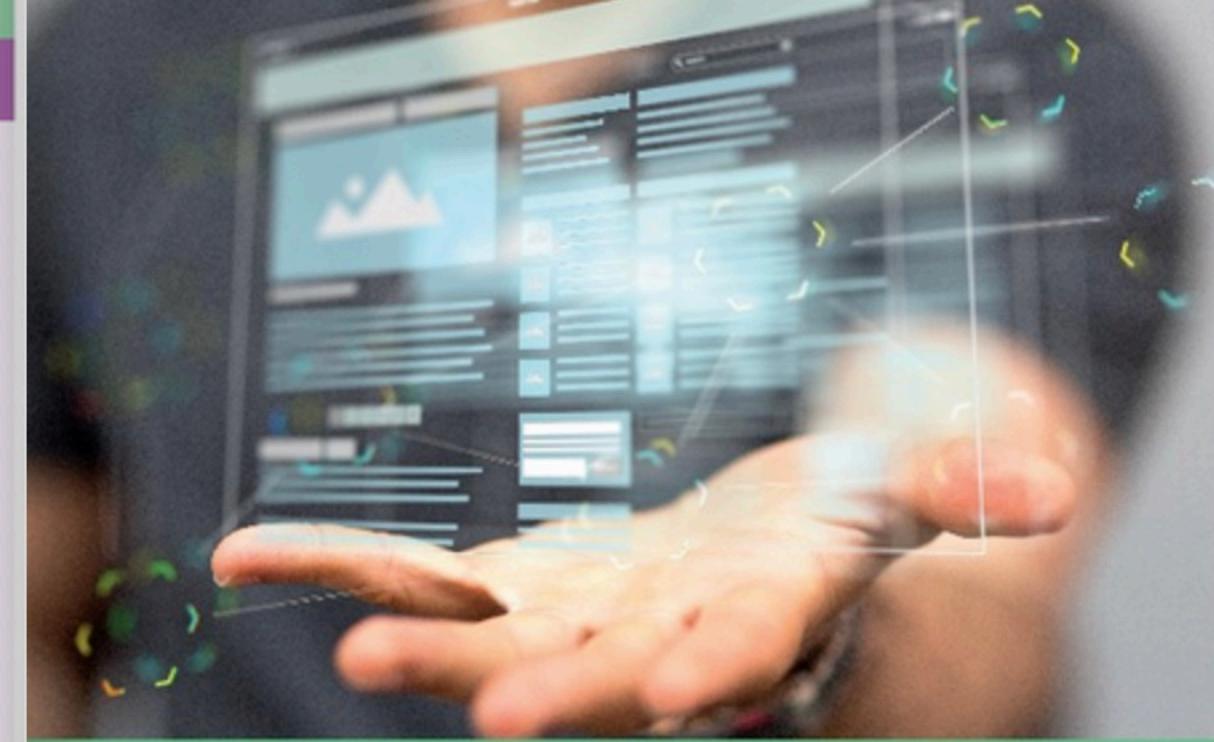


Enlaces web de interés

-  **Mozilla Developers** - https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming
(website para desarrolladores)
-  **freeCodeCamp** - <https://www.freecodecamp.org/>
(sitio web dedicado al aprendizaje de lenguajes de programación)
-  **W3Schools** - <https://www.w3schools.com/>
(sitio web dedicado a la programación)
-  **Stackoverflow** - <https://stackoverflow.com/>
(la mayor comunidad de programadores de internet)
-  **ECMAScript 2022** - <https://262.ecma-international.org/13.0/>
(última actualización de la especificación del lenguaje)
-  **W3Resource** - <https://www.w3resource.com/>
(recursos libres para programadores front-end)
-  **Jslib.dev** - <https://jslib.dev/>
(sitio web especializado en el desarrollo con JavaScript)
-  **JavaScript en Microsoft** - <https://docs.microsoft.com/es-es/javascript/>
(comunidad para el aprendizaje de JavaScript en Microsoft)



Modelo de objetos del cliente

Objetivos

- Entender la estructura del modelo de objetos del cliente.
- Analizar la relación que existe entre los diferentes objetos.
- Reconocer las diferencias entre el BOM y el DOM.
- Utilizar las propiedades y los métodos de los principales objetos del BOM.
- Crear y gestionar temporizadores.
- Aprender a seleccionar, obtener y modificar elementos del DOM.
- Describir las posibilidades que ofrece la manipulación del DOM.
- Gestionar cookies.

Contenidos

- 6.1. Un modelo, dos enfoques
- 6.2. Modelo de objetos del navegador
- 6.3. Modelo de objetos del documento

Introducción

Durante las unidades precedentes se ha invertido una cantidad importante de tiempo en aprender cómo funciona el lenguaje de referencia de la web. Era una tarea muy necesaria para conseguir el primer objetivo, alcanzar cierta destreza creando y manipulando distintas estructuras de datos. Sin embargo, todo lo anterior no tendría sentido si no se aplicara sobre el espacio de desarrollo correcto.

De la misma forma que un deportista necesita conocer su físico, saber sus límites y entender hasta dónde puede llegar, para poder sacar el mayor rendimiento sobre el terreno de juego, los programadores necesitan conocer el lugar sobre el que van a desplegar todo su potencial de programación. Ese lugar tan especial es el documento en el que se representan las aplicaciones web.

En esta unidad se estudia ese espacio, su estructura, cómo está configurado y qué elementos ofrece JavaScript para modelarlo.

6.1. Un modelo, dos enfoques

En muchas ocasiones, cuando se hace referencia al modelo de objetos del cliente, se generaliza con el DOM (modelo de objetos del documento), como si todos los objetos disponibles formaran parte del mismo modelo. Se trata de un error común que no proporciona una comprensión completa de cómo funcionan todos los elementos del cliente.

El DOM no lo es todo, mucho más sabiendo que técnicamente **el DOM no es más que una parte del BOM (modelo de objetos del navegador)**; al fin y al cabo, el navegador es toda la interfaz que tienen los usuarios a mano para usar una aplicación web.

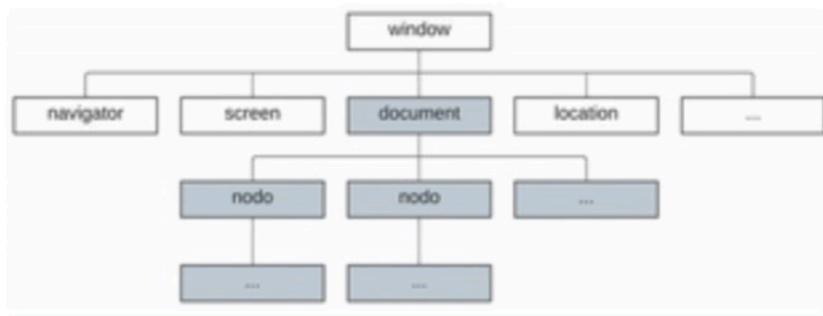


Figura 6.1. Jerarquía de objetos del modelo.

