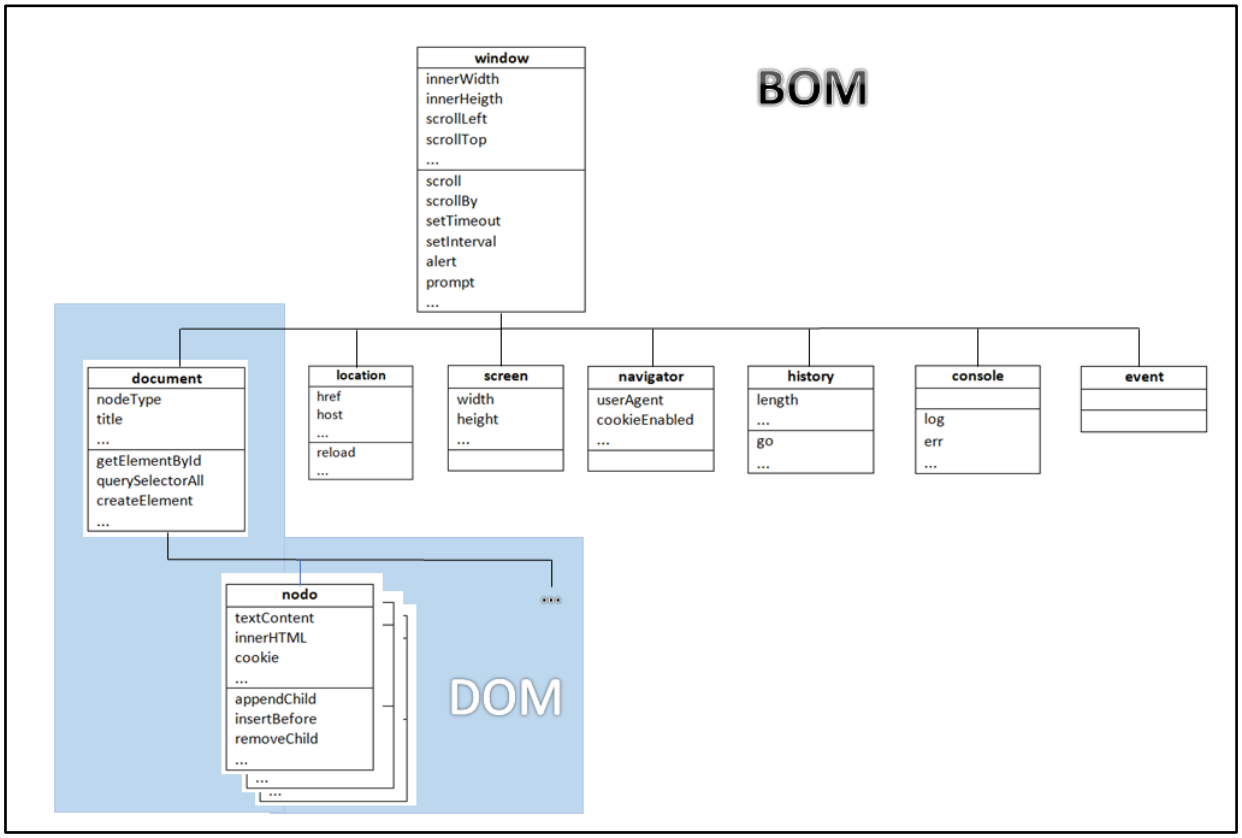
UD7. MANIPULACIÓN DEL MODELO DE OBJETOS DEL DOCUMENTO

1. EL OBJETO WINDOW

1.1. BOM, MODELO DE OBJETOS DEL NAVEGADOR



Hemos trabajado con varios métodos del objeto **window**, que resultan tan importantes que incluso se puede obviar su nombre al usar sus métodos y propiedades. Es el caso de **alert** que es un método que permite mostrar al usuario información en un cuadro de diálogo. Casi ningún desarrollador lo invoca con el nombre completo **window.alert**, usan **alert** a secas.

El objeto window es la propia ventana y es el elemento fundamental en el manejo de las aplicaciones web desde JavaScript. Es también la raíz de toda la organización de objetos a los que JavaScript puede acceder para manipular todos los aspectos de una aplicación web.

Una característica fundamental del JavaScript, de hoy en día, es que se puede ejecutar en todo tipo de entornos, no solo en los navegadores. Pero, por ejemplo, **node.js** no dispone del objeto **window** porque no es un entorno basado en la consola del navegador. Cuando usamos este objeto es señal de que estamos escribiendo una aplicación web en el lado del cliente.

Más adelante hablaremos del **DOM (Document Object Model)**, el modelo de objetos del documento que es la parte más importante del trabajo de JavaScript en la creación de aplicaciones web.

Pues bien, **DOM** es parte de lo que se conoce como **BOM (Browser Objects Model)** y que aglutina toda la estructura organizativa de los objetos del navegador. A partir de ahora, el navegador no es una simple pantalla, sino un conjunto de objetos que podemos consultar y manipular. En él tenemos objetos como el navegador, la barra de búsqueda, la consola, el documento, etc.

Aunque **BOM** no forma parte del estándar oficial del lenguaje JavaScript, lo cierto es que los distintos navegadores han igualado la forma de trabajar con estos objetos. Por tanto, podemos hablar de un uso estándar de los objetos del navegador.

1.2. OBJETO NAVIGATOR

Es un objeto que representa al navegador del usuario que está utilizando la aplicación web. Posee numerosas propiedades de lectura que nos permiten obtener información interesante. Quizá la más utilizada es userAgent que permite obtener la cadena de información del navegador mediante la que podremos saber el navegador del usuario, su versión, motor, etc. Abriendo el navegador y escribiendo ese comando en la consola:

```
console.log(navigator.userAgent);
```

Obtendremos información que varía en cada navegador. En Google Chrome veremos:

```
Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Mobile Safari/537.36  VM40:1
```

En Internet Explorer 11:

```
Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729; wbx 1.0.0; wbxapp 1.0.0; Zoom 3.6.0; rv:11.0) like Gecko
```

En Firefox:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:103.0) Gecko/20100101 Firefox/103.0
```

Otras propiedades y métodos del objeto **navigator** son:

PROPIEDAD O MÉTODO	USO
clipboard	Devuelve un objeto para gestionar desde el código el portapapeles
cookieEnabled	Valor booleano que indica si las cookies están activadas en el navegador
geolocation	Obtiene un objeto que permite usar la API de geolocalización para acceder a la posición GPS del usuario.
JavaEnabled()	Devuelve un valor booleano que indica si el plugin de Java está activado en el navegador
language	Devuelve un string con el código de lenguaje del navegador del usuario. Por ejemplo “es-ES”
mimeTypes	Obtiene un array con los tipos MIME aceptados por el navegador
onLine	Valor booleano que indica si el navegador está trabajando online o no.
plugins	Devuelve un array con información sobre los plugins instalados en el navegador
serviceWorker	Obtiene un objeto de tipo ServiceWorkerContainer capaz de trabajar con objetos de tipo ServiceWorker

storage	Devuelve un objeto de tipo StorageManager capaz de manejrar la API de almacenamiento de datos persistentes.

Algunas propiedades abren la puerta a manejar elementos avanzados en la creación de aplicaciones web, por lo que el objeto **navigator** no tiene que ser olvidado si se desea profundizar en esos aspectos.

1.3. OBJETO SCREEN

Se trata de otra propiedad miembro de window. Nos permite acceder a un objeto que representa a la pantalla. Permite obtener propiedades relacionadas con la pantalla del dispositivo en el que está navegando el usuario. Propiedades interesantes de este objeto son:

PROPIEDAD O MÉTODO	USO
availTop	Primera coordenada superior de la pantalla que se puede utilizar en nuestras aplicaciones
availLeft	Primera coordenada izquierda de la pantalla que se puede utilizar en nuestras aplicaciones
availHeight	Número de píxeles de la altura de pantalla del usuario
availWidth	Número de píxeles de la anchura de pantalla del usuario.
height	Altura completa de la pantalla del usuario en píxeles
width	Anchura completa de la pantalla del usuario en píxeles
colorDepth	Profundidad en bits de los colores de la pantalla. El número de colores disponibles en la pantalla será 2 elevado a la cantidad devuelta por esta propiedad.

orientation	Devuelve un objeto de tipo ScreenOrientation que sirve para saber la orientación de la pantalla. Ejemplo de uso:
	console.log(screen.orientation.type);
	En una pantalla normal de escritorio retornaría:
	`landscape-primary`

1.4. OBJETO LOCATION

Se trata de otro objeto interesante que posee el objeto window. La utilidad de location reside en el hecho de que representa la dirección URL de la propia aplicación. Esta misma propiedad es accesible mediante **document**.location, que es una referencia al mismo objeto.

La propiedad más importante del objeto es **href** que contiene la URL completa de la página. Ejemplo:

