que contiene el número de puerto del servidor, dentro de la URL.
protocol	Cadena que contiene el protocolo utilizado (incluyendo los dos puntos), dentro de la URL.

Propiedad	Descripción
search	Cadena que contiene la información pasada en una llamada a un script, dentro de la URL.

Métodos del objeto location:

Propiedad	Descripción
assign()	Abre una web (window.location.href sería una alternativa a ese método)
replace()	Igual que assign() pero no agrega entrada en el historial del navegador
reload()	Recargar la página actual (location = location sería una alternativa a este método)

Utilizando la propiedad window.history

Podemos acceder al historial de una ventana del navegador. El historial es la lista de visitas del usuario. Así y todo, por motivos de seguridad, únicamente podemos desplazarnos por el historial; no podemos obtener las URL de las páginas incluidas en el mismo.

Para desplazarnos por el historial utilizaremos la propiedad **history** del objeto **window**. Veamos unos ejemplos:

```
> window.history.length;//5 (por ejemplo)
> window.history[0];              //No devuelve nada
> history.forward();              //Una "adelante"
> history.back();               //Una "atrás"
> history.go(-1);               //Una atrás
> history.go(0);              //Recargar página actual
```

Utilizando la propiedad window.screen

Aunque la mayor parte de la información sobre el sistema del usuario se encuentra oculta por motivos de seguridad, se puede conseguir determinados datos sobre el monitor con ayuda del objeto **screen**.

window.screen
La diferencia entre **height** y **availHeight** es que **height** es el escritorio completo, mientras que **availHeight** es el tamaño sustrayendo los menús del sistema operativo como, por ejemplo, la barra de tareas de Windows (lo mismo ocurre con **width** y **availWidth**).

También existe la propiedad **DPR**, **devicePixelRatio**, esta propiedad nos indica la ratio entre los *pixels* lógicos y los *pixels* físicos de un dispositivo. Por ejemplo, en un portátil, *tablet* o móvil con pantalla retina, el valor suele ser 2 o mayor. Aquí tienes un enlace interesante donde explican como utilizar este parámetro cuando queremos que en nuestra web se muestren bien el texto y las imágenes.

Propiedades del objeto screen

Propiedad	Descripción
Propiedad	Descripción
availHeight	Devuelve la altura de la pantalla (sin incluir la barra de tareas de Windows)
availWidth	Devuelve el ancho de la pantalla (sin incluir la barra de tareas de Windows)
colorDepth	Devuelve la profundidad de bits de la paleta de colores.
height	Devuelve la altura total de la pantalla.
PixelDepth	Devuelve la resolución de color (en bits por pixel) de la pantalla.
width	Devuelve el ancho total de la pantalla.

Utilizando los métodos window.setTimeout() y setInterval()

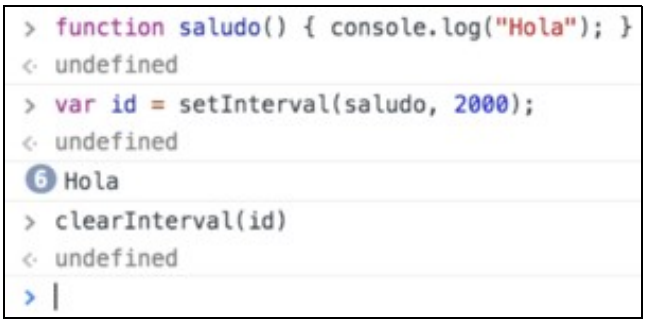
Los métodos setTimeout() y setInterval() permiten programar la ejecución de alguna parte del código de JavaScript.

- **setTimeout()** : Se ejecuta una función después de un tiempo pasado en milisegundos. Ejemplo:

```
<script type="text/javascript">
    function susto() { alert('Boooooo!') };
    setTimeout(susto, 3000);
</script>
```

Esta función devuelve un entero que representa el ID de la ejecución de esa función. Este ID es utilizado para cancelar la ejecución, empleando para ello el método **clearTimeout()**.

- **setInterval()** : Se ejecuta una función repetidamente con un intervalo de tiempo configurado en milisegundos.



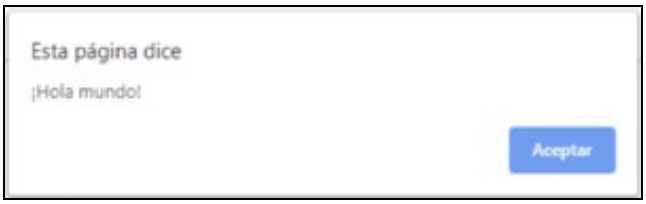
setinterval
Cuadros de diálogo de sistema

Además de las ventanas de navegador emergentes, existen otras técnicas para mostrar información al usuario que recurren a los métodos **alert()**, **confirm()** e **input()** del objeto **window**.