Dicho lo anterior, es importante aclarar que en esta obra se separarán porque, desde el punto de vista funcional, el DOM es la parte más importante a la hora de trabajar con JavaScript y es deseable dejar muy bien delimitadas las funciones de cada modelo.

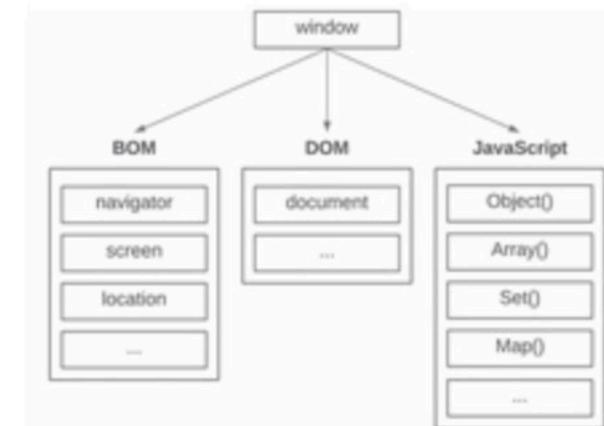


Figura 6.2. División conceptual para facilitar el estudio de JavaScript.

Por decirlo de una manera entendible por todos, el BOM da acceso a todo lo que tiene que ver con el navegador, y el DOM con el sitio web que en cada momento se representa en el navegador.

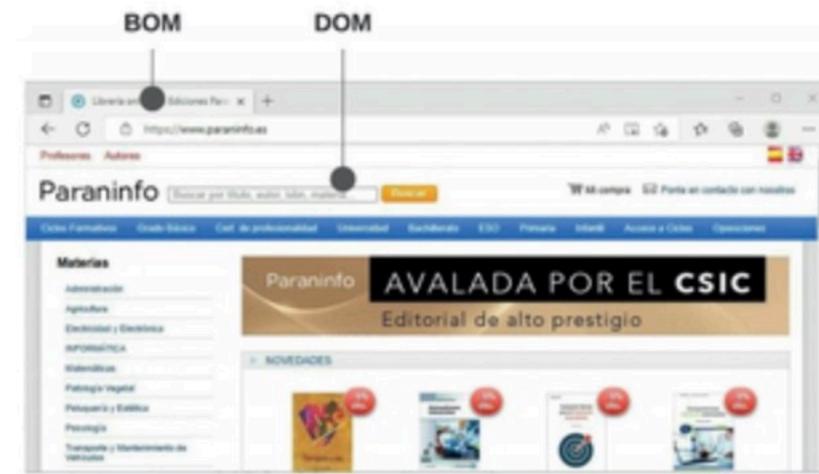


Figura 6.3. Identificación visual sencilla de entender de cada modelo.

En las siguientes páginas de esta unidad se repasa al detalle cada uno de los elementos de ambos modelos.

6.2. Modelo de objetos del navegador

El modelo de objetos del navegador, BOM, básicamente permite que JavaScript se comunique con el navegador que utilizan los usuarios para usar una aplicación web.

No existen estándares oficiales que lo normalicen; sin embargo, debido a que una extensa generalidad de los navegadores modernos ha implementado casi todos los métodos y propiedades con las que trabaja JavaScript, con frecuencia se los denomina propiedades y métodos del BOM.

En los siguientes apartados se verán cuáles son y cómo funcionan los objetos más importantes del modelo.

6.2.1. Objeto window

El objeto **window** representa la ventana del navegador. Todos los objetos, funciones y variables globales de JavaScript se convierten automáticamente en miembros de este objeto, de manera que las variables serán las propiedades, y las funciones los métodos. Tal y como se ha avanzado, hasta el DOM es una propiedad del objeto **window**.

Una de las principales utilidades que ofrece este objeto padre es poder obtener el tamaño de la ventana del navegador:

```
let ancho = window.innerWidth;
let alto = window.innerHeight;
console.log(`Ancho x Alto (en píxeles): ${ancho} x ${alto}`);
```

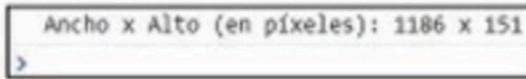


Figura 6.4. Valores devueltos por la consola del navegador.

Las propiedades disponibles no se quedan aquí; existen muchas otras, pero su funcionamiento desde el punto de vista del código es idéntico. Se anima al lector a que acuda a la documentación oficial del objeto para descubrirlas. Aquí, por cuestiones de espacio, no pueden cubrirse todas.

Para saber más

Con el siguiente enlace o código QR se accede a la lista de propiedades del objeto **window**:

<https://developer.mozilla.org/es/docs/Web/API/Window#propiedades>



Además, también tiene una lista muy útil de métodos a disposición del programador. En la Tabla 6.1 se recogen algunos de los más importantes.

Tabla 6.1. Métodos más útiles del objeto window

Método	Utilidad
alert()	Abre una ventana de alerta con un mensaje y un botón para aceptar.
blur()	Retira el foco de una ventana.
close()	Cierra la ventana.
confirm()	Abre una ventana de diálogo con dos botones: Aceptar y Cancelar . Devuelve true si se pulsa Aceptar , y false en caso contrario.
focus()	Asigna el foco a una ventana.
moveBy()	Mueve la ventana la cantidad de píxeles que se indique desde su posición actual.
moveTo()	Mueve la ventana a las coordenadas en píxeles que se le indiquen.
open()	Abre una URL en una nueva pestaña.
print()	Abre el cuadro de diálogo de impresión para imprimir una ventana.
prompt()	Abre un cuadro de diálogo para que el usuario introduzca datos y dos botones: Aceptar y Cancelar . Devuelve el valor introducido por el usuario si se pulsa en Aceptar , y null en otro caso.
resizeBy()	Redimensiona la ventana usando los incrementos o decrementos que se indican en píxeles.
resizeTo()	Redimensiona la ventana al tamaño indicado.
scrollBy()	Desplaza el contenido de la ventana el número de píxeles que se le indique.
scrollTo()	Desplaza el contenido de la ventana hasta la posición en píxeles que se indica.
stop()	Cancela la carga de la página.

Nota técnica

En el contexto de las aplicaciones web cuando se habla de foco se hace referencia al elemento que en cada momento tiene la capacidad de ser modificado o con el que se puede interactuar. Que una ventana reciba el foco, quiere decir que esa es la ventana activa en ese momento. Si desde un programa se pone el foco en un elemento de la página, las operaciones que se realicen en ese momento se realizarán sobre ese elemento activo.