```
console.log(location.href);
```

Podría devolver algo como: **https://www.beniarjo.es/va/portada**

Más interesante aún es el hecho de que podemos ir a otra página simplemente cambiando esta propiedad:

```
localtion.href="https://github.com";
```

Si ejecutamos este comando desde la consola, veremos cómo nos trasladamos a la página de GitHub, independientemente de la página que estuviéramos viendo. Hay una forma más cómoda incluso de obtener el mismo resultado, que es modificar directamente la propiedad **location**:

```
location="https://github.com";
```

Este objeto tiene otra serie de propiedades (que son modificables) y que nos permiten obtener cada apartado de la URL de la página:

PROPIEDAD O MÉTODO	USO
protocol	Obtiene el nombre del protocolo que se usa para acceder a la aplicación. Ese nombre incluye los 2 puntos. Ejemplo: **https:**
host	Toma de la URL lo que se corresponde con la dirección del host. Incluye el puerto si no es un puerto estándar. Ejemplo: www.beniarjo.es:4321
hostname	Idéntica a la anterior, pero sin incluir el puerto
pathname	Obtiene la ruta de directorios a partir del host en la URL. Por ejemplo: /manuales/html/introduccion-html.html
search	Obtiene la cadena de búsqueda de la URL. Por ejemplo: ?lang=es&registrado=si
hash	Obtiene el marcador, si es que lo lleva, de la URL. Por ejemplo en la dirección http://www.beniarjo.es/web#lmsgi obtendría #lmsgi.
username	Obtiene el nombre de usuario que haya en la URL, si es que lo hay
password	Si es el caso, devuelve la contraseña que haya en la URL
origin	Es la única propiedad no modificable, es de solo lectura. Permite obtener de la URL la dirección canónica. La cual está formada por el protocolo, el puerto y el nombre de host.
reload([servidor])	Recarga la página. El parámetro opcional servidor es un booleano que, si vale true, obliga a recargar la página desde el servidor, si vale false se permite cargar la página usando la caché del navegador.

1.5. OBJETO HISTORY

Se trata del objeto que representa el historial de páginas visitadas por el usuario. El método más interesante de este objeto es go, el cual recibe un número que nos permite navegar por el historial de

páginas. Así: go(-1) iría a la página anterior en el historial, go(1) iría a la página siguiente. Go(0) es otra forma de recargar la página actual.

La otra propiedad interesante de este objeto es length, que devuelve el tamaño actual del historial.

Ejemplo:

```
history.go(-(history.length-1))
```

Este código nos permite volver a la primera página desde el historial.

1.6. OTRAS PROPIEDADES Y MÉTODOS DE WINDOW

El objeto **window** tiene otras propiedades y métodos que pueden ayudarnos a realizar algunas acciones desde JavaScript. Ya conocemos las propiedades **history**, **screen** y **navigator** cuyas posibilidades hemos comentado en otros apartados. Hay otros objetos interesantes.

También conocemos métodos de window, como: **alert**, **confirm** y **prompt** que se encargan de los mensajes al usuario.

A continuación, se detallan algunas de propiedades:

PROPIEDAD O MÉTODO	USO
innerWidth	Anchura interior de la ventana. Es la anchura real disponible en el área de contenido de la ventana teniendo en cuenta su tamaño actual
innerHeight	Altura interior de la ventana. Es la altura real disponible en el área de contenido de la ventana teniendo en cuenta su tamaño actual.
outerWidth	Anchura exterior de la ventana
innerHeight	Altura exterior de la ventana
screenX	Distancia de la ventana al borde izquierdo de la pantalla
screenLeft	Idéntica a la anterior
screenY	Distancia de la ventana al borde superior de la pantalla
screenTop	Idéntica a la anterior
scrollX	Indica el desplazamiento horizontal que el usuario ha realizado usando las barras de desplazamiento horizontales o arrastrando un dispositivo táctil en horizontal.
scrollY	Indica el desplazamiento vertical que el usuario ha realizado usando las barras de desplazamiento verticales o arrastrando un dispositivo táctil en vertical.
status	Referencia a la **barra de estado** de la ventana. Permite ver y modificar su contenido. La barra de estado era una franja horizontal situada en la parte

	inferior de la ventana en la que se daba información del estado. Hoy en día no se utiliza esta barra, pero la propiedad sigue estando disponible.
event	Objeto del evento actualmente en ejecución.
fullScreen	Booleano que indica si la aplicación se muestra a pantalla completa. No funciona en Google Chrome en este momento (sí en Firefox por ejemplo)
localStorage	Referencia al objeto de tipo **Storage** que permite almacenamiento de datos en modo local sin tiempo de expiración.
getSelection()	Obtiene un objeto de tipo **Selection** que nos sirve para poder obtener información sobre el texto seleccionado. Si queremos, simplemente, obtener el texto seleccionado actualmente por consola, sería: `console.log(getSelection().toString())`

scroll(x,y)
 	
O bien,
 	
scroll(objetoScroll)
	Sirve para desplazar la página a una posición concreta. Permite indicar el desplazamiento mediante 2 coordenadas (x horizontal e y vertical). Ejemplo:
	
	scroll(0,100)
	
	Desplaza la página haciendo que la barra vertical se desplace a la posición marcada por 100 píxeles. En horizontal se desplaza totalmente a la izquierda.
	
	Se puede pasar, en lugar de las 2 coordenadas, un objeto de tipo **ScrollToOptions** (algunos navegadores como Edge o Explorer no aceptan este parámetro todavía). Los objetos **ScrollToOptions** tienen estas 3 propiedades:
	
	• **top**. Desplazamiento vertical
	• **left**. Desplazamiento horizontal
	• **behaviour**. Comportamiento. Por ahora admite como valores: **auto** (se desplaza sin animación) y **smooth** (con animación suave).
	
	Ejemplo:
	
	`scroll({`
	`  top:100,`
	`  left;0,`
	`  behaviour:”smooth”`
	`});`
	El documento se desplaza automáticamente a la posición 1000 píxeles de forma progresiva con una animación

scrollBy(x,y)
 	
O bien,
 	
scrollBy(objetoScroll)
	Los parámetros funcionan igual que con la propiedad anterior, pero ahora el desplazamiento es relativo respecto a la posición actual. Ejemplo:
	
	`scrollBy({`
	`  top:innerHeigth,`

	left;0, behaviour:"smooth" }); Este código hace que la página se desplace suavemente una pantalla completa hacia abajo
scrollTo(x,y) O bien, **scrollTo**(objetoScroll)	Los parámetros funcionan igual que con la propiedad **scroll**. La diferencia es que los valores indicados son coordenadas y no píxeles de desplazamiento (normalmente ambas cosas coinciden)
stop()	Detiene la carga de la página
find(texto)	Busca el texto en el documento actual y devuelve true o false si lo encuentra. Además, la mayoría de navegadores resaltan la palabra dentro de la página a la vez que realizan un scroll hacia ella.
getComputedStyle(elemento)	Permite obtener la información de las propiedades CSS en uso por parte de un elemento
open(URL,nombreVentana,ajustes)	Abre una ventana y muestra el contenido de la URL, indicada. Opcionalmente, se puede indicar un nombre para la ventana y los ajustes de la misma (tamaño, barras de desplazamiento, etc)

2. ¿QUÉ ES EL DOM?

2.1. EL MODELO DE OBJETOS DEL DOCUMENTO

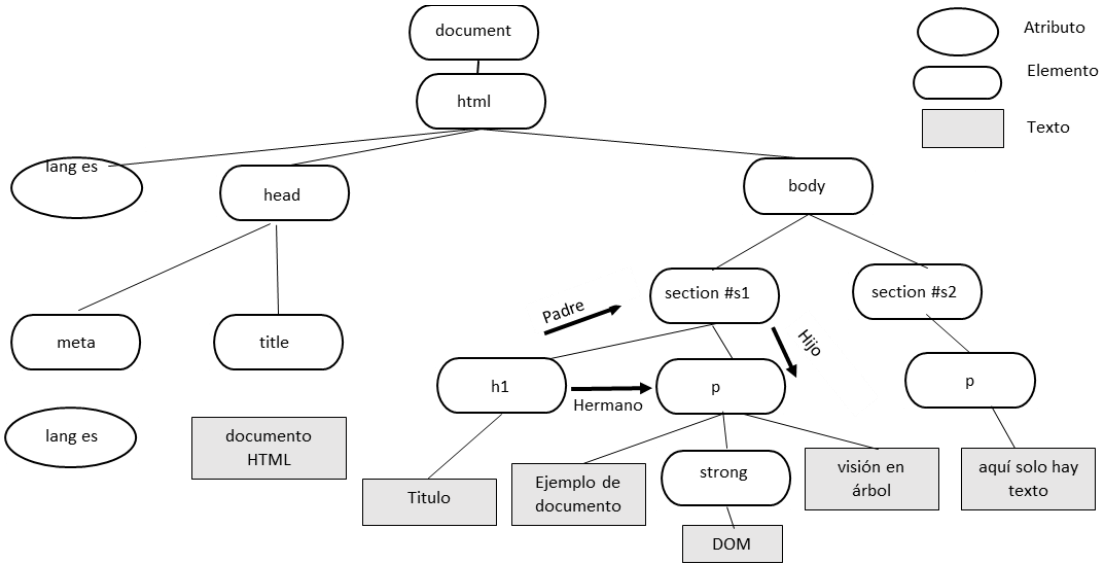
JavaScript se ideó para que fuera un lenguaje con capacidad de manipular el contenido y apariencia de una página web de forma dinámica. Para hacer esto posible, los elementos de la página web se tienen que poder manipular como una serie de objetos relacionados. Así apareció el término DOM (Document Objetcs Model) que define una página web como una estructura organizada de objetos que forman un árbol de recorrido.

Si, por ejemplo, tenemos este código HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Documento HTML</title>
</head>
        <body>
<section id="s1">
<h1>Título</h1>
<p>Ejemplo de documento <strong>DOM</strong>Visión en árbol</p>
</section>
```

```
<section id="s2">
<p>Aquí solo hay texto</p>
</section>
</body>
</html>
```