- **alert()**

La sitnaxis de la invocación **alert()** ya es conocida pues se ha utilizado en muchos ejemplos. Este método acepta un argumento, el texto que se va a mostrar al usuario.
Al invocar **alert()**, el navegador crea un cuadro de mensaje del sistema que muestra el texto proporcionado junto a un botón **OK (Aceptar)**.

```
alert("¡Hola mundo!");
```



alert()

- **confirm()**

  El cuadro de diálogo de confirmación es similar al de advertencia ya que muestra un mensaje al usuario. La principal diferencia entre amos es que muestra un botón **Cancel (Cancelar)** junto al botón **OK (Aceptar)**, que permite al usuario indicar si debe realizarse alguna acción.

  ```
  confirm("¿Estás seguro?");
  ```

  Para determinar si el usuario ha pulsado el botón **OK** o el botón **Cancel**, el método **confirm()** devuelve un valor booleano: **true -> OK** y **false -> Cancel**.

confirm()

- **prompt()**

  En este caso, se solicita una entrada del usuario. Junto con los botones **OK** y **Cancel**, el cuadro de diálogo incluye un cuadro de texto en el que el usuario debe introducir determinados datos. El método **prompt()** acepta dos argumentes: el texto que mostrar al usuario y el valor predeterminado del cuadro de texto (que puede ser una cadena vacía).

  ```
  var nombre = prompt("¿Cuál es tu nombre?", "Manuel");
  ```

  El valor del cuadro de texto se devuelve como valor de función si se pulsa el botón **OK**; si se hace *click* en el botón **Cancel**, se devuelve **null**.

prompt()

Para finalizar, decir que los cuadros de diálogo **son ventanas del sistema**, lo que significa que su aspecto puede variar en función del sistema operativo empleado (y, en ocasiones del navegador). También significa que carecemos de control sobre aspectos de la ventana, las fuentes, los colores, etc... También, saber que **los cuadros de diálogo son modales**, es decir, el usuario no puede hacer nada en el navegador hasta que cierre el cuadro de diálogo con los botones **OK** o **Cancel**. Así, es un método habitual para controlar el comportamiento del usuario y garantizar que la información importante se entrega de forma segura.

La barra de estado

La barra de estado es la zona del borde inferior en la que se muestra información al usuario.

Por lo general, la barra de estado indica al usuario cuándo se abre la página y cuándo termina de abirse; sin embargo, puede establecerse su valor con dos propiedades del objeto **window**:

- **status** : Cambia momentáneamente el texto de la barra de estado.
- **defaultStatus** : lo cambia mientras el usuario se encuentre en la página.

Por ejemplo, para utilizar un mensaje predeterminado para la barra de estado al abrir por primera vez la página:

```
window.defaultStatus = "Estás accediendo a la web del IES San Clemente."
```

También puede mostrar información sobre un determinado enlace cuando el usuario desplace el ratón sobre el mismo: </source lang="javascript"> Xunta </source>

Resulta especialmente útil al programar con JavaScript, ya que, por defecto, los navegadores muestran el valor del atributo **href** en la barra de estado cuando el usuario desplaza el ratón. Así, al establecer **window.status** se ocultan los detalles de la implementación del vínculo:

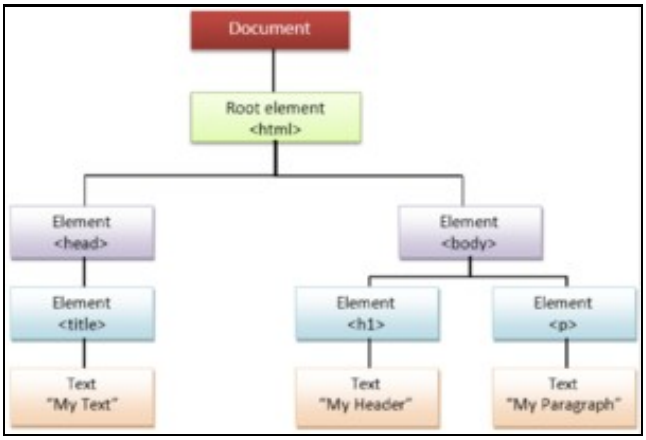
```
<a href="javascript:irAotraWeb(1,2,3,4)" onmouseover="window.status='Web de la Xunta.'">Xunta</a>
```

Así y todo, no se recomienda sobreutilizar la barra de estado, pues se puede volver una distracción para la navegación.

El DOM

El DOM representa un documento XML o HTML como un árbol de nodos. Utilizando métodos y propiedades de DOM, se puede acceder a todos los elementos de una página, modificarlos, eliminarlos y añadir nuevos. No es algo exclusivo de JavaScript, es una API que también se puede manejar desde otros lenguajes, como PHP por ejemplo.

A continuación se ve un gráfico ejemplo del DOM. Sin duda, es importante tener este árbol de objetos en mente porque va a ser la guía a lo largo de toda esta unidad.