Actividad resuelta 6.1

Métodos del objeto window

Escribe un programa que le pida al usuario qué porcentaje quiere aplicar para redimensionar la ventana. El programa debe pedir confirmación y solo en caso de que se acepte, se redimensionará la ventana el porcentaje indicado. Es necesario, igualmente, ir mostrando por la consola qué va ocurriendo en cada momento:



Solución

```
let prmpt = window.prompt("Indica una cantidad en píxeles, por favor:");
if (prmpt != null) {
    console.log("El usuario pulsó Aceptar en la introducción de datos");
    let cnfrm = window.confirm('¿Estás seguro de aplicar ${prmpt}px?');
    if (cnfrm === false) {
        console.log("El usuario no estaba seguro");
    }
    else {
        console.log("El usuario estaba seguro");
        window.resizeBy(prmpt,prmpt);
        console.log(`Ventana redimensionada ${prmpt}x${prmpt}px`);
    }
}
else {
    console.log("El usuario canceló la introducción de datos");
}
```

```
infoNavegador+="plugins: "+navigator.plugins+"\n";
infoNavegador+="storage: "+navigator.storage+"\n";
infoNavegador+="userAgent: "+navigator.userAgent+"\n";
console.log(infoNavegador);
```

```
clipboard: [object Clipboard]
cookieEnabled: true
geolocation: [object Geolocation]
language: es-ES
onLine: true
plugins: [object PluginArray]
storage: [object StorageManager]
userAgent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36
(modelo.js:9)
```

Figura 6.5. Resultado en consola de la consulta de propiedades del objeto navigator.

También proporciona algunos métodos, como `javaEnabled()`, para comprobar si el *plugin* de Java está activado, o `vibrate()`, para hacer vibrar al dispositivo en el que se ejecuta. Además, algunos navegadores extienden este objeto aportando otros métodos muy útiles.

6.2.2. Objeto navigator

Se trata del objeto de `window` que contiene la información sobre el navegador que está utilizando el usuario para interactuar con la aplicación web. Puede utilizarse directamente sin utilizar el prefijo `window`. En la Tabla 6.2 se muestran algunas de las propiedades más interesantes de este objeto.

Tabla 6.2. Propiedades más útiles del objeto navigator

Propiedad	Utilidad
<code>clipboard</code>	Devuelve un objeto para trabajar con el portapapeles.
<code>cookieEnabled</code>	Devuelve <code>true</code> si las <i>cookies</i> están habilitadas.
<code>geolocation</code>	Devuelve un objeto con la localización del usuario.
<code>language</code>	Devuelve el idioma del navegador.
<code>onLine</code>	Devuelve <code>true</code> si el navegador está <i>online</i> .
<code>plugins</code>	Devuelve información sobre los <i>plugins</i> instalados.
<code>storage</code>	Devuelve un objeto que permite trabajar con la API de almacenamiento de datos persistentes.
<code>userAgent</code>	Devuelve información sobre el navegador del usuario.

```
let infoNavegador="clipboard: "+navigator.clipboard+"\n";
infoNavegador+="cookieEnabled: "+navigator.cookieEnabled+"\n";
infoNavegador+="geolocation: "+navigator.geolocation+"\n";
infoNavegador+="language: "+navigator.language+"\n";
infoNavegador+="onLine: "+navigator.onLine+"\n";
```

Tabla 6.3. Propiedades más útiles del objeto screen

Propiedad	Utilidad
<code>availHeight</code>	Especifica la altura de la pantalla, en píxeles.
<code>availWidth</code>	Especifica la anchura de la pantalla, en píxeles.
<code>colorDepth</code>	Devuelve la profundidad de color de la pantalla.
<code>height</code>	Altura completa de la pantalla del usuario.
<code>orientation</code>	Devuelve un objeto con información sobre la orientación de la pantalla. Es una propiedad experimental que continuará, pero podría modificar su funcionamiento en el futuro próximo.
<code>pixelDepth</code>	Devuelve la profundidad de bits de la pantalla.
<code>width</code>	Anchura completa de la pantalla del usuario.

```
let infoPantalla="availHeight: "+screen.availHeight+"\n";
infoPantalla+="availwidth: "+screen.availwidth+"\n";
infoPantalla+="height: "+screen.height+"\n";
infoPantalla+="width: "+screen.width+"\n";
```

```
infoPantalla+="colorDepth: "+screen.colorDepth+"\n";
infoPantalla+="pixelDepth: "+screen.pixelDepth+"\n";
infoPantalla+="orientation: "+screen.orientation+"\n";
infoPantalla+="orientation.type: "+screen.orientation.type+"\n";
console.log(infoPantalla);
```

```
availHeight: 875
availwidth: undefined
height: 875
width: 1400
colorDepth: 24
pixelDepth: 24
orientation: [object ScreenOrientation]
orientation.type: landscape-primary
```

Figura 6.6. Resultado en consola de la consulta de propiedades del objeto screen.

Dispone también de otras propiedades que no se incluyen por estar obsoletas, en vías de eliminación, o por no ser compatibles con todos los navegadores, como, por ejemplo, `availLeft`, `availTop`, `left`, `top`.

Tampoco aporta métodos propios que sean estándares. Lo que sí hace es implementar métodos heredados de otras clases, que se verán al repasar estas últimas.

6.2.4. Objeto location

Es otro de los objetos interesantes de `window`. En este caso representa la localización (URL) del objeto al que está vinculado, es decir, la URL de la aplicación web. A esta propiedad se puede acceder desde el objeto `window` y desde el objeto `document`, ya que ambas son referencias al mismo objeto.

La propiedad más interesante de este objeto es `href`, que devuelve la URL de la aplicación. Pero también redirecciona a la URL indicada simplemente asignándole una nueva dirección:

```
console.log(location.href); // Devuelve la URL de la aplicación
location.href = "https://www.google.es"; // Carga google.es
```

Adicionalmente, se puede obtener y manipular otra información parcial de la localización de la aplicación (Tabla 6.4).

Tabla 6.4. Propiedades más útiles del objeto location

Propiedad	Utilidad
<code>hash</code>	Obtiene # seguido del marcador indicado en la URL. Por ejemplo, si la URL es https://paraninfo.es/libros/tecnicos#javascript devolvería <code>#javascript</code> .
<code>host</code>	Una cadena que contiene <code>nombrehost:puerto</code> de la URL. Por ejemplo: <code>paraninfo.es:886</code> .

Propiedad	Utilidad
<code>hostname</code>	El dominio de una URL: por ejemplo, <code>paraninfo.es</code> .
<code>origin</code>	Contiene la forma canónica del origen de una URL: protocolo, puerto y nombre de <code>host</code> .
<code>password</code>	Almacena la contraseña que vaya en la URL, si lo hay.
<code>pathname</code>	Saca la ruta de ficheros a partir del <code>host</code> en la URL: por ejemplo, <code>/libro/html/modelo.html</code> .
<code>port</code>	Obtiene el número del puerto de la URL.
<code>protocol</code>	Captura el esquema del protocolo de la URL, incluyendo los dos puntos: por ejemplo, <code>http:</code>
<code>search</code>	Contiene la cadena de consulta de una URL. Por ejemplo: si la URL es https://paraninfo.es/buscar.php?autor=Gonzalez&keyword=js , almacenaría <code>?autor=Gonzalez&keyword=js</code> .
<code>username</code>	Almacena el nombre de usuario que vaya en la URL, si lo hay.

También ofrece algunos métodos interesantes, como `reload()`, que recarga la página. Recibe un parámetro que cuando es `true` hace que la página siempre sea recargada desde el servidor. Si es `false`, o no se especifica, la página se puede recargar desde la caché.