Podemos entender que forman este tipo de estructura:



Ejemplo de estructura de modelo de objetos del documento (DOM)

Realmente la estructura DOM no se creó solo para JavaScript, la idea inicial es que fuera manipulable también mediante otros lenguajes. Claramente, JavaScript es el lenguaje que más se utiliza para el DOM, pero lo cierto es que es una estructura de objetos independiente de todo lenguaje. Eso hace que haya ciertas particularidades en el manejo de los objetos del documento que son extrañas desde el punto de vista de JavaScript.

Otro detalle importante es que esta estructura manipulativa, no solo se ideó para documentos HTML, sino que funciona de la misma manera para manipular documentos XML.

En todo caso, la idea es la imagen anterior: una estructura de objetos en forma de árbol donde los nodos representan objetos del documento y cada nodo posee un nodo padre a través del cual nos acercamos a la raíz del árbol.

2.2. EL OBJETO DOCUMENT

La forma de acceder a los objetos del documento JavaScript es a través de un objeto llamado precisamente **document**. Este objeto en realidad es una propiedad del objeto **window**. Podemos indicar simplemente la palabra **document** para acceder a este objeto, aunque su nombre es **window.document**.

Los diferentes métodos y propiedades del objeto document nos van a permitir acceder a los elementos del documento para su manipulación.

2.3. TIPOS DE NODOS

Más adelante veremos muchos métodos para acceder a los elementos de un documento, pero el más habitual es el método **getElementById** del objeto **document**. Así si usamos el código anterior podemos acceder a la primera sección a través del código:

```
let seccion=document.getElementById("#s1");
```

Este código hace que la variable seccion sea una referencia al elemento identificado como s1. Es decir, *seccion* es un nodo del documento. Todos los nodos poseen una propiedad llamada nodeType. Si la mostramos en este caso:

```
console.log(seccion.nodeType);
```

Se muestra el valor 1. Este valor significa que el node es un elemento. Si ejecutamos este otro código:

```
console.log(document.nodeType); //Escribe 9
```

La lista de posibles valores de nodeType es la siguiente:

VALOR	TIPO DE NODO	CONSTANTE RELACIONADA
1	Elemento	Node.ELEMENT_NODE
2	Atributo	Node.ATTRIBUTE_NODE
3	Texto	Node.TEXT_NODE
4	Apartado CDATA	Node.CDATA_SECTION_NODE
5	Referencia a entidad	Node.ENTITY_REFERENCE_NODE
6	Entidad	Node.ENTITY_NODE
7	Instrucción de procesado	Node.PROCESSING_INSTRUCTION_NODE
8	Comentario	Node.COMENT_NODE
9	Documento completo	Node.DOCUMENT_NODE
10	Nodo de tipo de documento	Node.DOCUMENT_TYPE_NODE
11	Nodo de fragmento de código	Node.DOCUMENT_FRAGMENT_NODE
12	Nodo de anotación	Node.NOTATION_NODE

Algunas entradas están muy relacionadas con documentos **XML** por lo que no nos serán útiles para manipular documentos HTML. Normalmente, nos bastará con las 3 primeras entradas de esta tabla. Las constantes definidas en el objeto **Node**, nos permiten no tener que aprender los números y manejar en su lugar las constantes que son más entendibles.

Hay que hacer notar que la propiedad **nodeType** es de lectura. Como también lo es la propiedad **nodeName**. Esta última propiedad nos permite obtener el nombre del nodo.

```
console.log(s1.nodeName);
```
El código anterior escribe **SECTION**, porque la variable *seccion* hace referencia a un elemento **section** de HTML. Sin embargo:

```
console.log(document.nodeName);
```
Escribe **#document**. Los valores de **nodeName** cambian en función del tipo de nodo que tengamos:

La lista de posibles valores de **nodeName** es la siguiente:

TIPO DE NODO	VALOR DEVUELTO POR NODENAME
Elemento	Nombre de la etiqueta del elemento
Atributo	Nombre del atributo
Texto	#text
Comentario	#comment
Documento	#document

Finalmente, la propiedad **nodeValue** es capaz de devolver el valor del nodo, aunque en el caso del documento completo **nodeValue** devuelve **null**.

Estas 3 propiedades (**nodeType**, **nodeName** y **nodeValue**) facilitan obtener la información más importante sobre un nodo a la hora de manipularlo.

3. SELECCIÓN DE ELEMENTOS DEL DOM

El objeto **document** proporciona numerosos métodos para seleccionar elementos. Las últimas revisiones DOM han añadido nuevos métodos que facilitan mucho la labor del desarrollador.

3.1. SELECCIÓN POR EL IDENTIFICADOR

Se trata de seleccionar un nodo del documento por el valor del atributo id. El método que lo permite es **getElementById** que recibe un string con el valor buscado. Retorna como resultado el elemento en sí o bien el valor **null** si ese identificador no se encuentra en el elemento.

```
let s1=document.getElementById("s1");
```

3.2. POR ETIQUETA

La selección por etiqueta permite seleccionar todos los elementos que tengan ese nombre de etiqueta. El método que lo consigue es **getElementByTagName**. En este caso se devuelve una lista de elementos de tipo **NodeList**. Es decir, no se selecciona un elemento (como ocurría con **getElementById**), sino todos los elementos que usen esa etiqueta.