El DOM

Para ver este "árbol" en una web real podemos ejecutar el siguiente código en el navegador que, a medida que avancemos en el tema, iremos entendiendo poco a poco:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8" />
<title>Probando el DOM</title>
<style type="text/css">
.callout {
border: solid 1px #ff0080;
margin: 2px 4px; padding: 2px 6px; }
.code {
background: #ccc; margin: 1px 2px;
padding: 1px 4px; font-family: monospace; }
</style>
</head>

<body>
  <header>
    <h1>HTML simple</h1>
  </header>
  <div id="content">
```

```
    <p>Este es un archivo HTML muy <i>simple</i></p>
    <div class="callout">
      <p>Este texto sí es importante!!</p>
    </div>
    <p>IDs (como <span class="code">#content</span>) son únicos
    (sólo puede existir uno por página).</p>
    <p>Clases (como <span class="code">.callout</span>)
    pueden ser utilizadas en muchos elementos.</p>
    <div id="callout2" class="callout fancy">
      <p>Un elemento HTML puede tener múltiples clases.</p>
    </div>
   </div>

<script type="text/javascript">
 function printDOM(node, prefix) {
        console.log(`${prefix} ${node.nodeName}  :  ${node.nodeType}  ->  ${node.nodeValue}`);
      for(let i=0; i<node.childNodes.length; i++) {
        printDOM(node.childNodes[i], `${prefix} \t`);
      }
    }
    printDOM(document, '');
</script>

</body>
</html>
```

Como veremos con más detenimiento más adelante, todos los nodos del árbol tienen las propiedades nodeType y nodeName:

- **nodeType** es un entero que identifica el tipo de nodo que es (en la siguiente web podemos ver los posibles valores).
- **nodeName** es una propiedad de solo lectura que devuelve el nombre del nodo.
- **nodeValue** es otra propiedad interesante de los nodos, muchos de ellos vemos que su valor es *null' pero, para los nodos tipo texto es el texto que se muestra por pantalla.*

Si nos centramos sólo en el anidamiento, vemos en el ejemplo anterior, la complejidad que va a tener acceder a cada uno de los elementos de la página, pero el **DOM** nos provee de métodos más directos para referirnos a cada uno de ellos.

- Por ejemplo, el objeto **TreeWalker** es un poderoso objeto **DOM** que permite filtrar fácilmente y crear colecciones personalizadas de nodos en el documento. Veamos un ejemplo para entender su funcionamiento:

```
var rootnode = document.getElementById("content");
var walker=document.createTreeWalker(rootnode, NodeFilter.SHOW_ELEMENT, null, false);

//El nodo de inicio (root node)
console.log(walker.currentNode.tagName); //Elemento con id=content

