UD03 - Document Object Model (DOM) y Browser Object Model (BOM)

1. Document Object Model (DOM)

1.1 Introducción

La mayoría de las veces que programamos con Javascript es para que se ejecute en una página web mostrada por el navegador. En este contexto tenemos acceso a ciertos objetos que nos permiten interactuar con la página (DOM) y con el navegador (Browser Object Model, BOM).

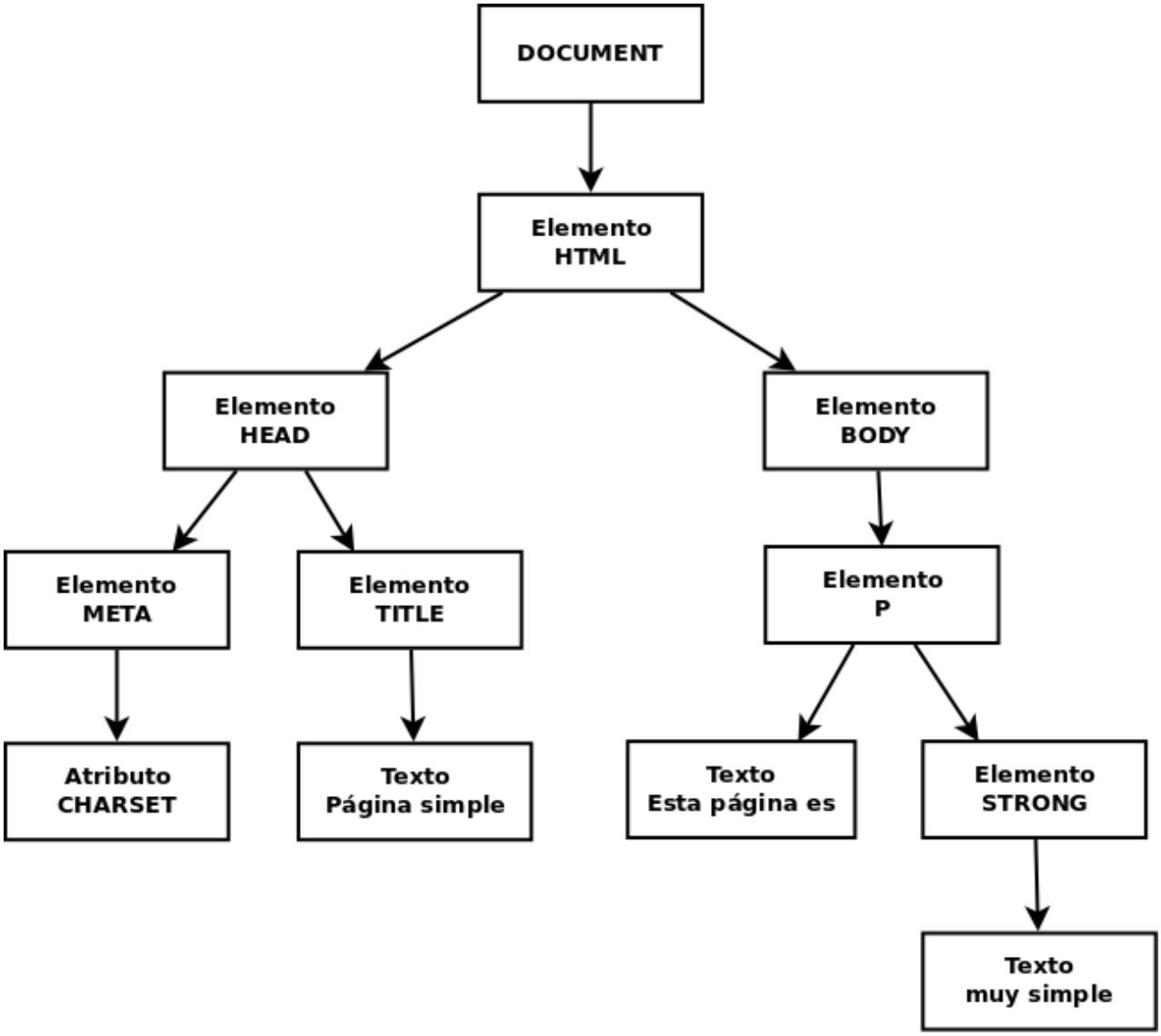
El **DOM** es una estructura en árbol que representa todos los elementos HTML de la página y sus atributos. Todo lo que contiene la página se representa como nodos del árbol y mediante el DOM podemos acceder a cada nodo, modificarlo, eliminarlo o añadir nuevos nodos de forma que cambiamos dinámicamente la página mostrada al usuario.

La raíz del árbol DOM es **document** y de este nodo cuelgan el resto de elementos HTML. Cada uno constituye su propio nodo y tiene subnodos con sus *atributos*, *estilos* y elementos HTML que contiene.

