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

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

Métodos tradicionales

Existen varios métodos, los más clásicos y tradicionales para realizar búsquedas de elementos en el documento. Observa que si lo que buscas es un elemento específico, lo mejor sería utilizar **getElementById()**, en caso contrario, si utilizamos uno de los 3 siguientes métodos, nos devolverá un donde tendremos que elegir el elemento en cuestión posteriormente:

Métodos de búsqueda	Descripción
.getElementById(id)	Busca el elemento HTML con el id **id**. Si no, devuelve .
.getElementsByClassName(class)	Busca elementos con la clase **class**. Si no, devuelve **[]**.
.getElementsByName(name)	Busca elementos con atributo name **name**. Si no, devuelve **[]**.
.getElementsByTagName(tag)	Busca elementos **tag**. Si no encuentra ninguno, devuelve **[]**.

Estos son los **4 métodos tradicionales** de Javascript para manipular el DOM. Se denominan tradicionales porque son los que existen en Javascript desde versiones más antiguas. Dichos métodos te permiten buscar elementos en la página dependiendo de los atributos **id**, **class**, **name** o de la propia etiqueta, respectivamente.

getElementById()

El primer método, **.getElementById(id)** busca un elemento HTML con el **id** especificado en **id** por parámetro. En principio, un documento HTML bien construído **no debería** tener más de un elemento con el mismo **id**, por lo tanto, este método devolverá siempre un solo elemento:

```
const page = document.getElementById("page");   // <div id="page"></div>
```

getElementsByClassName()

Por otro lado, el método **.getElementsByClassName(class)** permite buscar los elementos con la **clase** especificada en **class**. Es importante darse cuenta del matiz de que el metodo tiene **getElements** en plural, y esto es porque al devolver **clases** (*al contrario que los id*) se pueden repetir, y por lo tanto, devolvernos varios elementos, no sólo uno.

```
const items = document.getElementsByClassName("item");  // [div, div, div]

console.log(items[0]);      // Primer item encontrado: <div class="item"></div>
console.log(items.length);  // 3
```

Exactamente igual funcionan los métodos **getElementsByName(name)** y **getElementsByTagName(tag)**, salvo que se encargan de buscar elementos HTML por su atributo **name** o por su propia **etiqueta** de elemento HTML, respectivamente:

```
// Obtiene los elementos con atributo name="nickname"
const nicknames = document.getElementsByName("nickname");

// Obtiene todos los elementos <div> de la página
const divs = document.getElementsByTagName("div");
```

> Recuerda que el primer método tiene **getElement** en singular y el resto **getElements** en plural. Ten en cuenta ese detalle para no olvidarte que uno devuelve un sólo elemento y el resto una lista de ellos.

Métodos modernos

Aunque podemos utilizar los métodos tradicionales que acabamos de ver, actualmente tenemos a nuestra disposición dos nuevos métodos de búsqueda de elementos que son mucho más cómodos y prácticos si conocemos y dominamos los selectores CSS. Es el caso de los métodos **.querySelector()** y **.querySelectorAll()**:

Método de búsqueda	Descripción
.querySelector(sel)	Busca el primer elemento que coincide con el selector CSS **sel**. Si no, .
.querySelectorAll(sel)	Busca todos los elementos que coinciden con el selector CSS **sel**. Si no, **[]**.

Con estos dos métodos podemos realizar todo lo que hacíamos con los **métodos tradicionales** mencionados anteriormente e incluso muchas más cosas (*en menos código*), ya que son muy flexibles y potentes gracias a los **selectores CSS**.

querySelector()

El primero, **.querySelector(selector)** devuelve el primer elemento que encuentra que encaja con el selector CSS suministrado en **selector**. Al igual que su «equivalente» **.getElementById()**, devuelve un solo elemento y en caso de no coincidir con ninguno, devuelve :

```
const page = document.querySelector("#page");              // <div id="page"></div>
const info = document.querySelector("main .info");         // <div class="info"></div>
```

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

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

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

querySelectorAll()

Por otro lado, el método **.querySelectorAll()** realiza una búsqueda de elementos como lo hace el anterior, sólo que como podremos intuir por ese **All()**, devuelve un con todos los elementos que coinciden con el CSS:

```
// Obtiene todos los elementos con clase "info"
const infos = document.querySelectorAll(".info");

// Obtiene todos los elementos con atributo name="nickname"
const nicknames = document.querySelectorAll('[name="nickname"]');

// Obtiene todos los elementos <div> de la página HTML
const divs = document.querySelectorAll("div");
```

En este caso, recuerda que querySelectorAll**()** siempre nos devolverá un grupo de elementos. Depende de los elementos que encuentre mediante nos devolverá un valor de **0** elementos o de **1**, **2** o más elementos.

> Al realizar una búsqueda de elementos y guardarlos en una variable, podemos realizar la búsqueda posteriormente sobre esa variable en lugar de hacerla sobre **document**. Esto permite realizar búsquedas acotadas por zonas, en lugar de realizarlo siempre sobre **document**, que buscará en todo el documento HTML.

Sobre todo si te encuentras en fase de aprendizaje, lo normal suele ser crear código HTML desde un fichero HTML. Sin embargo, y sobre todo con el auge de las páginas **SPA** (*Single Page Application**) y los frameworks Javascript, esto ha cambiado bastante y es bastante frecuente **crear código HTML desde Javascript** de forma dinámica.

Esto tiene sus ventajas y sus desventajas. Un fichero **.html** siempre será más sencillo, más «estático» y más directo, ya que lo primero que analiza un navegador web es un fichero de marcado HTML. Por otro lado, un fichero **.js** es más complejo y menos directo, pero mucho más potente, «dinámico» y flexible, con menos limitaciones.