//Recorrer todos los subnodos y verlos por consola
while (walker.nextNode()){
console.log(walker.currentNode.tagName); //Vemos los subnodos
}
console.log("-----------------------------------------")
//Volver al primer nodo
walker.currentNode=rootnode; //reset TreeWalker
console.log(walker.firstChild().tagName); //Primer hijo de rootnode
```

  Nota sobre la Terminología: El concepto de árbol (**tree**) es sencillo e intuitivo, y se presta a una terminología igualmente intuitiva. El padre (**parent**) de un nodo es su padre directo y un hijo (**child**) es un hijo directo. El término descendiente (**descendant**) se usa para referirse a un hijo, o al hijo de un hijo, etc. El término ancestro (**ancestor**) se utiliza para referirse a un padre, al padre del padre, etc.

Objeto document

Cada documento cargado en una ventana del navegador, será un objeto de tipo **document**.

El objeto **document** proporciona a los *scripts*, el acceso a todos los elementos HTML dentro de una página. Este objeto forma parte además del objeto **window**, y puede ser accedido a través de la propiedad **window.document** o directamente **document** (ya que podemos omitir la referencia a la ventana actual).

Colecciones dentro del objeto document

Se creará una colección (un tipo de array) para cada tipo de elementos:

Colección	Descripción
anchors[ ]	Es un array que contiene todos los hiperenlaces del documento.
forms[ ]	Es un array que contiene todos los formularios del documento.
images[ ]	Es un array que contiene todas las imágenes del documento.
links[ ]	Es un array que contiene todos los enlaces del documento.

Propiedades del objeto document

El objeto **document** tiene estas propiedades:

Propiedad	Descripción
cookie	Devuelve todos los nombres/valores de las cookies en el documento.
domain	Cadena que contiene el nombre de dominio del servidor que cargó el documento.
referrer	Cadena que contiene la URL del documento desde el cuál llegamos al documento actual.
title	Devuelve o ajusta el título del documento.
URL	Devuelve la URL completa del documento.

Métodos del objeto document

El objeto **document** tiene los siguientes métodos:

Método	Descripción
close( )	Cierra el flujo abierto previamente con document.open().
getElementById( )	Devuelve el elemento identificado por el id escrito entre paréntesis.
getElementsByName( )	Devuelve una lista de elementos identificados por el atributo name escrito entre paréntesis.
getElementsByTagName( )	Devuelve una lista de elementos identificados por el tag o la etiqueta escrita entre paréntesis.
getElementsByClassName( )	Devuelve una lista de elementos identificados por el class definido.
getElementsByTagNameNS( )	Devuelve una lista de elementos cuyo nombre pertenece a un determinado *namespace*.
open( )	Abre el flujo de escritura para poder utilizar document.write( ) o document.writeln en el documento.
write( )	Para poder escribir expresiones HTML o código de JavaScript dentro de un documento.
writeln( )	Lo mismo que write( ) pero añade un salto de línea al final de cada instrucción.

Accediendo a los nodos DOM

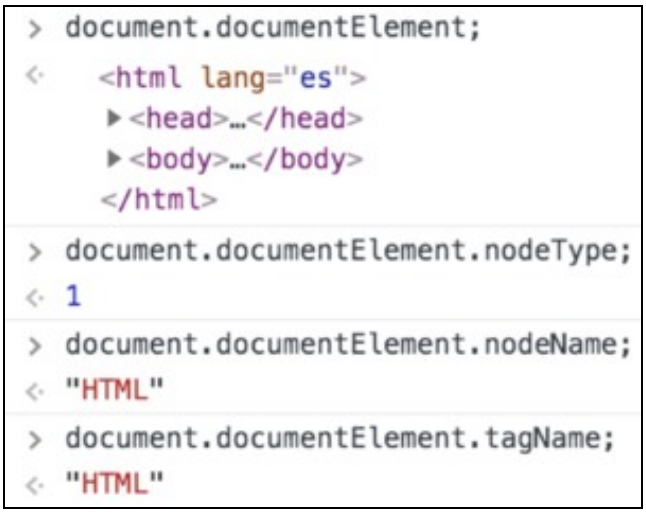
Antes de validar las entradas de un usuario en los formularios o intercambiar imágenes, se necesita tener acceso a los elementos que queremos leer o inspeccionar.

Como ya se comentó antes, todos los nodos, incluyendo el nodo **document**, nodos de texto, nodos de elementos y nodos de atributos tienen las propiedades **nodeType**, **nodeName** y **nodeValue**.

- **nodeType** : Hay 12 tipos de nodos, representados por enteros. Como pudimos ver con el código del punto anterior, el tipo de nodo **document** es el 9. Los más comunmente usados son 1 (elemento), 2 (atributo) y 3 (texto).
- **nodeName** : Los nodos también tienen nombres (propiedad **tagName**, **#text** si es un nodo de texto,...).
- **nodeValue** : Los nodos también tienen "valores". Por ejemplo, para nodos de texto, el valor es el texto actual.

DocumentElement

Ahora, vamos a movernos por el árbol. Para los documentos HTML, el elemento principal es la etiqueta **<html>**. Para acceder al elemento principal, debemos utilizar la propiedad **documentElement** del objeto **document**.



documentElement

Nodos hijo

Para comprobar que un nodo tiene un hijo empleamos el método **hasChildNodes()**.

El elemento HTML tiene tres hijos: el elemento **head**, el elemento **body** y el espacio en blanco existente entre ellos (muchos navegadores no cuentan este espacio en blanco). A esos tres elementos podemos acceder a ellos utilizando la colección **childNodes**, lo haremos del siguiente modo:

Nodos Hijo

Todo hijo tiene acceso a su padre utilizando la propiedad **parentNode**, podemos verlo en el siguiente ejemplo:

```
> document.documentElement.childNodes[1].parentNode; //<html>...
```

Si utilizamos el código HTML anterior, podemos ver que el número de hijos nodos existentes dentro del elemento **<body>** son 7:

Ejemplo Nodos Hijo

Elementos hermanos

Otras propiedades interesantes para navegar por el DOM, una vez seleccionado un elemento, son nextSibling y previousSibling. Veamos el siguiente código:

```
<script type="text/javascript">

//Accedemos al div con id="callout2"
const dCallout2 = document.getElementById('callout2');

//nextSibling : recorremos el árbol hacia delante
  console.log("--Hacia abajo de 'callout2'--");
  //Primer siguiente hermano
    console.log(dCallout2.nextSibling); // #text
  //Segundo siguiente hermano
console.log(dCallout2.nextSibling.nextSibling); // null

//previousSibling : recorremos el árbol hacia atrás
  console.log("--Hacia arriba de 'callout2'--");
  //Primer anterior hermano
    console.log(dCallout2.previousSibling); // #text
  //Segundo anterior hermano
    console.log(dCallout2.previousSibling.previousSibling) // <p>
  //Tercer...
    console.log(dCallout2.previousSibling.previousSibling.previousSibling) //#text

</script>
```

Las propiedades **firstChild** y **lastChild** también son interesantes. La propiedad **firstChild** es la misma que **childNodes[0]**, y **lastChild** es la misma que **childNodes[childNodes.length ? 1]**.