Por ejemplo, la página HTML:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Página simple</title>
</head>
<body>
  <p>Esta página es <strong>muy simple</strong></p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



Cada etiqueta HTML suele originar 2 nodos:

- Element: correspondiente a la etiqueta
- Text: correspondiente a su contenido (lo que hay entre la etiqueta y su par de cierre)

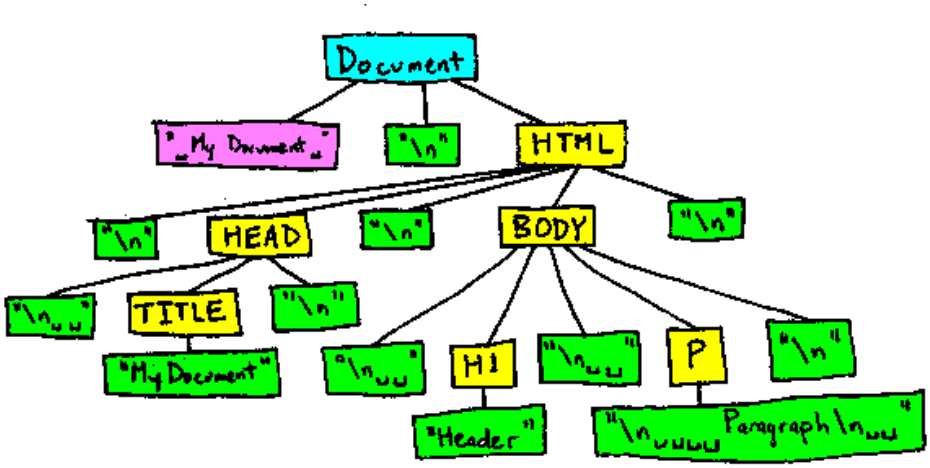
Cada nodo es un objeto con sus propiedades y métodos.

El ejemplo anterior está simplificado porque sólo aparecen los nodos de tipo ***elemento*** pero en realidad también generan nodos los saltos de línea, tabuladores, espacios, comentarios, etc. En el siguiente ejemplo podemos ver TODOS los nodos que realmente se generan. La página:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Document</title>
</head>
<body>
  <h1>Header</h1>
  <p>
    Paragraph
  </p>
</body>
</html>

```

se convierte en el siguiente árbol DOM:



1.2 Acceso a los nodos

Los principales métodos para acceder a los diferentes nodos son:

- **.getElementById(id)**: devuelve el nodo con la *id* pasada. Ej.:

```
let nodo = document.getElementById('main');   // nodo contendrá el nodo cuya id es _main_
```

- **.getElementsByClassName(clase)**: devuelve una colección (similar a un array) con todos los nodos de la *clase* indicada. Ej.:

```
let nodos = document.getElementsByClassName('error');   // nodos contendrá todos los nodos cuya clase es _error_
```

NOTA: las colecciones son similares a arrays (se accede a sus elementos con *[indice]*) pero no se les pueden aplicar sus métodos *filter*, *map*, ... a menos que se conviertan a arrays con *Array.from()*

- **.getElementsByTagName(etiqueta)**: devuelve una colección con todos los nodos de la *etiqueta* HTML indicada. Ej.:

```
let nodos = document.getElementsByTagName('p');   // nodos contendrá todos los nodos de tipo  _<p>_
```

- **.querySelector(selector)**: devuelve el primer nodo seleccionado por el *selector* CSS indicado. Ej.:

```
let nodo = document.querySelector('p.error');   // nodo contendrá el primer párrafo de clase _error_
```

- **.querySelectorAll(selector)**: devuelve una colección con todos los nodos seleccionados por el *selector* CSS indicado. Ej.:

```
let nodos = document.querySelectorAll('p.error');   // nodos contendrá todos los párrafos de clase _error_
```

NOTA: al aplicar estos métodos sobre *document* se seleccionará sobre la página pero podrían también aplicarse a cualquier nodo y en ese caso la búsqueda se realizaría sólo entre los descendientes de dicho nodo.

También tenemos 'atajos' para obtener algunos elementos comunes:

- `document.documentElement`: devuelve el nodo del elemento <*html*>
- `document.head`: devuelve el nodo del elemento <*head*>
- `document.body`: devuelve el nodo del elemento <*body*>
- `document.title`: devuelve el nodo del elemento <*title*>
- `document.link`: devuelve una colección con todos los hiperenlaces del documento
- `document.anchor`: devuelve una colección con todas las anclas del documento
- `document.forms`: devuelve una colección con todos los formularios del documento
- `document.images`: devuelve una colección con todas las imágenes del documento
- `document.scripts`: devuelve una colección con todos los scripts del documento

PRUEBA 01_UD3: Para hacer los ejercicios de este tema descárgate esta página de ejemplo y ábrela en tu navegador. Obtén por consola, al menos de 2 formas diferentes:

- El elemento con id 'input2'
- La colección de párrafos
- Lo mismo pero sólo de los párrafos que hay dentro del div 'lipsum'

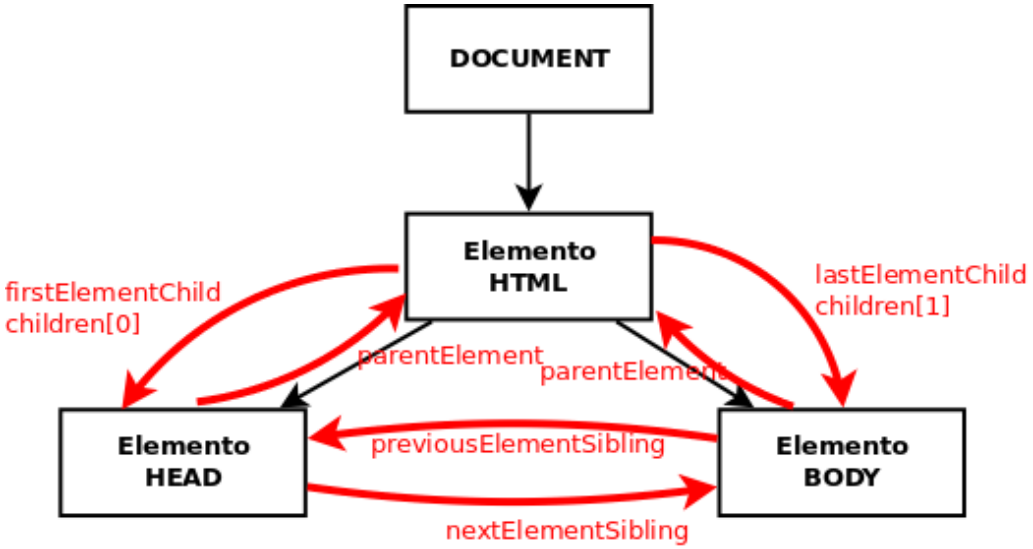
- El formulario (ojo, no la colección con el formulario sino sólo el formulario)
- Todos los inputs
- Sólo los inputs con nombre 'sexo'
- Los items de lista de la clase 'important' (sólo los LI)

1.3 Acceso a nodos a partir de otros

En muchas ocasiones queremos acceder a cierto nodo a partir de uno dado. Para ello tenemos los siguientes métodos que se aplican sobre un elemento del árbol DOM:

- `elemento.parentElement`: devuelve el elemento padre de *elemento*
- `elemento.children`: devuelve la colección con todos los elementos hijo de *elemento* (sólo elementos HTML, no comentarios ni nodos de tipo texto)
- `elemento.childNodes`: devuelve la colección con todos los hijos de *elemento*, incluyendo comentarios y nodos de tipo texto por lo que no suele utilizarse
- `elemento.firstElementChild`: devuelve el elemento HTML que es el primer hijo de *elemento*
- `elemento.firstChild`: devuelve el nodo que es el primer hijo de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.lastElementChild`, `elemento.lastChild`: igual pero con el último hijo
- `elemento.nextElementSibling`: devuelve el elemento HTML que es el siguiente hermano de *elemento*
- `elemento.nextSibling`: devuelve el nodo que es el siguiente hermano de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.previousElementSibling`, `elemento.previousSibling`: igual pero con el hermano anterior
- `elemento.hasChildNodes`: indica si *elemento* tiene o no nodos hijos
- `elemento.childElementCount`: devuelve el nº de nodos hijo de *elemento*

IMPORTANTE: a menos que me interesen comentarios, saltos de página, etc **siempre** debo usar los métodos que sólo devuelven elementos HTML, no todos los nodos.



PRUEBA 02_UD3: Siguiendo con la página de ejemplo obtén desde la consola, al menos de 2 formas diferentes:

- El primér párrafo que hay dentro del div 'lipsum'
- El segundo párrafo de 'lipsum'
- El último item de la lista
- La label de 'Escoge sexo'

1.4 Propiedades de un nodo

Las principales propiedades de un nodo son:

- `elemento.innerHTML`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, incluyendo otras etiquetas HTML. Por ejemplo si *elemento* es el nodo `<p>Esta página es <strong>muy simple</strong></p>`

```
let contenido = elemento.innerHTML;   // contenido='Esta página es <strong>muy simple</strong>'
```

- `elemento.textContent`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, pero ignorando otras etiquetas HTML. Siguiendo con el ejemplo anterior:

```
let contenido = elemento.textContent;   // contenido='Esta página es muy simple'
```

- `elemento.value`: devuelve la propiedad 'value' de un <input> (en el caso de un <input> de tipo text devuelve lo que hay escrito en él). Como los <inputs> no tienen etiqueta de cierre (</input>) no podemos usar *.innerHTML* ni *.textContent*. Por ejemplo si *elem1* es el nodo `<input name="nombre">` y *elem2* es el nodo `<input tipe="radio" value="H">Hombre`