6.2.5. Objeto history

Es el objeto destinado a almacenar el historial de páginas visitadas. Su utilidad más interesante reside en el método `go()`, que permite moverse por el historial. De esta manera, `go(-1)` sería equivalente a `back()` y llevaría a la página anterior; y `go(1)` sería equivalente a `forward()` y llevaría a la página siguiente.

6.2.6. Temporizadores

El objeto `window` también ofrece algunos métodos que permiten gestionar el tiempo en el navegador, una utilidad muy interesante que resuelve muchas tareas de forma sencilla.

En concreto son cuatro métodos: `setTimeout()` y `setInterval()` y sus correspondientes `clearTimeout()` y `clearInterval()`. La funcionalidad que se busca cubrir es tener la capacidad de ejecutar código cuando pase cierto tiempo. En los siguientes apartados se estudia su utilización.

setTimeout()

Este método acepta dos parámetros, el primero es el `callback` que se ejecutará y el segundo el tiempo en milisegundos que hay que esperar antes de que se ejecute:

```
let t1 = new Date();
console.log(t1.getHours()+":"+t1.getMinutes()+":"+t1.getSeconds());
```

```
window.setTimeout(()=>{
    let t2 = new Date();
    console.log(t2.getHours()+":"+t2.getMinutes()+":"+t2.getSeconds());
},5000);
```

Con la pieza de código anterior se saca en pantalla la hora a la que se inicia la ejecución del script, y luego se crea un temporizador que deberá ejecutarse tras una espera de cinco segundos. El callback del temporizador simplemente muestra en pantalla la hora que es. La salida en consola permite comprobar que efectivamente ha ocurrido lo que se esperaba (Figura 6.7).



Figura 6.7. El callback ejecutándose cinco segundos después.

En los programas se pueden tener todos los temporizadores que se desee, pero ¿cómo se identifican en caso de realizar alguna acción posterior sobre alguno de ellos? Capturando el valor devuelto por `window.setTimeout()`, que es precisamente su identificador. De esta manera, puede ejecutarse `clearTimeout(identificador)` y el callback de `setTimeout()` nunca se ejecutará.

```
let t1 = new Date();
console.log(t1.getHours()+":"+t1.getMinutes()+":"+t1.getSeconds());
identificador = window.setTimeout(()=>{
    let t2 = new Date();
    console.log(t2.getHours()+":"+t2.getMinutes()+":"+t2.getSeconds());
},5000);
clearTimeout(identificador);
console.log("Programa terminado");
```

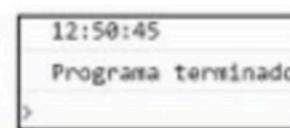


Figura 6.8. El callback cancelado antes de finalizar la cuenta del temporizador.

Como se puede ver, el callback nunca se ha llegado a ejecutar, porque la cancelación del temporizador llegó antes de que transcurrieran los cinco segundos.

Actividad propuesta 6.1

Tiempo de vida de un enlace

Crea un fragmento de HTML con las etiquetas que deseas, pero que contenga dos enlaces. A continuación, crea un temporizador que elimine el primer enlace tras haber transcurrido cinco segundos. Tras la eliminación del primer enlace debes crear otro temporizador que se activará a los diez segundos y que eliminará el segundo enlace.

setInterval()

Este método es idéntico al anterior, pero con un único matiz adicional: el `callback` se ejecuta cada vez que transcurre el tiempo indicado, no una sola vez como en el caso de `setTimeout`.

Es posible estar pensando en lo siguiente: si no se captura su identificador y nunca se invoca a `clearInterval()`, el `callback` se ejecutará indefinidamente cada vez que transcurre el tiempo que se le haya indicado. Efectivamente, hasta que no se cancele seguirá haciendo su trabajo una y otra vez.

```
let t1 = new Date();
console.log(t1.getHours()+":"+t1.getMinutes()+":"+t1.getSeconds());
identificador = window.setInterval(()=>{
    let t2 = new Date();
    console.log(t2.getHours()+":"+t2.getMinutes()+":"+t2.getSeconds());
},3000);
console.log("Programa terminado");
```

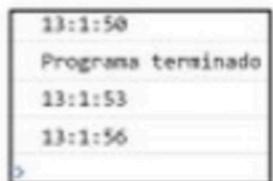


Figura 6.9. El callback ejecutándose indefinidamente cada tres segundos.

Al observar la salida de este programa (que se ha abortado manualmente), queda patente una cuestión importante que a veces se pasa por alto. Cuando se establece el temporizador, la ejecución del programa no se detiene en ese punto, sino que continúa con independencia de que el temporizador ejecute su `callback` indefinidamente tantas veces como la lógica del programa permita.

Actividad resuelta 6.2

Hasta el infinito, solo si tú quieres

Escribe un programa que le muestre un recordatorio al usuario cada cinco segundos. Mientras el usuario pulse en **Aceptar**, el recordatorio continuará. Cuando el usuario pulse en **Cancelar**, el recordatorio se anulará.

Solución

```
let identificador = window.setInterval(()=>{
    let respuesta = window.confirm("Tienes cita con el dentista. ¿Te lo sigo recordando?");
    if (respuesta === false){
        window.clearInterval(identificador);
        console.log("Programa terminado.");
    } else {
        console.log("Seguir recordando.");
    },5000);
```

6.3. Modelo de objetos del documento

El modelo de objetos de un documento (DOM) es un estándar del W3C (*World Wide Web Consortium*) que permite que los programas accedan y actualicen dinámicamente el contenido, la estructura y el estilo de un documento.

En el contexto de una aplicación web del lado cliente, los documentos a los que se refiere serán principalmente HTML (aunque son aplicables a otros, como XML). Por ello, con JavaScript es posible:

- Cambiar todos los elementos HTML de la página.
- Cambiar todos los atributos HTML de la página.
- Cambiar todos los estilos CSS de la página.
- Eliminar elementos y atributos HTML existentes.
- Agregar nuevos elementos y atributos HTML.
- Reaccionar a todos los eventos HTML existentes en la página.
- Crear nuevos eventos HTML en la página.