```
<script type="text/javascript">

//Accedemos al div con id="callout2"
  const divContent = document.getElementById('content');

//Elemento siguiente al Primer hijo, pues el primero está vacío
  console.log(divContent.firstChild.nextSibling.innerHTML);
    //Salida: Este es un archivo HTML muy <i>simple</i>
```

```
//Elemento anterior al Último hijo, pues el último está vacío
  console.log(divContent.lastChild.previousSibling.innerHTML);
    //Salida: <p>Un elemento HTML puede tener múltiples clases.</p>

</script>
```

Modos más precisos de acceder a los elementos de un documento

Utilizando las propiedades y los métodos **childNodes**, **firstChild**, **lastChild**, **nextSibling** y **previousSibling**, se puede navegar hacia arriba y hacia abajo por el árbol HTML y, así, hacer todo lo que nos interese en el documento. Sin embargo, el hecho de que los espacios son nodos de texto hace muy frágil este modo de trabajar con el DOM. Si la página cambia, nuestro *script* seguramente no trabaje correctamente. También, si nos interesa acceder al nodo más profundo del árbol, hará que el código sea un poco largo y tedioso. Por estos motivos, existen métodos más directos como pueden ser:

- **getElementsByTagName()** : Este método crea una colección con todos los elementos de la página que pertenecen a ese tipo (**p**, **a**, **table**, **div**,...).

- **getElementByClassName()** : Devuelve una lista de elementos que pertenecen a una clase determinada.

Veamos un ejemplo con nuestra página de trabajo:

```
<script type="text/javascript">
//Colección de los div del documento
  var cD = document.getElementsByTagName('div');

//Número de elementos div existentes
  console.log(cD.length);

//Recorremos toda la colección de elementos div
for(let i=0; i<cD.length; i++) {
    console.log(cD[i]);
}
console.log("-----------")

//Accedemos al contenido HTML del segundo div
console.log(cD[1].innerHTML);

//Accedemos al atributo class del segundo div
console.log(cD[1].className);
console.log("-----------")

//Si nos interesa seleccionar todos los elementos HTML de la página
let tElementos = document.getElementsByTagName('*');

//Número de elementos HTML en total
console.log(tElementos.length);
</script>
```

- **getElementsByName()** : Este método crea una colección con todos los elementos de la página que tienen la propiedad **name** configurada como nos interese buscar.

- **getElementById()** : Sin duda, es el modo más común de acceder **a un elemento.** Justamente se asignan IDs a los elementos para que sea sencillo acceder a ellos (debería ser un ID distinto por elemento). Veamos un ejemplo con nuestra página de trabajo:

```
<script type="text/javascript">
//Accedemos al div con id="content"
const dContent = document.getElementById('content');

//A partir de aquí,
//hacemos lo que nos interese con ese objeto seleccionado
//Por ejemplo, mostrar todos los nodos HTML existentes en su interior
for(let i=0; i<dContent.childNodes.length; i++) {
    console.log(dContent.childNodes[i]);
}
</script>
```

Decir que, recientemente, aparecieron nuevos métodos de acceso directo, como pueden ser:

- **querySelector()** : Este método encuentra elementos utilizando un selector CSS, pero sólo devuelve la primera coincidencia.
- **querySelectorAll()** : Hace lo mismo que el método anterior, pero devuelve una colección con todas las coincidencias.

Caminar por el DOM

La siguiente función nos permite "caminar por el DOM" desde un nodo dado:

```
<script type="text/javascript">
//Función caminarDOM(n)
function caminarDOM(n) {
do {
console.log(n);
if (n.hasChildNodes()) {
caminarDOM(n.firstChild);
}
} while ((n = n.nextSibling));
}

//Comenzamos a caminar en el <body>
caminarDOM(document.body);
</script>
```

Atributos

Como el primer hijo del elemento **<body>** es un espacio en blanco, el segundo hijo (elemento **[1]** de la colección) es el primer elemento real, luego hay otro espacio en blanco y, ya el elemento **[3]** de la colección es ** **:

```
> bd.childNodes[3];//<div id="content">...
```

Podemos chequear si un elemento tiene algún atributo, y si es así, podemos ver cuántos tiene:

```
> bd.childNodes[3].hasAttributes();//true
> bd.childNodes[3].attributes.length;//1
> bd.childNodes[3].attributes[0];//id="content"
> bd.childNodes[3].attributes['id'];//id="content"
> bd.childNodes[3].getAttribute('id');//"content"
```

Vemos que utilizamos:

- **hasAttributes()** : Método de los elementos HTML que devuelve un booleano indicando si tiene o no algún atributo. No requiere parámetros.
- **hasAttribute()** : Método de los elementos HTML que devuelve un booleano indicando si tiene un tipo de atributo en concreto pasado como parámetro.
- **attributes** : Propiedad que devuelve una lista de los atributos que tiene un elemento HTML.
- **getAttribute()** : Método de los elementos HTML que devuelve el valor configurado de un atributo determinado pasado como parámetro.

Accediendo al contenido del interior de un elemento

Para acceder al contenido de los elementos de una página HTML podemos utilizar las propiedades:

- **textContent** : Devuelve el texto existente en un nodo. Y también puede configurarlo.
- **innerHTML** : Devuelve o establece el valor *HTML* de un nodo.
- **nodeValue** : Devuelve o establece el "valor" del nodo en cuestión. Para el elemento *document* el valor es *null* y para los nodos de tipo texto devuelve el texto en sí.

Si seguimos con nuestro documento de trabajo, podemos utilizar el siguiente código de JavaScript para entender el funcionamiento de estas propiedades y métodos. En él leemos el texto existente en el párrafo:

```
<p>Este es un archivo HTML muy <i>simple</i></p>

<script type="text/javascript">

  //Seleccionamos el <body> del documento
    var bd = document.documentElement.childNodes[2]; //bd  = <body>
    console.log(bd);
  //Seleccionamos el <div> donde se encuentra el párrafo que nos interesa
    var dc = bd.childNodes[3]; //dc = <div id="content">
```

```
    console.log(dc);
  //Vemos todos los elementos dentro del <div>
    console.log(dc.childNodes.length); //11 elementos dentro del div
    for(i=0;i<dc.childNodes.length;i++) {
      console.log(dc.childNodes[i]);
    } //El <p> que nos interesa es el [1]
  //Seleccionamos ese <p>
    var p = dc.childNodes[1]; //p = <p>...
    console.log(p.nodeName); //"P"

//Utilizamos la propiedad textContent
  console.log(p.textContent); //"Este es un archivo HTML muy simple"
//Utilizamos la propiedad innerHTML
  console.log(p.innerHTML); //"Este es un archivo HTML muy <i>simple</i>"
//Utilizamos el método nodeValue de los nodos de p
  console.log(p.childNodes[0].nodeValue); //Este es un archivo HTML muy

</script>
```

Modificando los nodos DOM

Ahora ya nos podemos centrar en modificar el contenido de los nodos DOM, para ello, primeramente tendremos que seleccionar el nodo que nos interesa empleando cualquiera de los métodos vistos hasta ahora. Luego, utilizaremos las propiedades **innerHTML** o **nodeValue** para modificar su contenido. Podemos ver un nuevo ejemplo, que nos quedará de base para futuros puntos.

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8" />
<title>JavaScript</title>
</head>
<body>
<fieldset>
<legend>Jugar con el DOM</legend>
<label for="texto">Texto:</label>
<input type="text" id="texto" name="texto" />
<br /><br />
<input type="button" id="modificar" value="Modificar" />
<br /><br />
<label id="txtSal">Texto base</label>
</fieldset>

<script>
function modificar() {
//Seleccionamos el texto que escribimos en la caja con id='texto'
var t = document.getElementById("texto").value;
//Modificamos el contenido de la <label> con id='txtSal'
var etiqueta = document.getElementById("txtSal");
etiqueta.innerHTML = t;
}

//Agregamos el escuchador
document
.getElementById("modificar")
.addEventListener("click", modificar);
</script>
</body>
</html>
```

Modificando atributos

Para modificar atributos de un elemento determinado tenemos el método **setAttribute()** que establece el valor de un atributo en el elemento indicado. Si el atributo ya existe, el valor es actualizado, en caso contrario, el nuevo atributo es añadido con el nombre y valor indicado.

Para obtener el valor actual de un atributo, se utiliza **getAttribute()** y para eliminar un atributo, se llama a **removeAttribute()**.

Podemos ver el siguiente ejemplo donde se añade el atributo *class* con el valor de **clase1** al primer elemento **h1** del documento:

```
  document.getElementsByTagName("H1")[0].setAttribute("class", "clase1");
```

Se podría modificar cualquier atributo de un elemento en cuestión, como vemos, incluso los referentes a los estilos. Así y todo, en el siguiente punto veremos como modificar los estilos con métodos específicos para ello.

Modificando estilos

Además del contenido, es interesante también modificar los estilos de los elementos. Veamos algunas técnicas y, a continuación, los aplicaremos en el siguiente ejemplo utilizando de base el código anterior:

- Los elementos tienen una propiedad asociada a los estilos.
- Las propiedades CSS a menudo tienen guiones, pero los guiones no son aceptables en los identificadores de JavaScript. En tales casos, omite el guión y escribe en mayúscula la siguiente letra. Entonces, **padding-top** se convierte en **paddingTop**, **margin-left** se convierte en **marginLeft**, y así sucesivamente.
- También podemos leer el estilo aplicado en el nodo seleccionado.