Usando la misma página web podemos escribir este código JavaScript en la consola del navegador:

```
let elementosP=document.getElementsByTagName(“p”);
```

El cual permite que **elementosP** sea una referencia a todos los elementos de tipo p. Por eso este código:

```
console.log(elementosP.length);
```
Escribe el número 2 por consola, ya que hay 2 elementos de tipo p. Mientras que este otro código:

```
console.log(elementosP[0]);
```
Mostraría este resultado, que se corresponde con el contenido del primer elemento de tipo p.

```
> console.log(elementosP[0]);
  ▼<p>
     "Ejemplo de documento "
     <strong>DOM</strong>
     "Visión en árbol"
   </p>
```

El método getElementsByTagName lo tienen todos los elementos. Si en lugar de utilizar document, utilizamos un elemento concreto, entonces se buscan los elementos con esa etiqueta dentro del que digamos. Ejemplo:

```
<main id="principal">
<p>Uno</p>
<p>Dos</p>
<p>Tres</p>
</main>
<p>Cuatro</p>
<p>Cinco</p>
<script>
let principal=document.getElementById("principal");
let interiores=principal.getElementsByTagName("p");
console.log(interiores.length)
</script>
```
El resultado de el código anterior es el número 3. La expresión document.**getElementsByTagName**("p") hubiera devuelto 5.

3.3. OBJETOS NODELIST

El selector por etiqueta nos ha permitido conocer a los objetos **NodeList** que representan una colección de nodos del DOM, es decir, una serie de elementos del documento. Es fundamental este objeto porque son muchas las propiedades y métodos de document que devuelven este tipo de objetos.

Aunque los objetos NodeList no son realmente arrays, se manejan en gran medida como si lo fueran. Se pueden usar todos los bucles de recorrido vistos con los arrays, se puede acceder a cada elemento indicando su índice entre corchetes y dispone de propiedades comunes a los arrays como **length** o **forEach**. Sin embargo, otros métodos de los arrays como **slice**, **sort** o **join**, no están disponibles.

No obstante, si deseamos disponer de esos métodos y de otros, tendremos que convertir este tipo de objetos en un array, lo cual es posible mediante el operador de propagación:

```
let arrayElementosP=[...elementoP];
```

O, también:

```
let arrayElementosP=Array.from(elementosP);
```

Pero, lo cierto es que no suele ser necesaria esta conversión.

3.4. POR CLASE

Se trata de seleccionar todos los elementos que tienen asignada una clase determinada de CSS. Por ejemplo:

```
<ul>
<li class="verdura">Cebolla</li>
<li class="verdura">Ajo</li>
<li class="carne">Pollo</li>
<li class="verdura">Lechuga</li>
<li class="carne">Ternera</li>
<li class="hidrato">Pan</li>
<li class="hidrato">Arroz</li>
<li class="carne">Cerdo</li>
<li class="verdura">Tomate</li>
<li class="hidrato">Lentejas</li>
</ul>
```

En esta lista de alimentos cada alimento se relaciona con una clase concreta, eso permite aplicar un CSS distinto a cada clase de alimento. Si deseamos operar desde JavaScript con los elementos de una clase concreta, disponemos del método: **getElementsByClassName**, el cual nos devuelve un objeto de tipo **NodeList** con todos los elementos que sean de la clase indicada.

```
let verduras=document.getElementsByClassName("verdura");
```

3.5. POR SELECTOR CSS

Indudablemente el selector más poderoso es **querySelectorAll**. Permite indicar, como texto, cualquier selector válido CSS. Si la sintaxis del selector no es correcta, entonces devolverá un error. Lo que devuelve, nuevamente, es una lista de elementos con aquellos que cumplan el selector.

Por ejemplo, para seleccionar los elementos de clase verdura, podemos:

```
let verduras=document.querySelectorAll(".verdura");
```

Se usa el punto, porque ahora hay que hacerlo en el formato CSS. Pero las posibilidades son muy espectaculares gracias a la potencia de los selectores de CSS:

```
// Selecciona los elementos li que están dentro de ul
document.querySelectorAll(“ul li”);
// Selecciona el primer elemento li de cada lista
document.querySelectorAll(“li:first-of-type");
//Selecciona el primer elemento li de cada lista si es de clase verdura
document.querySelectorAll(“li.verdura:first-of-type");
```

Hay un método llamado querySelector a secas que funciona igual, pero no devuelve una lista, sino el primer elmento del documento que cumpla el criterio. No suele ser muy habitual su uso.

Hay que recordar que, aunque un código como document.**querySelectorAll**("#capa17"); está claro que solo devuelve un elemento, este método siempre devuelve una colección de elementos. Es decir, acceder al elemento en cuestión implica utilizar el índice cero del resultado.

4. OBTENER Y MODIFICAR DOM

4.1. MANIPULACIÓN DE ATRIBUTOS

4.1.1. OBTENER EL VALOR DE UN ATRIBUTO

Los elementos del DOM disponen del método getAttribute para obtener el valor de un atributo concreto. Por ejemplo:

```
let lis=document.querySelectorAll("li");
for(let li of list){
        console.log(li.getAttribute("class");
}
```

Este código recorre cada elemento de tipo **li** y nos muestra por consola la clase CSS de cada elemento.

Si un elemento no tiene atributo indicado, **getAttribute** devuelve **null**.

4.1.2. MODIFICAR EL VALOR DE UN ATRIBUTO

El método setAttribute es el que permite esta operación. El primer parámetro será el nombre del elemento y el segundo el nuevo valor.

```
elemento.setAttribute("class","verdura");
```

4.1.3. ELIMINAR UN ATRIBUTO

El método **removeAttribute** seguido de un nombre de atributo, nos permite eliminar el atributo que tenga ese nombre del elemento.

4.1.4. AÑADIR Y QUITAR ATRIBUTO DE FORMA RÁPIDA

El método **toggleAttribute** permite añadir al elemento el atributo que se indique si no está añadido ya, o quitar el atributo con ese nombre si ya está añadido. Si la primera vez que usamos el método **toggleAttribute**, añade el atributo, la siguiente lo eliminará, la siguiente lo volverá a añadir y así sucesivamente.

4.1.5. SABER SI UN ELEMENTO TIENE ATRIBUTOS

El método **hasAttribute** devuelve true si el elemento tiene atributo cuyo nombre se indica y false si no lo tiene.

4.1.6. OBTENER TODOS LOS ATRIBUTOS DE UN ELEMENTO

La propiedad *attributes* de un elemento permite obtener todos los atributos de un elemento Lo que devuelve es un objeto de tipo **NamedNodeList** que no es una estructura de array por lo que no tenemos

acceso a la mayoría de propiedades del array. Pero sí podemos acceder por número de índice y usar la propiedad **length**.

Cada elemento de la lista que devuelve esta propiedad posee una propiedad llamada **name** que contiene el nombre del atributo y **value** que contiene su valor. Por ejemplo, si tenemos este código HTML:

```
<form>
<label for="edad">Escriba su edad</label>
<input type="number" id="edad" name="edad" min="18" max="65">
</form>
```

Para navegar por los atributos del cuadro numérico y mostrarlos por consola podríamos escribir este código:

```
let listaAtributos=document.getElementById("edad").attributes;
for(let atributo of listaAtributos){
        console.log(`Atributo: ${atributo.name} `+`Valor: ${atributo.value}`);
}
```

El resultado es:

```
Atributo: type Valor: number
Atributo: id Valor: edad
Atributo: name Valor: edad
Atributo: min Valor: 18
Atributo: max Valor: 65
```

4.2. MANIPULACIÓN DEL CONTENIDO DE LOS ELEMENTOS

4.2.1. PROPIEDAD TEXTCONTENT

La propiedad **textContent** de un elemento permite obtener y modificar el texto que contiene el elemento. Hay que tener encuentra que un elemento puede tener dentro más elementos y que esta propiedad obtendrá el texto de todos ellos. Por ejemplo, si observamos este código:

```
<main>
<h1>Sistema Solar</h1>
<h2>Sol</h2>
<p id="pSol">
El Sol, una estrella de tipo espectral <strong>G2</strong> que contiene más del 99,85% de la masa del sistema. Con un diámetro de 1 400 000 km, se compone de un 75 % de hidrógeno, un 20 % de helio y 5 % de oxígeno, carbono, hierro y otros elementos.
</p>
<h2>Planetas</h2>
<p id="pPlanetas">
Los planetas, divididos en planetas interiores (también llamados terrestres o telúricos) y planetas exteriores o gigantes. Entre estos últimos <strong>Júpite</strong> y <strong>Saturno</strong> se denominan gigantes gaseosos, mientras que Urano y Neptuno suelen nombrarse gigantes helados. Todos los planetas gigantes tienen a su alrededor anillos.
```

```
</p>
</main>
```

Al ejecutar esta instrucción:

```
console.log(document.getElementById("pSol").textContent);
```

Escribirá:

```
                                                                                                    html4.h
El Sol, una estrella de tipo espectral G2 que contiene más del 99,85% de la masa del sistema. Con un diámetro
de 1 400 000 km, se compone de un 75 % de hidrógeno, un 20 % de helio y 5 % de oxígeno, carbono, hierro y otros elementos.
```

Que es el texto del párrafo indicado. Sin embargo, dentro de ese elemento hay otro de tipo **strong** y su texto aparece también en el resultado a la vez que la propia etiqueta strong desaparece del resultado Es decir, solo recoger el texto interior y no las etiquetas.