Un segundo concepto muy importante que se debe asimilar es la naturaleza arbórea del documento. HTML se comporta estructuralmente como un árbol. Cada elemento es un nodo y según el nivel de profundidad se establecen relaciones de parentesco, de manera que, si un nodo cuelga de otro, el más profundo es hijo, el inmediatamente superior es el padre y los que están al mismo nivel y comparten padre, son, evidentemente, hermanos.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
    <script src="DOM.js"></script>
    <title>Analizando el DOM</title>
  </head>
  <body>
    <nav>
      <a href="#">Enlace 1</a>
    </nav>
    <section>
      <h1>Primeros pasos</h1>
      <p>Lorem ipsum dolor sit amet...</p>
    </section>
  </body>
</html>
```

En el código HTML anterior se ve un documento típico que contiene un **head** con tres elementos (**meta**, **script** y **title**) y un **body** del que cuelgan dos elementos (**nav** y **section**), cada uno de ellos con otros elementos descendientes. Teniendo esto presente, su árbol asociado es el que se muestra en la Figura 6.10.

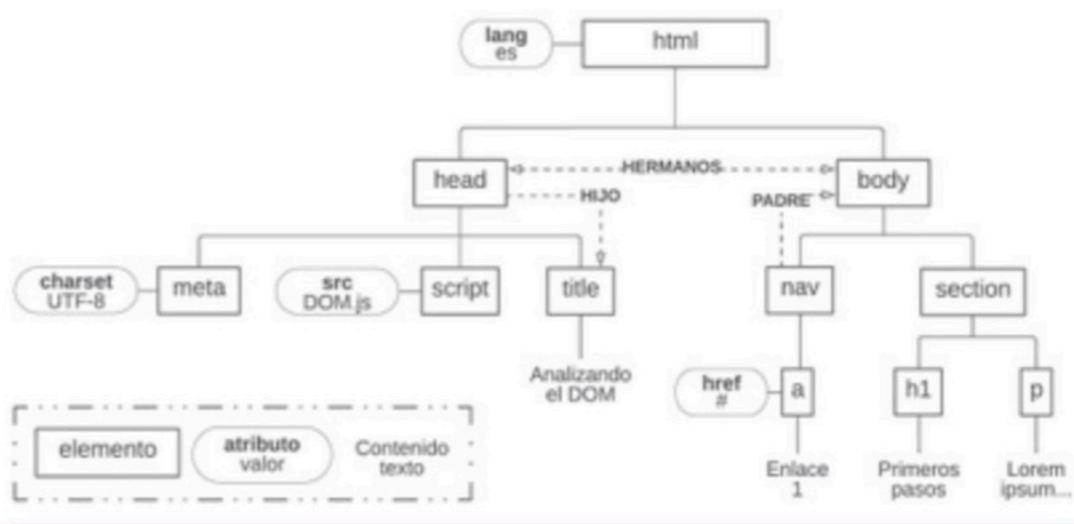


Figura 6.10. Árbol con relaciones de parentesco entre sus nodos.

Sabiendo que el DOM se construye de esta forma, el objeto que contiene toda esta información es **document** (hay que recordar que **document** también es una propiedad de **window**), que será siempre el punto de partida para seleccionar y modificar elementos de las páginas.

Pero antes de eso, y tomando como referencia la Figura 6.10, se observa que existen distintos **tipos de nodos** (y ahí no están todos). Por eso, se necesita alguna herramienta que permita conocer con qué tipo de nodo se está tratando. La propiedad **nodeType** se encarga de ese trabajo. Al consultar esa propiedad sobre un nodo seleccionado, se verá de qué tipo es:

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

La lista de posibles valores de **nodeType** se recoge en la Tabla 6.5.

Tabla 6.5. Tipos de nodos y su valor y constante asociados

Valor	Nodo	Constante asociada
1	Elemento	ELEMENT_NODE
2	Atributo*	ATTRIBUTE_NODE
3	Texto	TEXT_NODE
4	CDATA*	CDATA_SECTION_NODE
5	Referencia a entidad*	ENTITY_REFERENCE_NODE
6	Entidad*	ENTITY_NODE
7	Instrucción de procesado	PROCESSING_INSTRUCTION_NODE
8	Comentario	COMMENT_NODE
9	Documento	DOCUMENT_NODE

Valor	Nodo	Constante asociada
10	Tipo de documento	DOCUMENT_TYPE_NODE
11	Fragmento de código	DOCUMENT_FRAGMENT_NODE
12	Anotación*	NOTATION_NODE

*Actualmente obsoletos, no se recomienda su uso en sitios web nuevos.

Asociadas a `nodeType` están las propiedades `nodeName` (el nombre del nodo) y `nodeValue`, que es el valor del nodo (devuelve `null` cuando el nodo es el documento entero, un fragmento, el tipo del documento, un elemento, una referencia a entidad o una anotación).

El siguiente paso, tras conocer los tipos de nodos disponibles en el DOM, es saber cómo seleccionarlos para poder trabajar con ellos.

6.3.1. Selección de elementos

El objeto `document` ofrece múltiples vías para seleccionar nodos del DOM. Las cuatro opciones más interesantes son las que se estudian a continuación.

Identificadores

`getElementById` permite seleccionar nodos por medio del atributo `id` de sus elementos. Devuelve el nodo si lo encuentra, o `null` en caso contrario. El siguiente fragmento HTML muestra cómo funciona:

```
<span id="CEO">Joaquín Luque.</span>
<span id="CTO">Lucía Valdivia.</span>
<span id="CIO">Marisa Pons.</span>
```

En esta pieza de código HTML hay tres elementos `span`, cada uno de ellos identificado con un `id` y cuyo contenido es el nombre de una persona.

```
let ceo = document.getElementById("CEO");
console.info(ceo);
console.log(ceo.nodeType);
console.log(ceo.nodeName);
console.log(ceo.nodeValue);
```

```
<span id="CEO">Joaquín Luque.</span>
1
SPAN
null
>
```

Figura 6.11. Salida de la selección con identificador e información del nodo.

Etiquetas

`getElementsByName` permite seleccionar nodos por medio del nombre de su etiqueta. A diferencia del anterior, no devolverá un solo nodo, sino un `NodeList` o lista de nodos.

```
let equipo = document.getElementsByName("span");
console.info(equipo);
console.log(equipo.length);
console.log(equipo[1]);
```

```
▼ HTMLCollection(3) [span#CEO, span#CTO, span#CIO, CEO: span#CEO, CTO: span#CTO, CIO: span#CIO]
  ▶ 0: span#CEO
  ▶ 1: span#CTO
  ▶ 2: span#CIO
  ▶ CEO: span#CEO
  ▶ CTO: span#CTO
  ▶ CIO: span#CIO
  ▶ length: 3
  ▶ [[Prototype]]: HTMLCollection
3
<span id="CTO">Lucía Valdivia.</span>
model.js:4
```

Figura 6.12. Contenido de un objeto NodeList.

Observando la salida, se puede ver el contenido de un `NodeList`, al que se le pueden aplicar algunas utilidades de los arrays, como conocer la cantidad de elementos (`equipo.length`) o acceder a ellos por medio del operador corchetes (`equipo[1]`). También pueden usarse todos los bucles `for` vistos para recorrer arrays. Este concepto debe aplicarse con cuidado, porque no son aplicables todos los métodos disponibles en arrays, como por ejemplo `slice`, `join` o `sort`. Para ello, siempre puede convertirse un `NodeList` en un array puro usando el operador de propagación (...).

Actividad propuesta 6.2

El penúltimo párrafo

Crea un fragmento de HTML que contenga cinco párrafos y cuyas etiquetas no tengan ningún atributo. A continuación, muestra en la consola el contenido del penúltimo párrafo.

Clases

`getElementsByClassName` es idéntico en su funcionamiento al anterior, pero en este caso selecciona los elementos por el valor de su atributo `class`.