```
let cont1 = elem1.value;    // cont1 valdría lo que haya escrito en el <input> en ese momento
let cont2 = elem2.value;    // cont2="H"
```

Otras propiedades:

- `elemento.innerText`: igual que *textContent*
- `elemento.focus`: da el foco a *elemento* (para inputs, etc). Para quitarle el foco `elemento.blur`
- `elemento.clientHeight` / `elemento.clientWidth`: devuelve el alto / ancho visible del *elemento*
- `elemento.offsetHeight` / `elemento.offsetWidth`: devuelve el alto / ancho total del *elemento*
- `elemento.clientLeft` / `elemento.clientTop`: devuelve la distancia de *elemento* al borde izquierdo / superior
- `elemento.offsetLeft` / `elemento.offsetTop`: devuelve los píxels que hemos desplazado *elemento* a la izquierda / abajo

PRUEBA 03_UD3: Obtén desde la consola, al menos de 2 formas:

- El innerHTML de la etiqueta de 'Escoge sexo'
- El textContent de esa etiqueta
- El valor del primer input de sexo
- El valor del sexo que esté seleccionado (difícil, búscalo por Internet)

1.5 Manipular el árbol DOM

Vamos a ver qué métodos nos permiten cambiar el árbol DOM, y por tanto modificar la página:

- `document.createElement('etiqueta')`: crea un nuevo elemento HTML con la etiqueta indicada, pero aún no se añade a la página. Ej.:

```
let nuevoLi = document.createElement('li');
```

- `document.createTextNode('texto')`: crea un nuevo nodo de texto con el texto indicado, que luego tendremos que añadir a un nodo HTML. Ej.:

```
let textoLi = document.createTextNode('Nuevo elemento de lista');
```

- `elemento.appendChild(nuevoNodo)`: añade *nuevoNodo* como último hijo de *elemento*. Ahora ya se ha añadido a la página. Ej.:

```
nuevoLi.appendChild(textoLi);     // añade el texto creado al elemento LI creado
let miPrimeraLista = document.getElementsByTagName('ul')[0];  // selecciona el 1º UL de la página
miPrimeraLista.appendChild(nuevoLi);    // añade LI como último hijo de UL, es decir al final de la lista
```

- `elemento.insertBefore(nuevoNodo, nodo)`: añade *nuevoNodo* como hijo de *elemento* antes del hijo *nodo*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0];
 // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0];
  // selecciona el 1º LI de miPrimeraLista
miPrimeraLista.insertBefore(nuevoLi, primerElementoDeLista);
 // añade LI al principio de la lista
```

- `elemento.removeChild(nodo)`: borra *nodo* de *elemento* y por tanto se elimina de la página. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0];  // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0];
 // selecciona el 1º LI de miPrimeraLista
miPrimeraLista.removeChild(primerElementoDeLista);    // borra el primer elemento de la lista
// También podríamos haberlo borrado sin tener el padre con:
primerElementoDeLista.parentElement.removeChild(primerElementoDeLista);
```

- `elemento.replaceChild(nuevoNodo, viejoNodo)`: reemplaza *viejoNodo* con *nuevoNodo* como hijo de *elemento*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0];  // selecciona el 1º UL de la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0];
 // selecciona el 1º LI de miPrimeraLista
miPrimeraLista.replaceChild(nuevoLi, primerElementoDeLista);    // reemplaza el 1º elemento de la lista con nuevoLi
```

- `elementoAClonar.cloneNode(boolean)`: devuelve un clon de *elementoAClonar* o de *elementoAClonar* con todos sus descendientes según le pasemos como parámetro *false* o *true*. Luego podremos insertarlo donde queramos.

OJO: Si añado con el método `appendChild` un nodo que estaba en otro sitio **se elimina de donde estaba** para añadirse a su nueva posición. Si quiero que esté en los 2 sitios deberé clonar el nodo y luego añadir el clon y no el nodo original.

Para la creación de nuevos nodos, podemos utilizar la propiedad **innerHTML**, siendo el código a usar mucho más simple.

```
let ultimoParrafo = document.createElement('p');
ultimoParrafo.innerHTML = 'Párrafo añadido al final';
miDiv.appendChild(ultimoParrafo);
```

OJO: También se puede añadir un párrafo de la siguiente forma: con la siguiente:

```
miDiv.innerHTML+='<p>Párrafo añadido al final</p>';
```

Aunque es válida, no es muy eficiente ya que obliga al navegador a volver a pintar TODO el contenido de miDIV. La forma correcta de hacerlo sería:

```
let ultimoParrafo = document.createElement('p');
ultimoParrafo.innerHTML = 'Párrafo añadido al final';
miDiv.appendChild(ultimoParrafo);
```

Así sólo debe repintar el párrafo añadido, conservando todo lo demás que tenga *miDiv*.