Así, el código ejemplo queda del siguiente modo:

```
function modificar() {
//Seleccionamos el texto que escribimos en la caja con id='texto'
var t = document.getElementById("texto").value;

//Modificamos el contenido de la <label> con id='txtSal'
var etiqueta = document.getElementById("txtSal");
etiqueta.innerHTML = t;

//Modificamos los estilos
//Dibujar un borde a la etiqueta
etiqueta.style.border = "1px solid red";
//Poner en negrita el texto
etiqueta.style.fontWeight = "bold";

//También podemos leer el estilo actualmente aplicado
console.log(etiqueta.style.cssText);
}
```

También podemos modificar el código para añadir o modificar el atributo **class** del elemento seleccionado. Para acceder a las clases configuradas en un elemento y poder modificarlas tenemos las siguientes herramientas:

- La propiedad de solo lectura **classList** : Que es una propiedad que devuelve una lista con las clases definidas en el nodo seleccionado. Lógicamente, si el atributo clase no está definido o está vacío, **classList.length** devuelve 0.
- Como **classList** es de sólo lectura, esta propiedad tiene los siguientes métodos para modificar las clases configuradas en un elemento:

    ◊ **add(String [, String])** : Añade las clases indicadas. Si estas clases existieran en el atributo del elemento serán ignoradas.
    ◊ **remove(String [, String])** : Elimina las clases indicadas.
    ◊ **toggle(String [, condition])** : Alternar el valor de la clase indicada.
    ◊ **contains()** : Comprueba si la clase indicada existe en el atributo de clase del elemento.
    ◊ **replace(oldClass, newClass)** : Reemplazar el valor de la clase indicada por la otra.

- También existe la propiedad **className** : Obtiene y establece el valor del atributo **class** del elemento especificado.

Así quedaría el ejemplo visto antes:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8" />
<style type="text/css">
.labelSal {
border-style: dashed solid;
border-color: red;
font-weight: bold;
}
</style>
<title>JavaScript</title>
</head>
<body>
<fieldset>
<legend>Jugar con el DOM</legend>
```

```
<label for="texto">Texto:</label>
<input type="text" id="texto" name="texto" />
<br /><br />
<input
type="button"
id="divModificar"
name="Modificar"
value="Modificar"
/>
<br /><br />
<label id="txtSal">Texto base</label>
</fieldset>

<script>
function modificar() {
//Seleccionamos el texto que escribimos en la caja con id='texto'
  const t = document.getElementById("texto").value;
//Modificamos el contenido de la <label> con id='txtSal'
  let etiqueta = document.getElementById("txtSal");
  etiqueta.innerHTML = t;
//Modificamos los estilos agregando una clase a la <label>
  etiqueta.classList.add("labelSal");
}

//Agregamos el escuchador
document
.getElementById("divModificar")
.addEventListener("click", modificar);
</script>
</body>
</html>
```

Creando nuevos nodos

Para crear nuevos nodos, podemos emplear los métodos:

- **createElement(tagName)** : Crea un elemento HTML especificado por su **tagName**.
- **createTextNode(data)** : Crea un nuevo nodo de texto con contenido *data*.
- **[https://developer.mozilla.org/es/docs/Web/API/Document/createDocumentFragment createDocumentFragment()** : Se crea un objeto *DocumentFragment* vacío, el cual queda listo para que pueda insertársele nodos en el.

Una vez creados, podemos añadirlos al árbol DOM empleando uno de los métodos siguientes:

- **appendChild()** : Inserta un nuevo nodo dentro del siguiente modo **nombreNodoPadre.appendChild(nombreNodoHijo)**.
- **insertBefore()** : Inserta un nodo antes del nodo de referencia como hijo de un nodo padre indicado. Tendría la siguiente sintaxis: **nombreNodoPadre.insertBefore(nombreNuevoNodo, nodoReferenciaDelanteDelQueInsertamos)**;
- **after()** : Inserta uno o varios nodos después del nodo de referencia como hijo de un nodo padre indicado.
- **replaceChild()** : Reemplaza un nodo hijo por otro del elemento padre especificado.

Escribamos el siguiente código para probar estos métodos:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>My página</title>
</head>
<body>
<p class="primero">Primer párrafo</p>
<p><em>Segundo</em> párrafo</p>
<p id="cierre">Final</p>

</body>
</html>
```