```
<span id="CEO" class="jefatura">Joaquín Luque.</span>
<span id="CTO" class="direccion">Lucía Valdivia.</span>
<span id="CIO" class="direccion">Marisa Pons.</span>
```

Se ha modificado el fragmento HTML de partida para alterar sus identificadores y añadir un nuevo atributo **class** que permite aplicarles distintos estilos y que también puede utilizarse como selectores CSS.

```
let directores = document.getElementsByClassName("direccion");
// devuelve el segundo y tercer span
```

Selectores CSS

Existen varias alternativas para **seleccionar nodos usando selectores CSS**, pero el más importante y el que cubre toda la funcionalidad es **querySelectorAll**. Hay que indicarle un selector CSS como parámetro y siempre devolverá una lista de nodos que cumplen con los criterios de la selección. Es importante escribir correctamente la cadena de consulta para evitar que se produzcan errores.

```
<span id="CEO" class="jefatura">
    <a class="god_link" href="#">Joaquín Luque.</a>
</span>
<span id="CTO" class="direccion">
    <a class="top_link" href="#">Lucía Valdivia.</a>
</span>
<span id="CIO" class="direccion">
    <a class="top_link" href="#">Marisa Pons.</a>
</span>
```

Ahora, el fragmento HTML se ha extendido para incorporar dentro de cada **span** un enlace con dos atributos, **class** y **href**.

```
let enlaces = document.querySelectorAll("span > a");
let primerDirector = document.querySelectorAll("a.top_link")[0];
let errorInducido = document.querySelectorAll("span > p");
console.info(enlaces);
console.info(primerDirector);
console.info(errorInducido);
```

```
> NodeList(3) {a.god_link, a.top_link, a.top_link}
  <a class="top_link" href="#">Lucía Valdivia.</a>
> NodeList []
>
```

Figura 6.13. Distintos formatos de la salida al usar querySelectorAll.

Actividad propuesta 6.3

El HTML perdido

Crea una jerarquía de etiquetas HTML que consiga devolver un nodo para la siguiente instrucción JavaScript: `document.querySelectorAll("p.pp > span.lt")[3];`

6.3.2. Manipulación del DOM

De nada serviría adquirir gran destreza seleccionando nodos del DOM, de no ser capaces de obtener los valores que almacenan ni modificarlos según unos intereses. En primer lugar se verá cómo moverse por el DOM, para así conocer al detalle cómo seleccionar aquello que más interese. A continuación se analizarán todas las operaciones que pueden realizarse con los elementos y, finalmente, se estudiarán las posibilidades que ofrecen los atributos.

Moverse por el DOM

El DOM, como se dijo al inicio de la unidad, no es más que un árbol de nodos por el que es posible moverse con total libertad. Conocer las estrategias de acceso a sus distintos nodos es fundamental para escribir programas eficientes. Una de las tareas más comunes trata sobre **cómo obtener los hijos de un elemento**.

```
<ul id="indiceSeccion">
    <li class="menuSeccion">
        <a href="#">Moverse por el DOM</a>
    </li>
    <li class="menuSeccion">
        <a href="#">Manipular elementos</a>
    </li>
    <li class="menuSeccion">
        <a href="#">Manipular atributos</a>
    </li>
</ul>
```

El fragmento de código HTML representa una lista desordenada (con un identificador) formada por tres elementos. Cada elemento tiene aplicada una clase, y contiene un enlace con un único atributo.

Todos los elementos del DOM tienen asociada una propiedad (**childNodes**) que permite acceder a la lista de sus nodos hijos. Así, puede obtenerse esa lista del elemento **** y luego, por medio de un recorrido, acceder a sus hijos:

```
let indice = document.getElementById("indiceSeccion");
for (let i=0; i<indice.childNodes.length; i++) {
    console.log(indice.childNodes[i].nodeName);
}
```

Observando la salida de este programa, se ve que para cada hijo muestra lo que se observa en la Figura 6.14.

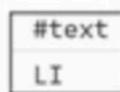


Figura 6.14. Nodos que forman parte de cada elemento.

Esto quiere decir que, para el DOM, el texto que contiene un elemento también es un nodo. Si solo se quieren obtener los elementos, se puede usar `children` en lugar de `childNodes`. Esta misma diferenciación se hace con el resto de las propiedades útiles que se tienen a mano para navegar por el DOM (Tabla 6.6).

Tabla 6.6. Propiedades más interesantes para navegar por el DOM

Propiedad	Utilidad
<code>parentNode</code>	Selecciona el nodo padre.
<code>childElementCount</code>	Devuelve el número de elementos hijos.
<code>firstChild</code>	Selecciona el primer nodo hijo.
<code>firstElementChild</code>	Selecciona el primer elemento hijo.
<code>lastChild</code>	Selecciona el último nodo hijo.
<code>lastElementChild</code>	Selecciona el último elemento hijo.
<code>previousSibling</code>	Selecciona el nodo hermano anterior.
<code>previousElementSibling</code>	Selecciona el elemento hermano anterior.
<code>nextSibling</code>	Selecciona el nodo hermano siguiente.
<code>nextElementSibling</code>	Selecciona el elemento hermano siguiente.

Actividad resuelta 6.3

Hermanos

Crea una lista ordenada de tres elementos, selecciona el elemento del medio y muestra en consola su hermano anterior y el siguiente.

Solución

HTML

```
<ol>
  <li>Hermano anterior.</li>
  <li>Elemento central.</li>
  <li>Hermano siguiente.</li>
</ol>
```

JAVASCRIPT

```
let mediano = document.getElementsByTagName("li")[1];
console.log(mediano.previousElementSibling);
console.log(mediano.nextElementSibling);
```

Manipulación de elementos

Cuando se comienza a obtener el contenido de los elementos existe un error común que consiste en no diferenciar las propiedades `textContent` e `innerHTML`. La primera de ellas obtiene el texto sin tener en cuenta las otras posibles etiquetas que contiene, de hecho ni siquiera las muestra. Sin embargo, la segunda sí que mostrará el contenido completo, incluyendo las posibles etiquetas que contenga. A continuación se muestra un ejemplo:

```
<section>
  Manipular elementos <i>parece</i> sencillo<br />
  pero <strike>debe</strike> puede complicarse bastante.
</section>
```

En esta ocasión la pieza de HTML representa una sección de una web que contiene texto que, a su vez, incorpora algunas etiquetas HTML para darle cierto formato (es una práctica desaconsejable, puesto que los estilos deberían ir en las hojas de estilo CSS, pero se usa en este punto solo con fines pedagógicos).