Podemos ver más ejemplos de creación y eliminación de nodos en W3Schools.

EJERCICIO 01_UD3: Añade a la página:

- Un nuevo párrafo al final del DIV '*lipsum*' con el texto "Nuevo párrafo **añadido** por javascript" (fíjate que una palabra está en negrita)
- Un nuevo elemento al formulario tras el '*Dato 1*' con la etiqueta '*Dato 1 bis*' y el INPUT con id '*input1bis*' que al cargar la página tendrá escrito "Hola"

1.5.1 Modificar el DOM con ChildNode

Childnode es una interfaz que permite manipular del DOM de forma más sencilla pero no está soportada en los navegadores Safari de IOS. Incluye los métodos:

- `elemento.before(nuevoNodo)`: añade el *nuevoNodo* pasado antes del nodo *elemento*
- `elemento.after(nuevoNodo)`: añade el *nuevoNodo* pasado después del nodo *elemento*
- `elemento.replaceWith(nuevoNodo)`: reemplaza el nodo *elemento* con el *nuevoNodo* pasado
- `elemento.remove()`: elimina el nodo *elemento*

1.6 Atributos de los nodos

Podemos ver y modificar los valores de los atributos de cada elemento HTML y también añadir o eliminar atributos:

- `elemento.attributes`: devuelve un array con todos los atributos de *elemento*
- `elemento.hasAttribute('nombreAtributo')`: indica si *elemento* tiene o no definido el atributo *nombreAtributo*
- `elemento.getAttribute('nombreAtributo')`: devuelve el valor del atributo *nombreAtributo* de *elemento*. Para muchos elementos este valor puede directamente con `elemento.atributo`.
- `elemento.setAttribute('nombreAtributo', 'valor')`: establece *valor* como nuevo valor del atributo *nombreAtributo* de *elemento*. También puede cambiarse el valor directamente con `elemento.atributo=valor`.
- `elemento.removeAttribute('nombreAtributo')`: elimina el atributo *nombreAtributo* de *elemento*

A algunos atributos comunes como `id`, `title` o `className` (para el atributo **class**) se puede acceder y cambiar como si fueran una propiedad del elemento (`elemento.atributo`). Ejemplos:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0];  // selecciona el 1º UL de la página
miPrimeraLista.id = 'primera-lista';
// es equivalente ha hacer:
miPrimeraLista.setAttribute('id', 'primera-lista');
```

1.6.1 Estilos de los nodos

Los estilos están accesibles como el atributo **style**. Cualquier estilo es una propiedad de dicho atributo pero con la sintaxis *camelCase* en vez de *kebab-case*. Por ejemplo para cambiar el color de fondo (propiedad background-color) y ponerle el color *rojo* al elemento *miPrimeraLista* haremos:

```
miPrimeraLista.style.backgroundColor = 'red';
```

De todas formas normalmente **NO CAMBIAREMOS ESTILOS** a los elementos sino que les pondremos o quitaremos clases que harán que se le apliquen o no los estilos definidos para ellas en el CSS.

1.6.2 Atributos de clase

Ya sabemos que el aspecto de la página debe configurarse en el CSS por lo que no debemos aplicar atributos *style* al HTML. En lugar de ello les ponemos clases a los elementos que harán que se les aplique el estilo definido para dicha clase.

Como es algo muy común en lugar de utilizar las instrucciones de `elemento.setAttribute('className', 'destacado')` o directamente `elemento.className='destacado'` podemos usar la propiedad ***classList*** que devuelve la colección de todas las clases que tiene el elemento. Por ejemplo si *elemento* es `<p class="destacado direccion">...`:

```
let clases=elemento.classList;   // clases=['destacado', 'direccion'], OJO es una colección, no un Array
```

Además dispone de los métodos:

- **.add(clase)**: añade al elemento la clase pasada (si ya la tiene no hace nada). Ej.:

```
elemento.classList.add('primero');   // ahora elemento será <p class="destacado direccion primero">...
```

- **.remove(clase)**: elimina del elemento la clase pasada (si no la tiene no hace nada). Ej.:

```
elemento.classList.remove('direccion');   // ahora elemento será <p class="destacado primero">...
```

- **.toogle(clase)**: añade la clase pasada si no la tiene o la elimina si la tiene ya. Ej.:

```
elemento.classList.toogle('destacado');   // ahora elemento será <p class="primero">...
elemento.classList.toogle('direccion');   // ahora elemento será <p class="primero direccion">...
```

- **.contains(clase)**: dice si el elemento tiene o no la clase pasada. Ej.:

```
elemento.classList.contains('direccion');   // devuelve true
```

- **.replace(oldClase, newClase)**: reemplaza del elemento una clase existente por una nueva. Ej.:

```
elemento.classList.replace('primero', 'ultimo');   // ahora elemento será <p class="ultimo direccion">...
```

Tened en cuenta que NO todos los navegadores soportan *classList* por lo que si queremos añadir o quitar clases en navegadores que no lo soportan debemos hacerlo con los métodos estándar, por ejemplo para añadir la clase 'rojo':

```
let clases = elemento.className.split(" ");
if (clases.indexOf('rojo') == -1) {
  elemento.className += ' ' + 'rojo';
}
```

2. Browser Object Model (BOM)

2.1 Introducción

Si en el apartado anterior vimos cómo interactuar con la página (DOM) en este veremos cómo acceder a objetos que nos permitan interactuar con el navegador (y Browser Object Model, BOM).

Usando los objetos BOM podemos:

- Abrir, cambiar y cerrar ventanas
- Ejecutar código en cierto tiempo (*timers*)
- Obtener información del navegador
- Ver y modificar propiedades de la pantalla
- Gestionar cookies, ...

2.2 Timers

Permiten ejecutar código en el futuro (cuando transcurran los milisegundos indicados). Hay 2 tipos:

- `setTimeout(función, milisegundos)`: ejecuta la función indicada una sóla vez, cuando transcurran los milisegundos
- `setInterval(función, milisegundos)`: ejecuta la función indicada cada vez que transcurran los milisegundos, hasta que sea cancelado el *timer*. A ambas se le pueden pasar más parámetros tras los milisegundos y serán los parámetros que recibirá la función a ejecutar.

Ambas funciones devuelven un identificador que nos permitirá cancelar la ejecución del código, con:

- `clearTiemout(identificador)`
- `clearInterval(identificador)`

Ejemplo:

```
let idTimeout=setTimeout(function() {
  alert('Timeout que se ejecuta al cabo de 1 seg.')
}, 1000);