En este artículo vamos a ver como podemos **crear elementos HTML desde Javascript** y aprovecharnos de la potencia de Javascript para hacer cosas que desde HTML, sin ayuda de Javascript, no podríamos realizar o costaría mucho más.

Crear elementos HTML

Existen una serie de métodos para **crear de forma eficiente** diferentes elementos HTML o nodos, y que nos pueden convertir en una tarea muy sencilla el crear estructuras dinámicas, mediante bucles o estructuras definidas:

Métodos	Descripción
.createElement(tag, options)	Crea y devuelve el elemento HTML definido por el **tag**.
.createComment(text)	Crea y devuelve un nodo de comentarios HTML **<!-- text -->**.
.createTextNode(text)	Crea y devuelve un nodo HTML con el texto **text**.

.cloneNode(deep)	Clona el nodo HTML y devuelve una copia. **deep** es **false** por defecto.
.isConnected	Indica si el nodo HTML está insertado en el documento HTML.

Para empezar, nos centraremos principalmente en la primera, que es la que utilizamos para **crear elementos HTML** en el DOM.

El método createElement()

Mediante el método **.createElement()** podemos crear un HTML **en memoria** (*¡no estará insertado aún en nuestro documento HTML!*). Con dicho elemento almacenado en una variable, podremos modificar sus características o contenido, para **posteriormente** insertarlo en una posición determinada del DOM o documento HTML.

Vamos a centrarnos en el proceso de **creación del elemento**, y en el próximo capítulo veremos el apartado de insertarlo en el DOM. El funcionamiento de **.createElement()** es muy sencillo: se trata de pasarle el nombre de la etiqueta **tag** a utilizar.

```
const div = document.createElement("div");      // Creamos un <div></div>
const span = document.createElement("span");    // Creamos un <span></span>
const img = document.createElement("img");      // Creamos un <img>
```

De la misma forma, podemos crear comentarios HTML con **createComment()** o nodos de texto sin etiqueta HTML con **createTextNode()**, pasándole a ambos un con el texto en cuestión. En ambos, se devuelve un que podremos utilizar luego para insertar en el documento HTML:

```
const comment = document.createComment("Comentario"); // <!--Comentario-->
const text = document.createTextNode("Hola");         // Nodo de texto: 'hola'
```

El método **createElement()** tiene un parámetro opcional denominado **options**. Si se indica, será un objeto con una propiedad **is** para definir un **elemento personalizado** en una modalidad menos utilizada. Se verá más adelante en el apartado de **Web Components**.

> Ten presente que en los ejemplos que hemos visto estamos creando los elementos en una constante, pero de momento **no están añadiéndose al documento HTML**, por lo que no aparecerían visualmente. Más adelante veremos como añadirlos.

El método cloneNode()

Hay que tener mucho cuidado al crear y **duplicar** elementos HTML. Un error muy común es asignar un elemento que tenemos en otra variable, pensando que estamos creando una copia (*cuando no es así*):

```
const div = document.createElement("div");
div.textContent = "Elemento 1";

const div2 = div;  // NO se está haciendo una copia
div2.textContent = "Elemento 2";

div.textContent;  // 'Elemento 2'
```

```
<div>Elemento 1</div>
```

Con esto, quizás pueda parecer que estamos duplicando un elemento para crearlo a imagen y semejanza del original. Sin embargo, lo que se hace es una **referencia** al elemento original, de modo que si se modifica el **div2**, también se modifica el elemento original. Para evitar esto, lo ideal es utilizar el método **cloneNode()**:

```
const div = document.createElement("div");
div.textContent = "Elemento 1";

const div2 = div.cloneNode();  // Ahora SÍ estamos clonando
div2.textContent = "Elemento 2";

div.textContent;  // 'Elemento 1'
```

El método **cloneNode(deep)** acepta un parámetro **deep** opcional, a **false** por defecto, para indicar el tipo de clonación que se realizará:

- Si es **true**, clonará también sus hijos, conocido como una **clonación profunda** (*Deep Clone*).
- Si es **false**, no clonará sus hijos, conocido como una **clonación superficial** (*Shallow Clone*).

La propiedad isConnected

Por último, la propiedad **isConnected** nos indica si el nodo en cuestión está conectado al DOM, es decir, si está insertado en el documento HTML:

- Si es **true**, significa que el elemento está conectado al DOM.
- Si es **false**, significa que el elemento no está conectado al DOM.

Hasta ahora, hemos creado elementos que no lo están (*permanecen sólo en memoria*). En el capítulo Insertar elementos en el DOM veremos como insertarlos en el documento HTML para que aparezca visualmente en la página.

Atributos HTML de un elemento

Hasta ahora, hemos visto cómo crear elementos HTML con Javascript, pero no hemos visto cómo modificar los atributos HTML de dichas etiquetas creadas. En general, una vez tenemos un elemento sobre el que vamos a crear algunos atributos, lo más sencillo es **asignarle valores como propiedades** de objetos:

```
const div = document.createElement("div"); // <div></div>
div.id = "page";           // <div id="page"></div>
div.className = "data";    // <div id="page" class="data"></div>
div.style = "color: red"; // <div id="page" class="data" style="color: red"></div>
```

Sin embargo, en algunos casos esto se puede complicar (*como se ve en uno de los casos del ejemplo anterior*). Por ejemplo, la palabra **class** (*para crear clases*) o la palabra **for** (*para bucles*) son palabras reservadas de Javascript y no se podrían utilizar para crear

atributos. Por ejemplo, si queremos establecer una clase, se debe utilizar la propiedad **className**.