```
console.log(document.getElementsByTagName("section")[0].textContent);
console.log(document.getElementsByTagName("section")[0].innerHTML);
```

Manipular elementos parece sencillo
pero debe puede complicarse bastante.

Manipular elementos <i>parece</i> sencillo

pero <strike>debe</strike> puede complicarse bastante.

Figura 6.15. Diferencias entre `textContent` e `innerHTML`.

Esta misma distinción es necesaria tenerla en mente al modificar el contenido. Para hacerlo simplemente se asigna un contenido nuevo a la selección:

```
document.getElementsByTagName("section")[0].textContent = "el <u>nuevo</u> contenido";
document.getElementsByTagName("section")[0].innerHTML = "el <u>nuevo</u> contenido";
```

Pero siempre sabiendo que las etiquetas que se incluyan en `textContent` no serán «entendidas» como HTML, algo que sí ocurrirá con `innerHTML`.

Se ha visto cómo seleccionar, obtener elementos y modificarlos, pero falta saber cómo crearlos. Los métodos que permiten crear elementos nuevos son `createElement`, para crear nodos de tipo elemento, y `createTextNode`, para crear nodos de texto nuevos. En ambos casos, su existencia no implica que sean visibles en el DOM, puesto que falta indicar en qué posición del árbol colocarlos:

```
let elementoNuevo = document.createElement("p");
let nodoTextoNuevo = document.createTextNode("Texto del nodo");
```

Para conseguir **insertar un nodo nuevo** en el árbol, una primera aproximación es hacerlo con **appendChild**, añadiendo el nodo al final de los hijos:

```
<ul id="lista">
    <li>Primer elemento hijo</li>
    <li>Segundo elemento hijo</li>
</ul>
```

El código HTML representa una lista desordenada identificada como **lista**, que contiene dos elementos.

```
let nuevoElemento = document.createElement("li");
nuevoElemento.innerHTML = "Tercer elemento";
document.getElementById("lista").appendChild(nuevoElemento);
```

- Primer elemento hijo
- Segundo elemento hijo
- Tercer elemento

Figura 6.16. Aspecto de la lista tras insertar al final.

Sin embargo, no siempre es deseable que el nuevo nodo ocupe la última posición. Para todos esos casos, se puede recurrir a **insertBefore**. Este nuevo método funciona con dos parámetros, el primero es el nodo que se va a añadir, y el segundo el nodo hijo que quedaría por debajo.

Partiendo del mismo código HTML anterior, ahora se inserta un nodo en la segunda posición:

```
// crear el nodo
let nuevoElemento = document.createElement("li");
nuevoElemento.innerHTML = "Último elemento insertado";
// obtener referencia a la posición
let nodoReferencia = document.querySelectorAll("li:nth-of-type(2)")[0];
// obtener el nodo padre
let nodoPadre = nodoReferencia.parentNode;
// insertar el elemento
nodoPadre.insertBefore(nuevoElemento,nodoReferencia);
```

- Primer elemento hijo
- Último elemento insertado
- Segundo elemento hijo

Figura 6.17. Aspecto de la lista tras insertar en la posición indicada.

También cabe la posibilidad de insertar un nodo, pero reemplazando al que ocupaba esa posición:

```
nodoPadre.replaceChild(nuevoElemento,nodoReferencia);
```

- Primer elemento hijo
- Último elemento insertado

Figura 6.18. Aspecto de la lista tras el reemplazo.

Por último, para **eliminar un nodo** se utiliza **removeChild** indicándole el nodo a eliminar:

```
nodoPadre.removeChild(nodoReferencia);
```

Finalizadas las operaciones que pueden realizarse para manipular los elementos del DOM, existen muchas otras propiedades y métodos que permiten realizar tareas interesantes con los nodos de tipo **Element**. Desafortunadamente la lista es demasiado larga y cambiante como para cubrirlos aquí.

Para saber más

Con este enlace o con el código QR se accede a la lista completa, y actualizada al día, de todas las propiedades y métodos vigentes del tipo de nodo **Element**:

<https://developer.mozilla.org/es/docs/Web/API/Element>



Manipulación de atributos

Para manipular los atributos de un elemento, lo primero es **comprobar si existe el atributo** para el elemento indicado. **hasAttribute** realiza esta comprobación y devuelve **true** si el elemento lo tiene o **false** en caso contrario.

```
document.getElementsByTagName("ul")[0].hasAttribute("id"); // true
document.getElementsByTagName("ul")[0].hasAttribute("class"); // false
```

Si lo que se pretende es conocer la **lista completa de atributos de un elemento**, se debe recurrir a **attributes**. Es una propiedad que devuelve un mapa de nodos con nombre, **NamedNodeMap**, donde cada elemento del mapa contiene el nombre y el valor del atributo; no es un **array**, pero es posible recorrerlo como si lo fuera.

```

```

Esta línea de código HTML representa la incorporación de una imagen por medio de la etiqueta **img** y tres de sus atributos típicos: **src**, **alt** y **title**.

```
let atributos = document.getElementsByTagName("img")[0].attributes;
for (let atributo of atributos) {
    console.log(`[Nombre,Valor]:[${atributo.name},${atributo.value}]`);
}
```

[Nombre,Valor]:[src,#]
[Nombre,Valor]:[alt, texto alternativo]
[Nombre,Valor]:[title,titulo]

Figura 6.19. Salida en consola de los pares clave-valor de cada atributo.

Otra forma de acceder al valor de un atributo es a través de su nombre y del método **getAttribute**:

```
document.getElementsByTagName("img")[0].getAttribute("title");
// devuelve título
```

También puede modificarse el valor de un atributo con **setAttribute**, simplemente indicando el nombre del atributo y su nuevo valor:

```
document.getElementsByTagName("img")[0].setAttribute("title", "nuevo título");
```

Otra de las operaciones comunes, la **eliminación de un atributo**, se consigue invocando a **removeAttribute** pasándole el nombre del atributo.

```
document.getElementsByTagName("img")[0].removeAttribute("title");
```

A veces, por la naturaleza de ciertos atributos, interesa eliminar un atributo, añadirlo más tarde, volver a eliminarlo, y así sucesivamente en función de las acciones que realice el usuario en la aplicación web. Esto se consigue con **toggleAttribute**. Si por ejemplo se hace **input.toggleAttribute("disabled")** se ocultará el atributo **disabled** si está presente, o lo insertará si no lo está.

El conocimiento de la manipulación de atributos visto hasta ahora es útil, pero cobra una nueva dimensión al aplicarlo a la gestión de los estilos de una aplicación web.

Es verdad que con los métodos usados se podrían modificar los atributos relacionados con el código CSS de una página. Sin embargo, JavaScript ofrece otras posibilidades cuyo manejo es todavía más sencillo. Para ello, se asocia a cada elemento la propiedad **style**, que contiene todas las propiedades CSS del elemento:

```
let ul = document.getElementById("lista");
ul.style.border = "1px solid lightblue";
```

De esta manera pueden modificarse los estilos de cualquier elemento HTML.

Es importante tener en cuenta que la propiedad **style** no permite obtener las propiedades CSS que se aplican a un elemento desde hojas de estilo externas. Para ello, se debe usar el método **getComputedStyle**, que permite ver qué estilos se están aplicando a un elemento, pero no modificarlos.