A través de esta propiedad podemos cambiar el contenido:

```
document.getElementById("pSol").textContent="Soy el <strong>Sol</strong>";
```

Sol

Soy el Sol

Vemos que, aunque hemos indicado que la palabra Sol se encuentre dentro de etiquetas **strong**, se toman de forma literal y no funcionan. La razón es que **textContent** es capaz de modificar el texto de un documento, pero no los elementos interiores.

4.2.2. PROPIEDAD INNERHTML

Se trata de una propiedad similar a la anterior, pero esta sí lee y manipula las etiquetas HTML. Por ejemplo:

```
console.log(document.getElementById("pSol").innerHTML);
```

```
                                                                                                    html4.htm:51
El Sol, una estrella de tipo espectral <strong>G2</strong> que contiene más del 99,85% de la masa del sistema. Con un diámetro
de 1 400 000 km, se compone de un 75 % de hidrógeno, un 20 % de helio y 5 % de oxígeno, carbono, hierro y otros elementos.
```

Parece el mismo resultado que obtuvimos con **textContent**. Pero la palabra G2 está contenida en una etiqueta **strong** que textContent no podíamos ver. La diferencia es que innerHTML sí lee las etiquetas.

Y no solo las lee, puede modificar el contenido usando etiquetas HTML:

```
document.getElementById("pSol").innerHTML="Soy el <strong>Sol</strong>";
```

Si vemos el resultado, ahora sí vemos que entiende las etiquetas y realmente vemos en negrita la palabra **Sol**.

Sol

Soy el **Sol**

4.3. MODIFICAR CSS

Es posible modificar el CSS de los elementos a través de los atributos **style** o **class** mediante el método **setAttribute**. Pero al ser un elemento tan importante, JavaScript proporciona métodos especiales para manipular las clases CSS de un elemento.

4.3.1. PROPIEDAD STYLE

Los elementos poseen una propiedad llamada style que permite acceder a las propiedades CSS de un elemento. Lo que realmente hace es modificar el atributo style del elemento, modifican los valores de la misma. La idea es que style es un objeto que tiene como propiedades todas las propiedades CSS en formato Camel Case (mayúsculas estilo camello). Veámoslo con ejemplos:

```
let párrafo=document.getElementsByTagName("p")[0];
```

El código anterior selecciona el primer párrafo (de tipo p) del documento, sea cual sea.

```
párrafo.style.color="red";
```

Esta línea colorea de color rojo el texto de ese párrafo.

Podemos de esa forma modificar todo el CSS que queramos, por ejemplo:

```
párrafo.style.border="1px solid black";
```

Y así con todas las propiedades. El problema viene si hubiéramos querido modificar solo el borde inferior. La propiedad CSS para hacer esa modificación se llama **border-bottom**, es un nombre problemático para la sintaxis anterior, porque hay un guión en él. En JavaScript los identificadores de propiedades, funciones, métodos, etc, no pueden tener guiones (se confunde con la resta). Lo mismo ocurre con la propiedad que modifica el color de fondo: **background-color**.

Hay que tener en cuenta que el formato de modificación CSS explicado anteriormente, es antiguo y entonces no se disponía de la posibilidad de acceder a la propiedad mediante corchetes. Por eso la propiedad style contiene una propiedad por cada propiedad CSS, pero usando el llamado formato **Camel Case**.

En este formato, **background-color** se convierte en **backgroundColor**, **border-bottom** en **borderBottom**, **text-align** en **textAlign** y así con todas las propiedades. Por lo que modificar el color de fondo se hace:

```
párrafo.style.backgroundColor="#CCC";
```

No obstante, la mayoría de navegadores actuales acepta también el formato idéntico a CSS a través de corchetes:

```
párrafo.style["background-Color"]="#FCC"; //Colorea de rojo claro
```

No obstante, hay varios problemas a tener en cuenta a la hora de usar style para modificar el CSS:

- Manipular la propiedad de los elementos style da prioridad a cualquier otro CSS (ya que se modifica la propiedad **style** de HTML), pero no siempre es lo deseable.
- Es más fácil de mantener el código si se usan clases para aplicar CSS. Especialmente, cuando queremos modificar varias propiedades CSS a la vez.

- CSS avanza muy rápido por lo que muchas propiedades CSS ya disponibles, en algunos navegadores no lo están a través de la propiedad style de JavaScript. Aunque la mayoría de navegadores se ponen al día rápido.
- No podemos consultar a través de esta propiedad las propiedades CSS que tienen un elemento al cual se le ha dado formato a través de hojas de estilo externas.

4.3.2. OBTENER LOS ESTILOS CSS QUE SE APLICAN A UN ELEMENTO

Precisamente para paliar el último problema comentado en el apartado anterior, disponemos de un método en el objeto window llamado **getComputedStyle**. Este método devuelve un objeto (idéntico al de la propiedad style vista en el apartado anterior) en el que se pueden consultar las propiedades CSS que se están aplicando a un elemento concreto. Este método solo permite ver las propiedades del elemento, pero no modificarlas.

Como hemos comentado, se trata de un método del objeto window y no de los elementos en sí. Por ello, el método requiere que se le pase el elemento a evaluar:

```
let cssPárrafo=window.getComputedStyle(párrafo);
console.log(cssPárrafo.fontFamily);
```

Este código mostrará por pantalla el valor de la propiedad CSS **font-family** que tiene el párafo. Se mostrará el valor, independientemente de si esa propiedad se asignó usando la propiedad style o mediante una hoja de estilos externa o mediante JavaScript.

4.3.3. MANIPULAR LAS CLASES CSS

Los elementos disponen de una propiedad llamada **className**. Mediante esa propiedad podemos asignar una clase CSS a un elemento de la página. Por ejemplo:

```
párrafo.className=”remarcado ";
```

También podemos simplemente obtener la clase del elemento. Por ejemplo:

```
console.log(párrafo.className); //Escrie remarcado
```

Los problemas surgen cuando más de una clase asignada al elemento. Sí podemos ver las clases asignadas. Por ejemplo:

```
<p id="nota20" class="remarcado anotacion">
A tener en cuenta que el 6 de abril se cierra por reforma </p>
```

Si ahora ejecutamos este código en la consola:

```
console.log(document.getElementById(“nota20”).className);
```

Obtendremos este resultado:

```
remarcado anotacion
```

La propiedad **classList** dispone de estas interesantes propiedades:

- **add**(nombreCalse1 [,nombreClase2]): Permite añadir una clase al elemento. Basta con indicar el nombre de la clase entre comillas. Admite indicar varias clases si se separan con comas.

- **remove**(nombreClase1 [,nombreClase2]): Método contrario al anterior. Retira del elemento la clase o clases que se indiquen.
- **toggle**(nombreClase [,forzar]): Si se usa un solo parámetro, será le nombre de una clase. El método hace que si el elemento no tiene asignada esa clase, se asigna. Si el elemento ya tenía esa clase, la quita. Es un método muy interesante y que se usa muy a menudo.

  El segundo parámetro (**forzar**) es un valor booleano que si vale **true** entonces añade la clase, independientemente de si el elemento ya lo tenía o no. Si vale **false**, entonces quita la clase del elemento. Es decir, este método es capaz de hacer lo que hacían los otros dos.

- **contains**(nombreClase): Devuelve **true** si el elemento tiene asignada la clase cuyo nombre se indica en el parámetro.
- **replace**(mombreViejo, nombreNuevo): Permite cambiar una clase por otra en el elemento.

En la práctica, dar formato CSS usando clases, en lugar de manipular propiedades CSS una a una, es más cómodo. Y **classList** aporta todavía más comodidad gracias a las propiedades que ofrece.