Aunque la forma anterior es la más rápida, tenemos algunos métodos para utilizar en un elemento HTML y añadir, modificar o eliminar sus atributos:unos ejemplos de uso donde podemos ver como funcionan:

```
// Obtenemos <div id="page" class="info data dark" data-number="5"></div>
const div = document.querySelector("#page");

div.hasAttribute("data-number");  // true (data-number existe)
div.hasAttributes();              // true (tiene 3 atributos)

div.getAttributeNames();          // ["id", "data-number", "class"]
div.getAttribute("id");           // "page"

div.removeAttribute("id");        // <div class="info data dark" data-number="5"></div>
div.setAttribute("id", "page");   // (Vuelve a añadirlo)
```

> Recuerda que hasta ahora hemos visto cómo crear elementos y cambiar sus atributos, pero **no los hemos insertado en el DOM** o documento HTML, por lo que no los veremos visualmente en la página.

Reemplazar contenido

Comenzaremos por la familia de propiedades siguientes, que enmarcamos dentro de la categoría de **reemplazar contenido** de elementos HTML. Se trata de una vía rápida con la cuál podemos añadir (*o más bien, reemplazar*) el contenido de una etiqueta HTML.

Las propiedades son las siguientes:

Propiedades	Descripción
.nodeName	Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura.
.textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar.
.innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar.
.outerHTML	Idem a **.innerHTML** pero incluyendo el HTML del propio elemento HTML.

La propiedad **nodeName** nos devuelve el nombre del todo, que en elementos HTML es interesante puesto que nos devuelve el nombre de la etiqueta **en mayúsculas**. Se trata de una propiedad de sólo lectura, por lo cual no podemos modificarla, sólo acceder a ella.

La propiedad textContent

La propiedad **.textContent** nos devuelve el **contenido de texto** de un elemento HTML. Es útil para obtener (*o modificar*) **sólo el texto** dentro de un elemento, obviando el etiquetado HTML:

```
const div = document.querySelector("div");  // <div></div>

div.textContent = "Hola a todos"; // <div>Hola a todos</div>
div.textContent;  // "Hola a todos"
```

Observa que también podemos utilizarlo para **reemplazar el contenido de texto**, asignándolo como si fuera una variable o constante. En el caso de que el elemento tenga anidadas varias etiquetas HTML una dentro de otra, la propiedad **.textContent** se quedará sólo con el contenido textual completo, como se puede ver en el siguiente ejemplo:

```
// Obtenemos <div class="info">Hola <strong>amigos</strong></div>
const div = document.querySelector(".info");

div.textContent;  // "Hola amigos"
```

La propiedad innerHTML

Por otro lado, la propiedad **.innerHTML** nos permite hacer lo mismo, pero interpretando el código HTML indicado y renderizando sus elementos:

```
const div = document.querySelector(".info"); // <div class="info"></div>

div.innerHTML = "<strong>Importante</strong>"; // Interpreta el HTML
div.innerHTML;  // "<strong>Importante</strong>"
div.textContent;  // "Importante"

div.textContent = "<strong>Importante</strong>";  // No interpreta el HTML
```

Observa que la diferencia principal entre **.innerHTML** y **.textContent** es que el primero renderiza e interpreta el marcado HTML, mientras que el segundo lo inserta como contenido de texto literalmente.

> Ten en cuenta que la propiedad **.innerHTML** comprueba y parsea el marcado HTML escrito (*corrigiendo si hay errores*) antes de realizar la asignación. Por ejemplo, si en el ejemplo anterior nos olvidamos de escribir el cierre **** de la etiqueta, **.innerHTML** automáticamente lo cerrará. Esto puede provocar algunas incongruencias si el código es

> incorrecto o una disminución de rendimiento en textos muy grandes que hay que preprocesar.

Por otro lado, la propiedad **.outerHTML** es muy similar a **.innerHTML**. Mientras que esta última devuelve el código HTML del interior de un elemento HTML, **.outerHTML** devuelve también el código HTML del propio elemento en cuestión. Esto puede ser muy útil para reemplazar un elemento HTML combinándolo con **.innerHTML**:

```
const data = document.querySelector(".data");
data.innerHTML = "<h1>Tema 1</h1>";

data.textContent;   // "Tema 1"
data.innerHTML;     // "<h1>Tema 1</h1>"
data.outerHTML;     // "<div class="data"><h1>Tema 1</h1></div>"
```

En este ejemplo se pueden observar las diferencias entre las propiedades **.textContent** (*contenido de texto*), **.innerHTML** (*contenido HTML*) y **.outerHTML** (*contenido y contenedor HTML*).

Insertar elementos

A pesar de que los métodos anteriores son suficientes para crear elementos y estructuras HTML complejas, sólo son aconsejables para pequeños fragmentos de código o texto, ya que en estructuras muy complejas (*con muchos elementos HTML*) la **legibilidad del código** sería menor y además, el rendimiento podría resentirse.

Hemos aprendido a crear elementos HTML y sus atributos, pero aún no hemos visto como añadirlos al documento HTML actual (*conectarlos al DOM*), operación que se puede realizar de diferentes formas mediante los siguientes métodos disponibles:

Métodos	Descripción
.appendChild(node)	Añade como hijo el nodo **node**. Devuelve el nodo insertado.
.insertAdjacentElement(pos, elem)	Inserta el elemento **elem** en la posición **pos**. Si falla, .
.insertAdjacentHTML(pos, str)	Inserta el código HTML **str** en la posición **pos**.
.insertAdjacentText(pos, text)	Inserta el texto **text** en la posición **pos**.