let i=1;
let idInterval=setInterval(function() {
  alert('Interval cada 3 seg. Ejecución nº: '+ i++);
   if (i==5) {
      clearInterval(idInterval);
      alert('Fin de la ejecución del Interval');
  }
}, 3000);
```

PRUEBA 04_UD3: Ejecuta en la consola cada una de esas funciones

2.3 Objetos del BOM

Al contrario que para el DOM, no existe un estándar de BOM pero es bastante parecido en los diferentes navegadores.

2.3.1 Objeto window

Representa la ventana del navegador y es el objeto principal. De hecho puede omitirse al llamar a sus propiedades y métodos, por ejemplo, el método `setTimeout()` es en realidad `window.setTimeout()`.

Sus principales propiedades y métodos son:

- `.name`: nombre de la ventana actual
- `.status`: valor de la barra de estado
- `.screenX`/`.screenY`: distancia de la ventana a la esquina izquierda/superior de la pantalla
- `.outerWidth`/`.outerHeight`: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- `.innerWidth`/`.innerHeight`: ancho/alto útil del documento, sin la toolbar y la scrollbar
- `.open(url, nombre, opciones)`: abre una nueva ventana. Devuelve el nuevo objeto ventana. Las principales opciones son:
  - `.toolbar`: si tendrá barra de herramientas
  - `.location`: si tendrá barra de dirección
  - `.directories`: si tendrá botones Adelante/Atrás

    - `.status`: si tendrá barra de estado
    - `.menubar`: si tendrá barra de menú
    - `.scrollbar`: si tendrá barras de desplazamiento
    - `.resizable`: si se puede cambiar su tamaño
    - `.width=px`/`.height=px`: ancho/alto
    - `.left=px`/`.top=px`: posición izq/sup de la ventana
- `.opener`: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con *open*)
- `.close()`: la cierra (pide confirmación, a menos que la hayamos abierto con open)
- `.moveTo(x,y)`: la mueve a las coord indicadas
- `.moveBy(x,y)`: la desplaza los px indicados
- `.resizeTo(x,y)`: la da el ancho y alto indicados
- `.resizeBy(x,y)`: le añade ese ancho/alto
- `.pageXoffset / pageYoffset`: scroll actual de la ventana horizontal / vertical
- Otros métodos: `.back()`, `.forward()`, `.home()`, `.stop()`, `.focus()`, `.blur()`, `.find()`, `.print()`, ...
  NOTA: por seguridad no se puede mover una ventana fuera de la pantalla ni darle un tamaño menor de 100x100 px ni tampoco se puede mover una ventana no abierta con .open() o si tiene varias pestañas

PRUEBA 05_UD3: Ejecuta desde la consola:

- abre una nueva ventana de dimensiones 500x200px en la posición (100,200)
- escribe en ella (con document.write) un título h1 que diga 'Hola'
- muévela 300 px hacia abajo y 100 a la izquierda
- ciérrala

Puedes ver un ejemplo de cómo abrir ventanas en este vídeo.

EJERCICIO 02_UD3: Haz que a los 2 segundos de abrir la página se abra un *popup* con un mensaje de bienvenida. Esta ventana tendrá en su interior un botón Cerrar que permitirá que el usuario la cierre haciendo clic en él. Tendrá el tamaño justo para visualizar el mensaje y no tendrá barras de scroll, ni de herramientas, ni de dirección... únicamente el mensaje.

2.3.1.1 Diálogos

Hay 3 métodos del objeto *window* que ya conocemos y que nos permiten abrir ventanas de diálogo con el usuario:

- `window.alert(mensaje)`: muestra un diálogo con el mensaje indicado y un botón de 'Aceptar'
- `window.confirm(mensaje)`: muestra un diálogo con el mensaje indicado y botones de 'Aceptar' y 'Cancelar'. Devuelve *true* si se ha pulsado el botón de aceptar del diálogo y *false* si no.
- `window.prompt(mensaje [, valor predeterminado])`: muestra un diálogo con el mensaje indicado, un cuadro de texto (vacío o co el valor predeterminado indicado) y botones de 'Aceptar' y 'Cancelar'. Si se pulsa 'Aceptar' devolverá un *string* con el valor que haya en el cuadro de texto y si se pulsa 'Cancelar' o se cierra devolverá *null*.

2.3.2 Objeto location

Contiene información sobre la URL actual del navegador y podemos modificarla. Sus principales propiedades y métodos son:

- `.href`: devuelve la URL actual completa
- `.protocol`, `.host`, `.port`: devuelve el protocolo, host y puerto respectivamente de la URL actual
- `.pathname`: devuelve la ruta al recurso actual
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

PRUEBA 06_UD3: Ejecuta en la consola

- muestra la ruta completa de la página actual
- muestra el servidor de esta página
- carga la página de Google usando el objeto *location*

2.3.3 Objeto history

Permite acceder al historial de páginas visitadas y navegar por él:

- `.length`: muestra el número de páginas almacenadas en el historial
- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página

- `.go(num)`: se mueve *num* páginas hacia adelante o hacia atrás (si *num* es negativo) en el historial

> **PRUEBA 07_UD3**: desde la consola vuelve a la página anterior

2.3.4 Otros objetos

Los otros objetos que incluye BOM son:

- document: el objeto *document* que hemos visto en el DOM
- navigator: nos informa sobre el navegador y el sistema en que se ejecuta
  - `.userAgent`: muestra información sobre el navegador que usamos
  - `.plataform`: muestra información sobre la plataforma sobre la que se ejecuta
  - ...
- screen: nos da información sobre la pantalla
  - `.width`/`.height`: ancho/alto total de la pantalla (resolución)
  - `.availWidth`/`.availHeight`: igual pero excluyendo la barra del S.O.
  - ...

> **PRUEBA 08_UD3**: obtén desde la consola todas las propiedades width/height y availWidth/availHeight del objeto *screen*. Compáralas con las propiedades innerWidth/innerHeight y outerWidth/outerHeight de *window*.

3. El patrón Modelo-Vista-Controlador

Modelo-vista-controlador (MVC) es el patrón de arquitectura de software más utilizado en la actualidad en desarrollo web (y también en muchas aplicaciones de escritorio). Este patrón propone separar la aplicación en tres **componentes** distintos: el **modelo**, la **vista** y el **controlador**:

- El **modelo** es el conjunto de todos los datos o información con la que trabaja la aplicación. Normalmente serán variables extraidas de una base de datos y el modelo gestiona los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Normalmente el modelo no tiene conocimiento de las otras partes de la aplicación.
- La **vista** muestra al usuario el modelo (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario). Es la intermediaria entre la aplicación y el usuario

- El **controlador** es el encargado de coordinar el funcionamiento de la aplicación. Responde a los eventos del usuario para lo que hace peticiones al modelo (para obtener o cambiar la información) y a la vista (para que muestre al usuario dicha información).

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

3.1 Una aplicación sin MVC

Si una aplicación no utiliza este modelo cuando una función modifica los datos debe además reflejar dicha modificación en la página para que la vea el usuario. Por ejemplo vamos a hacer una aplicación para gestionar un almacén. Entre otras muchas cosas tendrá una función (podemos llamarle *addProduct*) que se encargue de añadir un nuevo producto al almacén y dicha función deberá realizar:

- añadir el nuevo producto al almacén (por ejemplo añadiéndolo a un array de productos)
- pintar en la página ese nuevo producto (por ejemplo añadiendo una nueva línea a una tabla donde se muestran los productos)

Es posible que en algún momento decidamos cambiar la forma en que se muestra la información al usuario para lo que deberemos modificar la función *addProduct* y si cometemos algún error podría hacer que no se añadan correctamente los productos al almacén. Además va a ser una función muy grande y va a ser difícil mantener ese código.

3.2 Nuestro patrón MVC

En una aplicación muy sencilla podemos no seguir este modelo pero en cuanto la misma se complica un poco es imprescindible programar siguiendo buenas prácticas ya que si no lo hacemos nuestro código se volverá rápidamente muy difícil de mantener.

Hay muchas formas de implementar este modelo. Si estamos haciendo un proyecto con POO podemos seguir el patrón MVC usando clases. Crearemos dentro de la carpeta *src* 3 subcarpetas:

- *model*: aquí incluiremos las clases que constituyen el modelo de nuestra aplicación
- *view*: aquí crearemos un fichero JS que será el encargado de la UI de nuestra aplicación
- *controller*: aquí crearemos el fichero JS que contendrá el controlador de la aplicación

De este forma, si quiero cambiar la forma en que se muestra algo no hace falta tocar nada del modelo sino que voy directamente a la vista y modifico la función que se ocupa de ello.

La vista será una clase sin propiedades (no tendrá un constructor), sólo contendrá métodos para renderizar los distintos elementos de la vista.

El controlador será una clase cuyas propedades serán el modelo y la vista, de forma que pueda acceder a ambos elementos. Tendrá métodos para las distintas acciones que pueda hacer el usuario (que se ejecutarán como respuesta a las acciones del usuario sobre nuestra página, lo veremos en el tema de *eventos*). Cada uno de esos métodos llamará a métodos del modelo (para obtener o cambiar la información necesaria) y posteriormente de la vista (para reflejar esos cambios en lo que ve el usuario).

El fichero principal de la aplicación instanciará un controlador y lo inicializará.

Por ejemplo, siguiendo con la aplicación para gestionar un almacén. El modelo constará de la clase *Store* que es nuestro almacén de productos (con métodos para añadir o eliminar productos, etc) y la clase *Product* que gestiona cada producto del almacén (con métodos para crear un nuevo producto, cambiar sus características, etc).

El fichero principal sería algo como:

- main.js