4.4. OBTENER ATRIBUTOS DATA

En los documentos HTML existe la posibilidad de crear atributos **data**. Se trata de un tipo de atributos creados por los propios desarrolladores a voluntad. Se usan para almacenar información que puede ser manipulada desde JavaScript.

Los atributos data empiezan por ese mismo término (data) seguidos de un guión y luego el nombre en sí que queramos data al atributo. Por ejemplo:

```
<p id="pi" data-tipo="inicio de libro" data-libro="Don Quijote"
        data-autor-principal="Miguel de Cervantes">
En un lugar de la Mancha...
</p>
```

En este ejemplo se han diseñado 2 atributos de forma personal. No es muy lógico grabar tanta información, pero nos interesa como ejemplo formativo. Ahora la cuestión es cómo se manipulan estos atributos desde JavaScript.

La propiedad **dataset** es la encargada de conseguirlo. El manejo es muy similar a lo que vimos respecto a la propiedad **style**.

dataset es un objeto que contiene como propiedades cada una de los atributos data del elemento actual. Para acceder concretamente a uno se usa una sintaxis de tipo **Camel Case**. Concretamente en este ejemplo, podemos ver los valores de los 3 atributos de esta forma:

```
let p1=document.getElementById("pi");
console.log(p1.dataset.libro);
console.log(p1.dataset.autorPrincipal);
console.log(p1.dataset.tipo);
```

```
Don Quijote
Miguel de Cervantes
inicio de libro
```

Es posible también modificar un atributo data, o crearlo si no existe:

```
p1.dataset.publicacion=”Siglo XVII”;
```

4.5. NAVEGAR POR EL DOM

Los métodos que se explican en este apartado nos permiten obtener información de la estructura del DOM del documento. Son imprescindibles para manipular dicha estructura y par.

4.5.1. OBTENER LOS HIJOS DE UN ELEMENTO

Veamos este código:

```
<section id="s1">
        <h1 id="titulo">¿Quién soy?</h1>
<p id="parrafo1">Hola, soy <strong>Gema</strong>, soy la profe</p>
<p id="parrafo2">Pero solo hace 2 años</p>
<p id="parrafo3">Y este es mi instituto</p>
</section>
```

Todos los elementos disponen de una propiedad childNodes, la cual devuelve una lista de nodos. Como ejemplo, utilizaremos el párrafo identificado como parrafo1, para obtener su lista de nodos hijo a través de la consola de JavaScript.

```
let parrafo1=document.getElementById("parrafo1");
for(let hijo of parrafo1.childNodes){
console.log(`Texto: ${hijo.nodeValue}"\n`+
`Tipo de nodo: ${hijo.nodeType}, ${hijo.nodeName}`);
}
```

```
Texto: Hola, soy "
Tipo de nodo: 3, #text
Texto: null"
Tipo de nodo: 1, STRONG
Texto: , soy la profe"
Tipo de nodo: 3, #text
```

Un detalle que suele confundir es que el texto que está dentro de un elemeneto es también un nodo del DOM. Si mostramos los tipos, al texto se le otorga el número 3 y a los elementos el número 1.

Hay una propiedad llamada **children** que funciona igual que childNodes, pero solo obtiene los nodos hijos que sean elementos:

```
Texto: null"
Tipo de nodo: 1, STRONG
```

Solo hay un nodo hijo del primer párrafo que sea un elemento.

4.5.2. PROPIEDADES QUE FACILITAN LA NAVEGACIÓN POR EL DOM

La figura anterior nos da una idea de cómo podemos recorrer el DOM. Aparte del método que obtiene todos los nodos hijos, tenemos las siguientes propiedades que permiten referenciar nodos relacionados con el elemento actual:

PROPIEDAD	USO
firstChild	Selecciona el primer nodo hijo del elemento actual
lastChild	Selecciona el último nodo hijo del elemento actual
nextSibling	Selecciona el siguiente hermano respecto del nodo actual. Son hermanos los nodos que están al mismo nivel en el árbol DOM.
previousSibling	Obtiene el hermano anterior al nodo actual
parentNode	Obitene el padre del nodo actual
firstElementChild	Obtiene el primer hijo del nodo actual que sea un elemento
lastElementChild	Obtiene el último hijo del nodo actual que sea un elemento
nextElementSibling	Obtiene el siguiente hermano anterior que sea de tipo elemento
previousElementSibling	Obtiene el elemento hermano anterior al nodo actual
childElementCount	Número de elementos hijos del nodo actual

Como se puede apreciar en la tabla, todas las propiedades tienen 2 versiones, una sin la palabra **Element** y otra con ella. Suelen ser más interesantes las que tienen la palabra Element, ya que solo realizan recorridos a través de nodos que sean elementos de la página(**p, strong, h1, div,** etc.)

4.6. AÑADIR ELEMENTOS

4.6.1. CREAR ELEMENTOS

El elemento createElement del objeto document es el que permite crear elementos. Requiere que indiquemos el nombre d ela etiqueta de dicho elemento:

```
let capa1=document.createElement(“div”);
```

La variable capa1 es la referencia al nuevo elemento. Este elemento no se mostrará toavía en el documento, porque realmente no forma parte de los nodos visibles al no haber indicado su posición en el árbol de nodos que cuelgan de document.

Si tiene disponibles las propiedades habituales de los elementos.

```
capa1.innerHTML=”<p>Esta es mi nueva capa</p>;
capa1.setAttribute(“id”,”capa1”);
```

4.6.2. CREAR NODOS DE TEXTO

El método **createTextNode** crea un nodo de tipo texto. Para ello basta con indicar el texto que deseamos colocar en el nodo:

```
let nodoT1=document.createTextNode("Este texto no se muestra todavía");
```

Al igual que ocurría con createElement, los nodos recién creados no se muestran hasta que no los coloquemos dentro del documento

4.6.3. AÑADIR UN NODO HIJO

El método **appendChild** permite colocar un nodo como hijo de otro elemento. Es un método que poseen los nodos del DOM y que tiene como único parámetro el nodo que deseamos colocar como hijo.

Ejemplo:

```
<main id="principal">
<p>Soy el primer párrafo</p>
<p>Soy el segundo párrafo</p>
<p>Soy el tercer párrafo</p>
</main>
<script>
let capa=document.getElementById("principal");
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
capa.appendChild(nuevoP);
</script>
```

En el código anterior la capa HTML de tipo main identificada como principal tiene 3 párrafos. El código JavaScript crea un nuevo párrafo y le coloca el texto "Soy el nuevo párrafo". Cuando usamos el método appenChild para añdirlo como hijo de la capa principal, aparecerá al final de los párrafos anteriores. Resultado:

Soy el primer párrafo

Soy el segundo párrafo

Soy el tercer párrafo

Soy el nuevo párrafo

4.6.4. INSERCIÓN DE NODOS EN POSICIONES CONCRETAS

El método anterior tiene como inconveniente, que el elemento que deseamos añadir debe ir siempre colocado al final. Por ello disponemos de otro método llamado insertBefore que permite indicar la posición en la que queremos colocar el elemento. PAra ello el nodo que será padre del que qeuremos añadir, debe de tener nodos hijo. De otro modo deberíamos usar appendChild.

El método insertBefore tiene 2 parámetros: el primero es el nodo que deseamos añadir y el segundo es el nodo hijo delante del cual quedará colocado el que deseamos añadir. Evidentemente, no podemos, con este método, añadir un nodo al final, pero para eso disponemos del appendChild.

Usando el mismo código HTML, veamos cómo colocar un nuevo párrafo como segundo:

```
let capa=document.getElementById("principal");
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
//El elemento posterior es el segundo párrafo de la capa principal
let pPosterior=document.querySelectorAll(
          #principal p:nth-of-type(2)")[0];
capa.insertBefore(nuevoP,pPosterior);
```

Soy el primer párrafo

Soy el nuevo párrafo

Soy el segundo párrafo

Soy el tercer párrafo

4.7. REEMPLAZAR ELEMENTOS

El método **replaceChild** es más agresivo. Permite cambiar un nodo por otro. Necesita como parámetros: primero el nuevo nodo que se desea colocar y después el nodo que se desea quitar. El nuevo nodo reemplaza al antiguo.