De ellos, probablemente el más extendido es **.appendChild()**, no obstante, la familia de métodos **.insertAdjacent*()** también tiene buen soporte en navegadores y puede usarse de forma segura en la actualidad.

El método appendChild()

Uno de los métodos más comunes para añadir un elemento HTML creado con Javascript es **appendChild()**. Como su propio nombre indica, este método realiza un **«append»**, es decir, inserta el elemento como un hijo al final de todos los elementos hijos que existan.

Es importante tener clara esta particularidad, porque aunque es lo más común, no siempre querremos insertar el elemento en esa posición:

```
const img = document.createElement("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";

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

En este ejemplo podemos ver como creamos un elemento **<img>** que aún no está conectado al DOM. Posteriormente, añadimos los atributos **src** y **alt**, obligatorios en una etiqueta de imagen. Por último, conectamos al DOM el elemento, utilizando el método **.appendChild()** sobre **document.body** que no es más que una referencia a la etiqueta **<body>** del documento HTML.

Veamos otro ejemplo:

```
const div = document.createElement("div");
div.textContent = "Esto es un div insertado con JS.";

const app = document.createElement("div"); // <div></div>
app.id = "app"; // <div id="app"></div>
app.appendChild(div); // <div id="app"><div>Esto es un div insertado con JS</div></div>
```

En este ejemplo, estamos creando dos elementos, e insertando uno dentro de otro. Sin embargo, a diferencia del anterior, el elemento **app** no está conectado aún al DOM, sino que lo tenemos aislado en esa variable, sin insertar en el documento. Esto ocurre porque **app** lo acabamos de crear, y en el ejemplo anterior usabamos **document.body** que es una referencia a un elemento que ya existe en el documento.

Los métodos insertAdjacent*()

Los métodos de la familia **insertAdjacent** son bastante más versátiles que **.appendChild()**, ya que permiten muchas más posibilidades. Tenemos tres versiones diferentes:

- **.insertAdjacentElement()** donde insertamos un objeto

- **.insertAdjacentHTML()** donde insertamos **código HTML** directamente (*similar a innerHTML*)
- **.insertAdjacentText()** donde no insertamos elementos HTML, sino un con texto

En las tres versiones, debemos indicar por parámetro un **pos** como primer parámetro para indicar en que posición vamos a insertar el contenido. Hay 4 opciones posibles:

- **beforebegin**: El elemento se inserta **antes** de la etiqueta HTML de apertura.
- **afterbegin**: El elemento se inserta **dentro** de la etiqueta HTML, **antes de su primer hijo**.
- **beforeend**: El elemento se inserta **dentro** de la etiqueta HTML, **después de su último hijo**. Es el equivalente a usar el método **.appendChild()**.
- **afterend**: El elemento se inserta **después** de la etiqueta HTML de cierre.

Veamos algunos ejemplo aplicando cada uno de ellos con el método **.insertAdjacentElement()**:

```
const div = document.createElement("div");  // <div></div>
div.textContent = "Ejemplo";                // <div>Ejemplo</div>

const app = document.querySelector("#app"); // <div id="app">App</div>

app.insertAdjacentElement("beforebegin", div);
// Opción 1: <div>Ejemplo</div> <div id="app">App</div>

app.insertAdjacentElement("afterbegin", div);
// Opción 2: <div id="app"> <div>Ejemplo</div> App</div>

app.insertAdjacentElement("beforeend", div);
// Opción 3: <div id="app">App <div>Ejemplo</div> </div>

app.insertAdjacentElement("afterend", div);
// Opción 4: <div id="app">App</div> <div>Ejemplo</div>
```

Ten en cuenta que en el ejemplo muestro **varias opciones alternativas**, no lo que ocurriría tras ejecutar las cuatro opciones una detrás de otra.

Por otro lado, notar que tenemos **tres versiones** en esta familia de métodos, una que actua sobre elementos HTML (*la que hemos visto*), pero otras dos que actuan sobre código HTML y sobre nodos de texto. Veamos un ejemplo de cada una:

```
app.insertAdjacentElement("beforebegin", div);
// Opción 1: <div>Ejemplo</div> <div id="app">App</div>

app.insertAdjacentHTML("beforebegin", '<p>Hola</p>');
// Opción 2: <p>Hola</p> <div id="app">App</div>

app.insertAdjacentText("beforebegin", "Hola a todos");
// Opción 3: Hola a todos <div id="app">App</div>
```

Eliminar elementos

Al igual que podemos insertar o reemplazar elementos, también podemos eliminarlos. Ten en cuenta que al «eliminar» un nodo o elemento HTML, lo que hacemos realmente no es borrarlo, sino **desconectarlo del DOM o documento HTML**, de modo que no están conectados, pero siguen existiendo.

El método remove()

Probablemente, la forma más sencilla de eliminar nodos o elementos HTML es utilizando el método **.remove()** sobre el nodo o etiqueta a eliminar:

```
const div = document.querySelector(".deleteme");

div.isConnected;    // true
div.remove();
div.isConnected;    // false
```

En este caso, lo que hemos hecho es buscar el elemento HTML ** ** en el documento HTML y desconectarlo de su elemento padre, de forma que dicho elemento pasa a no pertenecer al documento HTML.

Sin embargo, existen algunos métodos más para eliminar o reemplazar elementos:

Métodos	Descripción
.remove()	Elimina el propio nodo de su elemento padre.
.removeChild(node)	Elimina y devuelve el nodo hijo **node**.
.replaceChild(new, old)	Reemplaza el nodo hijo **old** por **new**. Devuelve **old**.

El método **.remove()** se encarga de desconectarse del DOM a sí mismo, mientras que el segundo método, **.removeChild()**, desconecta el nodo o elemento HTML proporcionado. Por último, con el método **.replaceChild()** se nos permite cambiar un nodo por otro.

El método removeChild()

En algunos casos, nos puede interesar eliminar un nodo hijo de un elemento. Para esas situaciones, podemos utilizar el método **.removeChild(node)** donde **node** es el nodo hijo que queremos eliminar:

```
const div = document.querySelector(".item:nth-child(2)");   // <div class="item">2</div>

document.body.removeChild(div); // Desconecta el segundo .item
```

El método replaceChild()

De la misma forma, el método **replaceChild(new, old)** nos permite cambiar un nodo hijo **old** por un nuevo nodo hijo **new**. En ambos casos, el método nos devuelve el nodo reemplazado:

```
const div = document.querySelector(".item:nth-child(2)");

const newnode = document.createElement("div");
newnode.textContent = "DOS";

document.body.replaceChild(newnode, div);
```

La propiedad className

Javascript tiene a nuestra disposición una propiedad **.className** en todos los elementos HTML. Dicha propiedad contiene el valor del atributo HTML **class**, y puede tanto leerse como reemplazarse:

Propiedad	Descripción
.className	Acceso directo al valor del atributo HTML **class**. También se puede asignar.
.classList	Objeto especial para manejar clases CSS. Contiene métodos y propiedades de ayuda.

La propiedad **.className** viene a ser la modalidad directa y rápida de utilizar el getter **.getAttribute("class")** y el setter **.setAttribute("class", v)**. Veamos un ejemplo utilizando estas propiedades y métodos y su equivalencia:

```
const div = document.querySelector(".element");

// Obtener clases CSS
div.className;             // "element shine dark-theme"
div.getAttribute("class");  // "element shine dark-theme"
```

```
// Modificar clases CSS
div.className = "elemento brillo tema-oscuro";
div.setAttribute("class", "elemento brillo tema-oscuro");
```

Trabajar con **.className** tiene una limitación cuando trabajamos con **múltiples clases CSS**, y es que puedes querer realizar una manipulación sólo en una clase CSS concreta, dejando las demás intactas. En ese caso, modificar clases CSS mediante una asignación **.className** se vuelve poco práctico. Probablemente, la forma más interesante de manipular clases desde Javascript es mediante el objeto **.classList**.

El objeto classList

Para trabajar más cómodamente, existe un sistema muy interesante para trabajar con clases: el objeto **classList**. Se trata de un objeto especial (*lista de clases*) que contiene una serie de ayudantes que permiten trabajar con las clases de forma más intuitiva y lógica.

Si accedemos a **.classList**, nos devolverá un (*lista*) de clases CSS de dicho elemento. Pero además, incorpora una serie de métodos ayudantes que nos harán muy sencillo trabajar con clases CSS:

Método	Descripción
.classList	Devuelve la lista de clases del elemento HTML.
.classList.item(n)	Devuelve la clase número **n** del elemento HTML.
.classList.add(c1, c2, ...)	Añade las clases **c1**, **c2**... al elemento HTML.

.classList.remove(c1, c2, ...)	Elimina las clases **c1**, **c2**... del elemento HTML.
.classList.contains(clase)	Indica si la **clase** existe en el elemento HTML.
.classList.toggle(clase)	Si la **clase** no existe, la añade. Si no, la elimina.
.classList.toggle(clase, expr)	Si **expr** es **true**, añade **clase**. Si no, la elimina.
.classList.replace(old, new)	Reemplaza la clase **old** por la clase **new**.

Veamos un ejemplo de uso de cada método de ayuda. Supongamos que tenemos el siguiente elemento HTML en nuestro documento. Vamos a acceder a él y a utilizar el objeto **.classList** con dicho elemento:

```
<div id="page" class="info data dark" data-number="5"></div>
```

Observa que dicho elemento HTML tiene:

- Un atributo **id**
- Tres clases CSS: **info**, **data** y **dark**
- Un metadato HTML **data-number**

Añadir y eliminar clases CSS

Los métodos **classList.add()** y **classList.remove()** permiten indicar una o múltiples clases CSS a añadir o eliminar. Observa el siguiente código donde se ilustra un ejemplo:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.add("uno", "dos"); // No devuelve nada.
div.classList; // ["info", "data", "dark", "uno", "dos"]

div.classList.remove("uno", "dos"); // No devuelve nada.
div.classList; // ["info", "data", "dark"]
```

En el caso de que se añada una clase CSS que ya existía previamente, o que se elimine una clase CSS que no existía, simplemente no ocurrirá nada.