```
const storeApp = new StoreController();   // crea el controlador
storeApp.init();        // lo inicializa, si es necesario

// Podemos añadir algunas líneas que luego quitaremos para imitar acciones del usuario
// y así ver el funcionamiento de la aplicación:
storeApp.addProductToStore({ name: 'Portátil Acer Travelmate E2100', price: 523.12 });
storeApp.changeProduct({ id: 1, price: 515.95 });
storeApp.deleteProduct(1);
```

- controller/index.js

```
class StoreController {
    constructor() {
        this.productStore = new Store(1);   // crea el modelo, un Store con id 1
        this.storeView = new StoreView();   // crea la vista
    }

    init() {
        this.storeView.init();      // inicializa la vista, si es necesario
    }

    addProductToStore(prod) {
        // haría las comprobaciones necesarias sobre los datos y luego
```

```
        const newProd = this.productStore.addProduct(prod); // dice al modelo que añada el producto
        if (newProd) {
            this.storeView.renderNewProduct(newProd);   // si lo ha hecho le dice a la vista que lo pinte
        } else {
            this.storeView.showErrorMessage('error', 'Error al añadir el producto');
        }
    }
    ...
    // También se incluirían los métodos escuchadores para las acciones del usuario sobre la página
}
```

- view/index.js

```
class StoreView{
    init() {
        ...     // inicializa la vista, si es necesario
    }

    renderNewProduct(prod) {
        // código para añadir a la tabla el producto pasado añadiendo una nueva fila
    }
    ...

    showMessage(type, message) {
        // código para mostrar mensajes al usuario y no tener que usar los alert
    }
}
```

- model/store.class.js

```
class Store {
    constructor (id) {
        this.id=Number(id);
        this.products=[];
    }

    addProduct(prod) {
```

```
        // comprueba que los datos sean correctos y llama a la clase Product para que cree un nuevo producto
        ...
        let newProd = new Product(id, name, price, units);
        this.products.push(newProd);
        return newProd;
    }

    findProduct(id) {
        ...
    }
    ...
}
```

- model/product.class.js

```
class Product {
    constructor (id, name, price, units) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.units = units;
    }
    ...
}
```

Podéis obtener más información y ver un ejemplo más completo en [https://www.natapuntes.es/patron-mvc-en-vanilla-javascript/](

Bibliografía

- Curso 'Programación con JavaScript'. CEFIRE Xest. Arturo Bernal Mayordomo
- Curso de JavaScript y TypeScript de Arturo Bernal en Youtube
- MDN Web Docs. Moz://a. https://developer.mozilla.org/es/docs/Web/JavaScript
- Introducción a JavaScript. Librosweb. http://librosweb.es/libro/javascript/
- Curso de Javascript (Desarrollo web en entorno cliente). Ada Lovecode - Didacticode (90 vídeos)
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). García Barea, Sergi
- Apuntes Desarrollo Web en Entorno Cliente (DWEC). Segura Vasco, Juan. CIPFP Batoi.