El método devuelve una referencia al nodo retirado, que realmente no se quita del todo, sino que simplemente se retira del árbol del DOM. Podemos usar la referencia al nodo quitado para colocarlo dentro de otro elemento.

Ejemplo:

```
let capa=document.getElementById("principal");
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
//El elemento posterior es el segundo párrafo de la capa principal
let pPosterior=document.querySelectorAll(
"#principal p:nth-of-type(2)")[0];
capa.replaceChild(nuevoP,pPosterior);
</script>
```

Soy el primer párrafo

Soy el nuevo párrafo

Soy el tercer párrafo

4.8. ELIMINAR ELEMENTOS

El método removeChild es un método de los nodos que recibe como único parámetro el nodo a eliminar. Este nodo no se eleminará realmente de lamemoria solo se retira del árbol de elementos. De hecho, el método devuelve una referencia al objeto retirado.

En este ejemplo se quitaría el segundo párrafo y se colocaría al final con ayuda de appendChild.

```
let capa=document.getElementById("principal");
let pSegunda=document.querySelectorAll(
"#principal p:nth-of-type(2)")[0];
//Se elimina el segundo párrafo
capa.removeChild(pSegunda);
capa.appendChild(pSegunda);
```

Soy el primer párrafo

Soy el tercer párrafo

Soy el segundo párrafo

4.9. COLECCIONES VIVAS

IMAGEN

Las acciones vistas en los apartados anteriores permiten modificar los elementos que inicialmente aparecían en la página. Los elementos creados, modificados o eliminados desde JavaScript se dice que han sido acciones dinámicas o acciones vivas o en directo.

No hay ningún problema en hacer esas modificaciones, es verdad que el código HTML del documento mostrará elementos que ya no están en la página. Pero los navegadores permiten ver la realidad de los elementos tal cual se muestran en cada instante a través del panel de depuración.

Pero sí hay problemas con algunos métodos porque algunos retornan colecciones de datos que no se actualizan cuando la página cambia. La mayoría sí lo hace como podemos ver en este ejemplo:

```
let capa=document.getElementById("principal");
let hijos=capa.children;
console.log(hijos.length);
```

Supongamos, si seguimos teniendo el mismo código HTML original de los ejemplos anteriores, que por consola aparece el número 3, indicando que el elemento con identificador principal tiene 3 hijos. Este otro código añade un nuevo hijo:

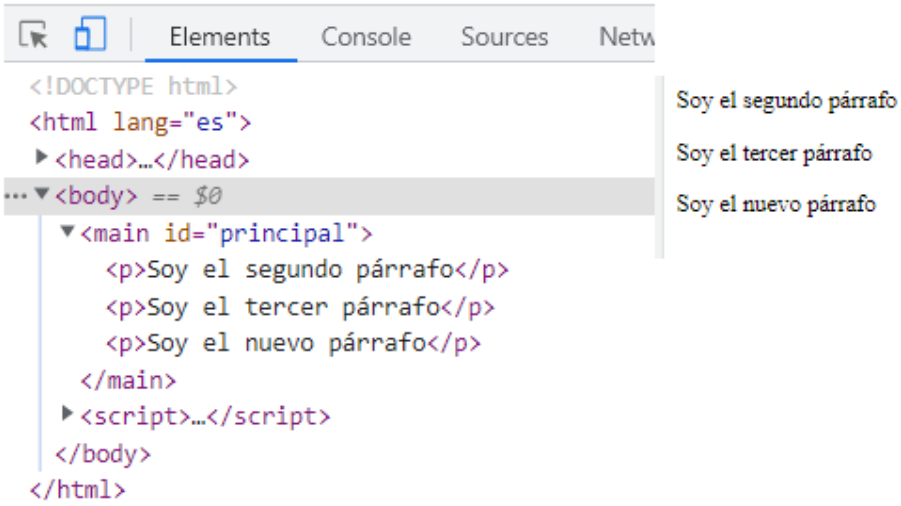
```
let nuevoP=document.createElement("p");
nuevoP.textContent="Soy el nuevo párrafo";
capa.appendChild(nuevoP);
console.log(hijos.length);
```

Escribe directamente 4. Eso significa que la propiedad children retorna una colección viva de nodos, porque se actualiza al instante con los cambios que JavaScript realiza al DOM.

Lo mismo si borramos elementos

```
capa.removeChild(capa.firstElementChild);
console.log(hijos.length);
```

Vuelve a aparecer el número 3 por consola y vemos que las capas que quedan son estas:



La colección children es una lista viva que se actualiza con los cambios del DOM. Son también métodos que devuelven colecciones vivas:

- getElementsByTagName
- getElementsByClassName
- atributes
- classList
- children

Es decir, la mayoría de métodos.

De los que no son vivos, el más preocupante es **querySelectorAll**, que es muy utilizado por su potencia para seleccionar elementos, pero que la colección de elementos que devuleve no está viva, es estática. Si queremos la lista al día deberemos ejecutarlo de nuevo. Para entender mejor el problmea veamos este código:

```
<main id="principal">
<p class="especial">Uno</p>
<p>Dos</p>
<p class="especial">Tres</p>
</main>
<p>Cuatro</p>
<p>Cinco</p>
<script>
let principal=document.getElementById("principal")
let lista1=document.getElementsByClassName("especial");
let lista2=document.querySelectorAll(".especial");
console.log(lista1.length); //Escribe 2
console.log(lista2.length); //Escribe 2
//Eliminamos el párrafo de la clase especial con el texto "Uno"
principal.removeChild(principal.children[0]);
console.log(lista1.length); //Escribe 1, está al día
console.log(lista2.length); //Escribe 2, no actualiza los cambios
```

```
</script>
```

4.10. OTRAS PROPIEDADES Y MÉTODOS DE LOS ELEMENTOS

Los objetos de tipo Element poseen numerosas propiedades, anteriormente hemos visto las fundamentales, pero hay bastantes más. Algunas son repetitivs sobre lo ya visto y otras se consideran obsoletas, enunciamos algunas de las que nos pueden ser útiles, y que no hemos visto hasta el momento.

PROPIEDAD O MÉTODO	USO
clientHeight	Altura en píxeles del elemento
clientWidth	Anchura en píxeles del elemento
clientLeft	Anchura del borde izquierdo del elemento en píxeles
clientTop	Altura del borde izquierdo del elemento en píxeles
outerHTML	A diferencia de **innerHTML**, esta propiedad devuelve el código HTML, completo del elemento, incluidas sus propias etiquetas
scrollHeight	Tamaño completo en píxeles del scroll vertical del elemento. Indica cuánto se puede desplazar en vertical el contenido del elemento haciendo scroll. Por ejemplo, si tenemos un elemento que mide 500 píxeles en vertical (**clientHeight**=500) y tenemos que puede haber 1500 pixeles de scrollHeight, eso significa que el contenido interior mide 3 veces lo que mide el elemento.
scrollWidth	Tamaño completo en píxeles del scroll horizontal del elemento.
scrollTop	Posición actual del scroll vertical en el elemento. Si el **scrollHeight** fuera de 1500 y el **scrollTop** es de 750, estamos a mitad de camino en el desplazamiento vertical. Esta propiedad adminte modificar su valor para hacer de forma manual el desplazamiento.
scrollLeft	Posición actual del scroll horizontal en el elemento. Esta propiedad adminte modificar su valor.
scroll(x,y) o bien scroll(objetoScroll)	Desplaza la página. Permite indicar el desplazamiento mediante 2 coordenadas (x horizontal e y vertical, o por un objeto de tipo scrollToOpions.
scrollBy(x,y) o bien scrollBy(objetoScroll)	Los parámetros fuincionan igual que von la propiedad anterior, pero ahora el desplazamiento es relativo respecto a la posicoón actual

5. TEMPORIZADORES

5.1. ¿QUÉ SON LOS TEMPORIZADORES?

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

Todas estas capacidades tienen que ver con 2 métodos que permiten ejecutar un determinado código (normalente mediante un función callback) cuando pase cierto tiempo.

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

5.2. SETTIMEOUT

El método setTimeout de window tiene 2 parámetros: el primero es la funión que se ejecutará cuando se cumpla el tiempo, el segundo es un valor numérico que indica el tiempo a cumplir en milisegundos. Ejemplo:

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

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

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

Ejemplo:

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

Tal cual está escrito el código anterior, jamás aparecerá el mensaje Hola porque se cancela el temporizador inmediatamente.