Conmutar o alternar clases CSS

Un ayudante muy interesante es el del método **classList.toggle()**, que lo que hace es **añadir o eliminar la clase CSS** dependiendo de si ya existía previamente. Es decir, añade la clase si no existía previamente o elimina la clase si existía previamente:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.toggle("info"); // Como "info" existe, lo elimina. Devuelve "false"
div.classList; // ["data", "dark"]

div.classList.toggle("info"); // Como "info" no existe, lo añade. Devuelve "true"
div.classList; // ["info", "data", "dark"]
```

Observa que **.toggle()** devuelve un que será **true** o **false** dependiendo de si, tras la operación, la clase sigue existiendo o no. Ten en cuenta que en **.toggle()**, al contrario que **.add()** o **.remove()**, sólo se puede indicar una clase CSS por parámetro.

Otros métodos de clases CSS

Por otro lado, tenemos otros métodos menos utilizados, pero también muy interesantes:

- El método **.classList.item(n)** nos devuelve la clase CSS ubicada en la posición **n**.
- El método **.classList.contains(name)** nos devuelve si la clase CSS **name** existe o no.
- El método **.classList.replace(old, current)** cambia la clase **old** por la clase **current**.

Veamos un ejemplo:

```
const div = document.querySelector("#page");

div.classList; // ["info", "data", "dark"]

div.classList.item(1); // 'data'
div.classList.contains("info"); // Devuelve `true` (existe la clase)
div.classList.replace("dark", "light"); // Devuelve `true` (se hizo el cambio)
```

Con todos estos métodos de ayuda, nos resultará mucho más sencillo manipular clases CSS desde Javascript en nuestro código.

Navegar a través de elementos

Las propiedades que veremos a continuación devuelven información de otros elementos relacionados con el elemento en cuestión.

Propiedades de elementos HTML	Descripción
children	Devuelve una lista de elementos HTML hijos.

parentElement	Devuelve el padre del elemento o si no tiene.
firstElementChild	Devuelve el primer elemento hijo.
lastElementChild	Devuelve el último elemento hijo.
previousElementSibling	Devuelve el elemento hermano anterior o si no tiene.
nextElementSibling	Devuelve el elemento hermano siguiente o si no tiene.

En primer lugar, tenemos la propiedad **children** que nos ofrece un con una lista de elementos HTML hijos. Podríamos acceder a cualquier hijo utilizando los corchetes de array y seguir utilizando otras propiedades en el hijo seleccionado.

- La propiedad **firstElementChild** sería un acceso rápido a **children[0]**
- La propiedad **lastElementChild** sería un acceso rápido al último elemento hijo.

Por último, tenemos las propiedades **previousElementSibling** y **nextElementSibling** que nos devuelven los elementos hermanos anteriores o posteriores, respectivamente. La propiedad **parentElement** nos devolvería el padre del elemento en cuestión. En el caso de no existir alguno de estos elementos, nos devolvería .

Consideremos el siguiente documento HTML:

```
<html>
  <body>
    <div id="app">
      <div class="header">
        <h1>Titular</h1>
      </div>
      <p>Párrafo de descripción</p>
      <a href="/">Enlace</a>
    </div>
  </body>
</html>
```

Si trabajamos bajo este documento HTML, y utilizamos el siguiente código Javascript, podremos «navegar» por la jerarquía de elementos, **moviéndonos entre elementos** padre, hijo o hermanos:

```
document.body.children.length;  // 1
document.body.children;         // <div id="app">
document.body.parentElement;    // <html>

const app = document.querySelector("#app");

app.children;                   // [div.header, p, a]
app.firstElementChild;          // <div class="header">
app.lastElementChild;           // <a href="/">

const a = app.querySelector("a");

a.previousElementSibling;       // <p>
a.nextElementSibling;           // null
```

Eventos

En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados. Sin embargo, en la actualidad el modelo predominante es el de la programación basada en eventos. Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento. Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.

JavaScript permite realizar scripts con ambos métodos de programación: secuencial y basada en eventos. Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, se produce un evento. Cada vez que se pulsa una tecla, también se produce un evento. No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que, por ejemplo, cada vez que se carga una página, también se produce un evento.

TIPOS DE EVENTOS

Cada elemento HTML tiene definida su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos HTML y un mismo elemento HTML puede tener asociados diferentes eventos.

El nombre de los eventos se construye mediante el prefijo `on`, seguido del nombre en inglés de la acción asociada al evento. Así, el evento de pinchar un elemento con el ratón se denomina `onclick` y el evento asociado a la acción de mover el ratón se denomina `onmousemove`.

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
`onblur`	Un elemento pierde el foco	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`onchange`	Un elemento ha sido modificado	`<input>`, `<select>`, `<textarea>`
`onclick`	Pulsar y soltar el ratón	Todos los elementos

ondblclick	Pulsar dos veces seguidas con el ratón	Todos los elementos
onfocus	Un elemento obtiene el foco	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla y no soltarla	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	Página cargada completamente	<body>
onmousedown	Pulsar un botón del ratón y no soltarlo	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón "sale" del elemento	Todos los elementos
onmouseover	El ratón "entra" en el elemento	Todos los elementos
onmouseup	Soltar el botón del ratón	Todos los elementos

onreset	Inicializar el formulario	<form>
onresize	Modificar el tamaño de la ventana	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página, por ejemplo al cerrar el navegador	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son onload para esperar a que se cargue la página por completo, los eventos onclick, onmouseover, onmouseout para controlar el ratón y onsubmit para controlar el envío de los formularios.

Algunos eventos de la tabla anterior (onclick, onkeydown, onkeypress, onreset, onsubmit) permiten evitar la "acción por defecto" de ese evento. Más adelante se muestra en detalle este comportamiento, que puede resultar muy útil en algunas técnicas de programación.

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un botón de tipo <input type="submit"> se desencadenan los eventos onmousedown, onclick, onmouseup y onsubmit de forma consecutiva.

MANEJADORES DE EVENTOS

Un evento de JavaScript por sí mismo carece de utilidad. Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede responder ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *manejador de eventos* (*event handlers* en inglés).

HANDLERS Y LISTENERS

En las secciones anteriores se introdujo el concepto de "event handler" o manejador de eventos, que son las funciones que responden a los eventos que se producen. Además, se vieron tres formas de definir los manejadores de eventos para el modelo básico de eventos:

1. Código JavaScript dentro de un atributo del propio elemento HTML
2. Definición del evento en el propio elemento HTML pero el manejador es una función externa
3. Manejadores semánticos asignados mediante DOM sin necesidad de modificar el código HTML de la página

Cualquiera de estos tres modelos funciona correctamente en todos los navegadores disponibles en la actualidad. Las diferencias entre navegadores surgen cuando se define más de un manejador de eventos para un mismo evento de un elemento. La forma de asignar y *"desasignar"* manejadores múltiples depende completamente del navegador utilizado.

MANEJADORES DE EVENTOS DE DOM

La especificación DOM define otros dos métodos similares a los disponibles para Internet Explorer y denominados addEventListener() y removeEventListener() para asociar y desasociar manejadores de eventos.

La principal diferencia entre estos métodos y los anteriores es que en este caso se requieren tres parámetros: el nombre del *"event listener"*, una referencia a la función encargada de procesar el evento y el tipo de flujo de eventos al que se aplica.

El primer argumento no es el nombre completo del evento como sucede en el modelo de Internet Explorer, sino que se debe eliminar el prefijo on. En otras palabras, si en Internet Explorer se utilizaba el nombre onclick, ahora se debe utilizar click.

A continuación, se muestran los ejemplos anteriores empleando los métodos definidos por DOM:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}
var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje);

// Más adelante se decide desasociar la función al evento
elDiv.removeEventListener("click", muestraMensaje);
```