```
console.log(window.getComputedStyle(ul).border);
// escribe: 1px solid rgb(173, 216, 230)
```

En cualquier caso, la modificación de propiedades CSS debe usarse de forma discreta puesto que genera un código difícilmente mantenible. Siempre es buena idea mantener las recomendaciones de desarrollo de hojas de estilo y su aplicación a elementos HTML mediante clases e identificadores. Por ello, JavaScript también ofrece la posibilidad de **asignar clases CSS a un elemento con className**:

```
let ul = document.getElementById("lista");
ul.className = "bordeCeleste";
```

Así mismo, puede obtenerse una lista de todas las clases que se aplican a un elemento con **classList**, un objeto que se puede recorrer como un array, pero no lo es:

```
let ul = document.getElementById("lista");
ul.className = "bordeCeleste letraBlanca fondoGris";
for (let clase of ul.classList)
    console.log(clase);
```

Además, **classList** tiene la ventaja de que aporta métodos típicos para manipular la lista de forma sencilla como **add**, **remove**, **toggle**, **contains** o **replace**, cuyo uso es trivial.

Actividad propuesta 6.4

Bloques con estilo

Crea una caja HTML (div) a la que apliques el siguiente CSS:

```
div {
    width: 50%;
    height: 100px;
    border-top: 1px solid #FF0000;
    padding-left: 20px;
    margin-bottom: 10px;
}
```

Y solo usando JavaScript modifica el estilo para que los valores sean respectivamente: 30%, 250px, 5px dashed #00FF00, 15px y 25px.

Junto con los estilos, la otra aplicación de la gestión de atributos mediante JavaScript, que proporciona importantes ventajas, es la **manipulación de atributos data**. Son atributos que los programadores pueden crear en un elemento para almacenar cualquier dato que deba utilizarse con JavaScript. Por ejemplo:

```
<ul id="precios" data-divisa="euro" data-iva="21">
    <li>14.95</li>
    <li>95.99</li>
</ul>
```

En este ejemplo, se aprovechan los atributos **data** para mejorar la presentación de los datos de forma programática, ya que con JavaScript se puede recoger no solo el contenido de los elementos **li**, sino también el porcentaje de IVA que se aplica y en qué moneda están expresados los precios.

Aunque se trate de atributos personalizados por los programadores, es posible recogerlos, recorrerlos y conocer sus valores exactamente igual que se hacía con la propiedad **attributes**. No obstante, la gestión de atributos **data** tiene a **dataset** como homólogo de **style**, ya que contiene una lista de propiedades con los nombres de cada atributo **data** y su valor (internamente es un mapa):

```
let ul = document.getElementById("precios");
console.info(ul.dataset);
console.log(ul.dataset.divisa);
console.log(ul.dataset.iva);
ul.dataset.iva = 10;
console.log(ul.dataset.iva);
```



Figura 6.20. Estructura de un dataset y resultado del acceso a sus elementos.

En la Figura 6.20 puede verse la estructura interna del **dataset**, el resultado de cómo se accede a sus atributos **data** y, por último, el resultado de cómo se puede modificar su valor.

Actividad resuelta 6.4

Enlaces con estilo

Crea un menú básico de navegación con cinco enlaces. Asigna a dos de ellos el atributo **class="enlace"**. Luego recorre todo el menú y comprueba si existe el atributo **class**. Si existe, debes cambiar el valor del atributo a "**nuevoEnlace**". Si no existe debes ponerle el valor "**enlace**".

Solución

HTML

```
<nav>
    <a href="#">Enlace 1.</a>
    <a class="enlace" href="#">Enlace 2.</a>
    <a class="enlace" href="#">Enlace 3.</a>
    <a href="#">Enlace 4.</a>
    <a href="#">Enlace 5.</a>
</nav>
```

JAVASCRIPT

```
let enlaces = document.getElementsByTagName("a");
for (enlace of enlaces) {
    if (enlace.hasAttribute("class")) {
        enlace.setAttribute("class", "nuevoEnlace");
    } else {
        enlace.setAttribute("class", "enlace");
    }
}
```

6.3.3. Cookies

Las **cookies** son unos pequeños (≤ 4 kB) ficheros de texto que las aplicaciones web almacenan en el navegador del usuario con el objetivo de recuperar en futuras sesiones los datos que almacenan. ¿Por qué es esto necesario? Por la propia naturaleza del protocolo que se utiliza para acceder a internet.

Se dice que **http** es un protocolo sin estado, es decir, no conserva información entre peticiones. Con lo cual, si no se utilizan **cookies** hay cierta información de utilidad para la

usabilidad de la aplicación que habría que solicitar al usuario constantemente, como, por ejemplo, el idioma de la web, su esquema de color preferido o la fecha de su última visita.

Si bien es cierto que esta posibilidad ha abierto la puerta a numerosos usos abusivos e incluso maliciosos de las **cookies** (algo que les granjeó muy mala fama), ya existe una fuerte regulación europea que trata de evitarlos.

El objeto que proporciona JavaScript para trabajar con **cookies** es **document.cookie**, con el que pueden definirse y obtenerse **cookies** propias (un máximo de 20 por dominio).

Para crear una **cookie** y guardarla en el navegador del usuario simplemente hay que asignar al objeto **document.cookie** una cadena de caracteres con la forma **clave=valor**. Por ejemplo:

```
document.cookie = "idioma=es";
document.cookie = "esquemaColor=dark";
document.cookie = "haVotadoEncuesta=sí";
```

Para conocer todas las **cookies** hay que obtenerlas por medio de **document.cookie** en una única cadena en la que cada par **clave=valor** estará separado del siguiente por un punto y coma (;).

Otra interesante utilidad de las **cookies** es que se les puede indicar un horizonte de vida, es decir, una fecha a partir de la cual la **cookie** se destruye. Técnicamente, cada vez que el usuario cierra el navegador, las **cookies** se liberan, pero si se desea que una **cookie** persista hasta una fecha indicada, hay que indicárselo incluyendo un valor. Para expresar este valor se puede utilizar **max-age** con un número de segundos (la cuenta de segundos comienza en el momento en el que se crea la **cookie**), o **expires** con una fecha en formato GMT:

```
let limiteEnSegundos = 60*60*24*365; //dentro de un año
document.cookie = `esquemaColor=dark;max-age=${limiteEnSegundos}`;
```

Actividad propuesta 6.5

Cookies con efeméride

Crea una **cookie** que almacene cualquier dato y que expire la fecha de tu próximo cumpleaños.

También es importante destacar que las **cookies** solo son accesibles por aplicaciones del mismo dominio donde se crearon y siempre que estén en el mismo directorio. Si la página que grabó la **cookie** estaba en [paraninfo.es/libros](#), cualquier página fuera de esa ruta no podrá leerla. No obstante, se puede indicar desde qué ruta del dominio se quiere que se pueda acceder a la **cookie**, expresando la ruta como valor de la clave **path**:

```
document.cookie = "esquemaColor=dark;path=/";
```

Así, aunque la **cookie** haya sido creada desde [/libros](#), es posible leerla desde cualquier localización bajo [paraninfo.es](#).

Por último, para borrar una **cookie** es suficiente con indicar como fecha de expiración cualquier momento del pasado.