5.3. SETINTERVAL

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

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

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

El código es casi idéntico al que vimos anteriormente, pero el cuadro de mensaje diciendo Hola aparecerá cada 5 segundos. No dejará de aparecer hasta que no lo cancelemos.

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

Este código muestra la palabra **Hola** en un mensaje cada 5 segundos, pero en este caso se repetirá 10 veces únicamente.

6. COOKIES

6.1. ¿QUÉ SON LAS COOKIES?

Las cookies son una serie de archivos de texto que las aplicaciones web graban en el ordenador del usuario. Estos archivos contienen datos que las aplicaciones pueden volver a recuperar. Dado que http es un protocolo sin estado, cada petición http es independiente de la anterior. Las cookies se envían automáticamente en la cabecera de cada petición permitiendo a las aplicaciones recordar aspectos de peticiones anteriores gracias a los datos que han grabado las cookies.

Cada cookie puede grabar como mucho 4Kb de datos y se permiten 20 cookies por dominio. Suelen ser suficientes estas restricciones con casi cualquier tipo de aplicación.

Todos los usuarios europeos de Internet han oído hablar de las cookies, ya que en la UE hay una directiva relativa a su uso que obliga a las aplicaciones web a pedir permiso al usuario antes de poder grabar cookies. Ese aviso tiene que incluir una explicación (o un enlace a la explicación) que aclare la utilización y necesidad que la aplicación requiere de las cookies.

En realidad, esta directiva distingue entre las cookies técnicamente necesarias y las no necesarias. Las primeras son imprescindibles para el correcto funcionamiento de la aplicación. Un ejemplo de este tipo de cookies son las que graban los datos de inicio de sesión de un usuario que es clave para que ese usuario acceda a su carrito de la compra o a su espacio personal. Las cookies técnicamente necesarias, se permite que se graben directamente y que luego se pida permiso al usuario. Si el usuario deniega el permiso habría que borrarlas y, siendo imprescindibles, la aplicación dejará de trabajar con normalidad, pudiendo incluso redirigir al usuario a una página en blanco.

Las cookies que, según la UE, se consideran técnicamente no necesarias son las cookies de seguimiento y de análisis de la acción del usuario. Estas no se pueden utilizar antes de pedir permiso al usuario.

La controversia y el uso malévolo por parte de algunos servicios de las cookies, las ha creado mala fama, por lo que muchas aplicaciones para almacenar los datos no imprescindibles utilizan otras opciones como, por ejemplo, la API **localStorage** que tiene más posibilidades para almacenar datos.

En todo caso, las cookies siguen siendo la opción habitual para almacenar datos relacionados con la identificación del usuario ya que, tiene la ventaja de enviarse automáticamente como datos anexos en cada petición http, tanto del cliente como del servidor.

6.2. LECTURA Y GRABACIÓN DE COOKIES

El navegador por defecto imposibilita la opción de que puedas crear cookies en local y trabajar con ellas.

Esto significa que si quieres hacer pruebas debes hacerlo en un entorno real.

Es el objeto **document**, una vez más, es el que posee la propiedad cookie que es la que controla las cookies. A esta propiedad se asigna un texto que es el que graba en sí la cookie. La sintaxis básica es la siguiente:

```
document.cookie= "nombreCookie=valor";
```
Por ejemplo:

```
document.cookie="usuario=Gema";
```
Hemos grabado una nueva cookie con el nombre usuario y el valor Gema. Cada nueva cookie puede utilizar el mismo formato, tendría otro nombre y el valor que le queramos dar.

Si queremos modificar el valor de una cookie ya creada basta con indicar el mismo nombre e indicar el nuevo valor:

```
document.cookie="usuario=Aina";
```
Para leer el contenido basta con ver el contenido de esta propiedad. Ejemplo:

```
console.log(document.cookie);
```
El resultado podría ser:

```
» document.cookie;
← "user=Gema; lenguaje=es; fechaUltimaSesion=22/8/2022"
```

El problema en la lectura es cómo leer cada cookie por separado. En ese sentido nos puede ayudar la función **split** de los strings la cual puede dividir las cookies usando el separador entre cada una (; seguido de un espacio en blanco). Cada elemento del array resultante lo volveremos a dividir mediante el mismo método **split** para separar el nombre y el valor.

```
let arrayCookie=document.cookie.split ("; ");
for(let cookie of arrayCookie){
        let [nombre,valor]=cookie.split ("=");
        console.log(`La cookie "${nombre}" tiene el valor "${valor} "`);
}
```

```
La cookie "usuario" tiene el valor "Gema "
La cookie "lenguaje" tiene el valor "es "
La cookie "fechaUltimaSesion" tiene el valor "22/08/2022 "
```

6.3. FECHAS DE EXPIRACIÓN

Por defecto, las cookies se eliminan cuando el usuario finaliza su sesión. El fin de sesión ocurre cuando se cierra el navegador.

Pero hay cookies que se requiere que sean **persistentes**; es decir, que sus valores se mantengan entre sesión y sesión. Para ello hay que indicar una fecha de expiración futura. Esta fecha es el momento máximo en el que esa cookie sobrevive. Los navegadores eliminan las cookies que han caducado cada vez que el usuario inicia sesión en el dominio que grabó la cookie.

La fecha de expiración se debe indicar al modificar el valor de una cookie colocando un punto y coma tras el valor e indicando la palabra **expires** seguida de la fecha de expiración en formato UTC. La forma de grabar la fecha de expiración de JavaScript es obtener la fecha actual, después añadimos el tiempo de duración de la cookie. Añadir tiempo a las fechas mediante objetos de tipo **Date** nos obliga a hacerlo en milisegundos. Finalmente, la fecha ya incrementada se pasa a formato UTC.

Veamos un ejemplo:

```
//Crea un objeto de tipo fecha con la fecha actual
let hoy=new Date();
//Crea una variable tomando los milisegundos de hoy
//y sumando los milisegundos de la semana que viene
let caducidadMs=hoy.getTime() + 1000 * 60 * 60 * 24 * 7;
//Convierte los milisegundos a un objeto Date que representa la fecha
//de dentro de una semana
let caducidad=new Date(caducidadMs);
//Se coloca la fecha de expiración en formato UTC, la cookie "usuario"
document.cookie=`usuario="Gema"; expires=${caducidad.toUTCString()}`;
```

6.4. RUTA Y DOMINIO DE LAS COOKIES

Por defecto una cookie solo puede ser leída por aplicaciones del dominio que creó la cookie y que estén en el mismo directorio en el que se creó la cookie. Eso significa, por ejemplo, que, si una cookie se graba en un directorio como ***http://gemasite.net/ahorcado***, los documentos de la raíz, fuera de la carpeta de **ahorcado**, no pueden leer esas cookies. La propiedad path de las cookies permite indicar la ruta raíz desde la que cualquier aplicación puede leer esa cookie:

```
document.cookie=”usuario=Gema;path=/”;
```

Con la propiedad path, hemos indicado que la raíz desde la que cualquier aplicación puede leer la cookie es “/”, que es el nombre del directorio raíz del dominio. Es decir, en este caso, cualquier documento del dominio puede leer esa cookie, ya que se incluyen los subdirectorios de la ruta.

Con el dominio ocurre lo mismo. Si hemos grabado la cookie en un subdominio y queremos que otro subdominio lea esa cookie, por defecto no podrá. Por ello, deberemos indicar un dominio más general. Por ejemplo:

document.cookie=”teleo=si ;path=/ ;domain=gemasite.net”;
Al indicar como dominio: ***gemasite.net***, aunque hayamos grabado la cookie en un documento del subdominio ***www.gemasite.net***, un documento de ***practicas.gemasite.net*** también las podrá leer.

6.5. BORRAR COOKIES

La forma de que una aplicación elimine una de sus cookies es indicar como fecha de expiración una fecha del pasado. Ejemplo:

document.cookie=”usuario=Gema; **expires**=Sat, 01 Jan 2000 00:00:01 GMT”;
Al colocar como fecha el 1 de enero del año 2000, la cookie quedará caduca y el navegador la borrará