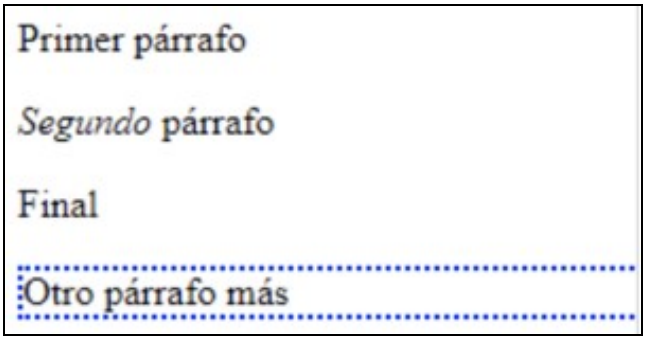
En la consola podemos probar las siguientes líneas de código:

```
//1. Crear un nuevo elemento p y modificar su propiedad innerHTML:
    > var miP = document.createElement('p');
```

```
    > miP.innerText = 'Otro párrafo más';
    //¡Ojo! NO va a aparecer en el Navegador hasta que lo añadamos a 'document'

//2. El nuevo elemento, automáticamente, tiene ya configuradas todas sus propiedades, en concreto style. Además de verlas podemos mo
    > miP.style;
        CSSStyleDeclaration {alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: "", ?}
    > miP.style.border = '2px dotted blue';

//3. Empleando el método appendChild() podemos agregar el párrafo al documento:
    > document.body.appendChild(miP);
```



Creando nuevos nodos
Utilizando sólo métodos DOM

La propiedad **innerHTML** nos permite hacer las cosas más fácilmente que utilizando sólo métodos DOM. Podemos verlo con el siguiente ejemplo, donde se quiere añadir, al final del elemento **body**, el siguiente párrafo:

```
<p>Un párrafo más<strong> resaltado</strong></p>
```

Con la propiedad **innerHTML** el método sería exactamente igual al del ejemplo anterior (cambiando **innerHTML** por **innerText**).

Utilizando únicamente métodos DOM, el código quedaría así:

```
//1. Crear p
  > var miP = document.createElement('p');
  //undefined

//2. Crear el nodo texto y agregárselo a p
  > var miT = document.createTextNode('Un párrafo más ');
  //undefined
  > myp.appendChild(miT);
  //#text "Un párrafo más"

//3. Crear el elemento strong y agregarle un nodo texto a él
  > var strong = document.createElement('strong');
  //undefined
  > strong.appendChild(document.createTextNode(' resaltado'));
  //#text "resaltado"

//4. Agregar strong a p
  > miP.appendChild(strong);
  //<strong>

//5. Agregar p al final del body
  > document.body.appendChild(miP);
  //<p>
```

Trabajando con el método insertBefore()

Como ya se comentó antes, utilizando **appendChild()**, tú puedes solo añadir un nuevo hijo al final del elemento seleccionado. Para más control sobre la localización exacta, existe el método **insertBefore()**. Este hace lo mismo que **appendChild()**, pero acepta un parámetro extra especificando dónde (antes de qué elemento) se va a insertar el nuevo nodo.

Por ejemplo, el siguiente código inserta un nodo texto al final del elemento **body**:

```
> document.body.appendChild(document.createTextNode('¡hola!'));
```

Sin embargo, este código crea otro nodo texto y lo añade como primer hijo del elemento **body**:

```
> document.body.insertBefore(document.createTextNode('¡Primer saludo!'), document.body.firstChild);
```

Siguiendo con el ejemplo anterior, podríamos insertar el nuevo párrafo antes del id='cierre' luego de haberlo insertado al final del body:

```
> let idCierre = document.getElementById('cierre');
> document.body.insertBefore(miP, idCierre);
  //<p>
  //Vemos que el párrafo 'miP' cambia de lugar...
```

Emplear el método cloneNode()

Otro modo de crear nodos es ?clonando? nodos ya existentes. El método **cloneNode()** acepta un parámetro *booleano*:

- **true** = copia el nodo y todos sus hijos.
- **false** = copia únicamente el nodo.

Así, podemos ver un ejemplo siguiendo con el anterior:

```
//Hacemos un clon de 'miP' con todo su interior
  > let miPtrue = miP.cloneNode(true);
  //undefined
  > document.body.appendChild(miPtrue);
  //<p>
//Vemos que aparece al final del 'body' sin desaparecer 'miP' de la web

//Ahora hacemos otro clon de 'miP' pero sin clonar sus hijos
  > let miPfalse = miP.cloneNode(false);
  //undefined
  > document.body.appendChild(miPfalse);
  //<p>
//Aparentemente no se agrega nada, pero si inspeccionamos el HTML
//vemos que se añade a la web un nuevo <p> sin nada en su interior
```

Eliminando nodos

Para eliminar nodos del árbol DOM, podemos emplear el método **removeChild()**.

Así y todo, ya se comentó antes que también existe el el método **replaceChild()** que elimina un nodo y pone otro en su lugar.

Por otro lado, eliminar elementos utilizando la propiedad **innerHTML** es mucho más rápido y fácil en la mayoría de los casos.

Como ejemplo, aquí tenemos una pequeña función que elimina, utilizando métodos DOM, todos los nodos desde uno dado como parámetro:

```
function removeAll(n) {
  while (n.firstChild) {
    n.removeChild(n.firstChild);
  }
}
```

Para borrar todo:

```
> removeAll(document.body);
```

Volver