Asociando múltiples funciones a un único evento:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}

function muestraOtroMensaje() {
  console.log("Has pulsado el ratón y por eso se muestran estos mensajes");
}

var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje);
elDiv.addEventListener("click", muestraOtroMensaje);
```

Si se asocia una función a un flujo de eventos determinado, esa función sólo se puede desasociar en el mismo tipo de flujo de eventos. Si se considera el siguiente ejemplo:

```
function muestraMensaje() {
  console.log("Has pulsado el ratón");
}

var elDiv = document.getElementById("div_principal");
elDiv.addEventListener("click", muestraMensaje);

// Más adelante se decide desasociar la función al evento
elDiv.removeEventListener("click", muestraMensaje);
```

EL OBJETO EVENT

Cuando se produce un evento, no es suficiente con asignarle una función responsable de procesar ese evento. Normalmente, la función que procesa el evento necesita información relativa al evento producido: la tecla que se ha pulsado, la posición del ratón, el elemento que ha producido el evento, etc.

El objeto `event` es el mecanismo definido por los navegadores para proporcionar toda esa información. Se trata de un objeto que se crea automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.

El estándar DOM especifica que el objeto `event` es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos.

```
elDiv.onclick = function(event) {
  ...
}
```

El funcionamiento de los navegadores que siguen los estándares puede parecer "mágico", ya que en la declaración de la función se indica que tiene un parámetro, pero en la aplicación no se pasa ningún parámetro a esa función. En realidad, los navegadores que siguen los estándares crean automáticamente ese parámetro y lo pasan siempre a la función encargada de manejar el evento.

PROPIEDADES Y MÉTODOS

A pesar de que el mecanismo definido por los navegadores para el objeto event es similar, existen numerosas diferencias en cuanto a las propiedades y métodos del objeto.

PROPIEDADES DEFINIDAS POR DOM

La siguiente tabla recoge las propiedades definidas para el objeto event en los navegadores que siguen los estándares:

Propiedad/Método	Devuelve	Descripción
altKey	Boolean	Devuelve true si se ha pulsado la tecla ALT y false en otro caso
button	Número entero	El botón del ratón que ha sido pulsado. Posibles valores: 0 – Ningún botón pulsado 1 – Se ha pulsado el botón izquierdo2– Se ha pulsado el botón derecho3– Se pulsan a la vez el botón izquierdo y el derecho4– Se ha pulsado el botón central5– Se pulsan a la vez el botón izquierdo y el central6– Se pulsan a la vez el botón derecho y el central7` – Se pulsan a la vez los 3 botones
cancelable	Boolean	Indica si el evento se puede cancelar
charCode	Número entero	El código unicode del carácter correspondiente a la tecla pulsada
clientX	Número entero	Coordenada X de la posición del ratón respecto del área visible de la ventana
clientY	Número entero	Coordenada Y de la posición del ratón respecto del área visible de la ventana
ctrlKey	Boolean	Devuelve true si se ha pulsado la tecla CTRL y false en otro caso

currentTarget	Element	El elemento que es el objetivo del evento
detail	Número entero	El número de veces que se han pulsado los botones del ratón
isChar	Boolean	Indica si la tecla pulsada corresponde a un carácter
keyCode	Número entero	Indica el código numérico de la tecla pulsada
metaKey	Número entero	Devuelve true si se ha pulsado la tecla META y false en otro caso
pageX	Número entero	Coordenada X de la posición del ratón respecto de la página
pageY	Número entero	Coordenada Y de la posición del ratón respecto de la página
preventDefault()	Función	Se emplea para cancelar la acción predefinida del evento
relatedTarget	Element	El elemento que es el objetivo secundario del evento (relacionado con los eventos de ratón)
screenX	Número entero	Coordenada X de la posición del ratón respecto de la pantalla completa
screenY	Número entero	Coordenada Y de la posición del ratón respecto de la pantalla completa
shiftKey	Boolean	Devuelve true si se ha pulsado la tecla SHIFT y false en otro caso

target	Element	El elemento que origina el evento
timeStamp	Número	La fecha y hora en la que se ha producido el evento
type	Cadena de texto	El nombre del evento

Al contrario de lo que sucede con Internet Explorer, la mayoría de propiedades del objeto event de DOM son de sólo lectura. En concreto, solamente las siguientes propiedades son de lectura y escritura: altKey, button y keyCode.

La tecla META es una tecla especial que se encuentra en algunos teclados de ordenadores muy antiguos. Actualmente, en los ordenadores tipo PC se asimila a la tecla Alt o a la tecla de Windows, mientras que en los ordenadores tipo Mac se asimila a la tecla Command.

Prevenir el comportamiento por defecto

Una de las propiedades más interesantes es la posibilidad de impedir que se complete el comportamiento normal de un evento. En otras palabras, con JavaScript es posible no mostrar ningún carácter cuando se pulsa una tecla, no enviar un formulario después de pulsar el botón de envío, no cargar ninguna página al pulsar un enlace, etc. El método avanzado de impedir que un evento ejecute su acción asociada depende de cada navegador:

```
// Navegadores que siguen los estándares
objetoEvento.preventDefault();
```

TIPOS DE EVENTOS

La lista completa de eventos que se pueden generar en un navegador se puede dividir en cuatro grandes grupos. La especificación de DOM define los siguientes grupos:

- Eventos de ratón: se originan cuando el usuario emplea el ratón para realizar algunas acciones.
- Eventos de teclado: se originan cuando el usuario pulsa sobre cualquier tecla de su teclado.
- Eventos HTML: se originan cuando se producen cambios en la ventana del navegador o cuando se producen ciertas interacciones entre el cliente y el servidor.
- Eventos DOM: se originan cuando se produce un cambio en la estructura DOM de la página. También se denominan "eventos de mutación".

EVENTOS DE RATÓN

Los eventos de ratón son, con mucha diferencia, los más empleados en las aplicaciones web. Los eventos que se incluyen en esta clasificación son los siguientes:

Evento	Descripción
click	Se produce cuando se pulsa el botón izquierdo del ratón. También se produce cuando el foco de la aplicación está situado en un botón y se pulsa la tecla ENTER
dblclick	Se produce cuando se pulsa dos veces el botón izquierdo del ratón
mousedown	Se produce cuando se pulsa cualquier botón del ratón
mouseout	Se produce cuando el puntero del ratón se encuentra en el interior de un elemento y el usuario mueve el puntero a un lugar fuera de ese elemento
mouseover	Se produce cuando el puntero del ratón se encuentra fuera de un elemento y el usuario mueve el puntero hacia un lugar en el interior del elemento
mouseup	Se produce cuando se suelta cualquier botón del ratón que haya sido pulsado
mousemove	Se produce (de forma continua) cuando el puntero del ratón se encuentra sobre un elemento

Todos los elementos de las páginas soportan los eventos de la tabla anterior.

PROPIEDADES

El objeto event contiene las siguientes propiedades para los eventos de ratón:

- Las coordenadas del ratón (todas las coordenadas diferentes relativas a los distintos elementos)
- La propiedad type
- La propiedad srcElement (Internet Explorer) o target (DOM)

- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)
- La propiedad button (sólo en los eventos mousedown, mousemove, mouseout, mouseover y mouseup)

Los eventos mouseover y mouseout tienen propiedades adicionales. Internet Explorer define la propiedad fromElement, que hace referencia al elemento desde el que el puntero del ratón se ha movido y toElement que es el elemento al que el puntero del ratón se ha movido. De esta forma, en el evento mouseover, la propiedad toElement es idéntica a srcElement y en el evento mouseout, la propiedad fromElement es idéntica a srcElement.

En los navegadores que soportan el estándar DOM, solamente existe una propiedad denominada relatedTarget. En el evento mouseout, relatedTarget apunta al elemento al que se ha movido el ratón. En el evento mouseover, relatedTarget apunta al elemento desde el que se ha movido el puntero del ratón.

Cuando se pulsa un botón del ratón, la secuencia de eventos que se produce es la siguiente: mousedown, mouseup, click. Por tanto, la secuencia de eventos necesaria para llegar al doble click llega a ser tan compleja como la siguiente: mousedown, mouseup, click, mousedown, mouseup, click, dblclick.

EVENTOS DE TECLADO

Los eventos que se incluyen en esta clasificación son los siguientes:

Evento	Descripción
keydown	Se produce cuando se pulsa cualquier tecla del teclado. También se produce de forma continua si se mantiene pulsada la tecla
keypress	Se produce cuando se pulsa una tecla correspondiente a un carácter alfanumérico (no se tienen en cuenta teclas como SHIFT, ALT, etc.). También se produce de forma continua si se mantiene pulsada la tecla
keyup	Se produce cuando se suelta cualquier tecla pulsada

PROPIEDADES

El objeto event contiene las siguientes propiedades para los eventos de teclado:

- La propiedad keyCode
- La propiedad charCode (sólo DOM)

- La propiedad srcElement (Internet Explorer) o target (DOM)
- Las propiedades shiftKey, ctrlKey, altKey y metaKey (sólo DOM)

Cuando se pulsa una tecla correspondiente a un carácter alfanumérico, se produce la siguiente secuencia de eventos: keydown, keypress, keyup. Cuando se pulsa otro tipo de tecla, se produce la siguiente secuencia de eventos: keydown, keyup. Si se mantiene pulsada la tecla, en el primer caso se repiten de forma continua los eventos keydown y keypress y en el segundo caso, se repite el evento keydown de forma continua.

EVENTOS HTML

Los eventos HTML definidos se recogen en la siguiente tabla:

Evento	Descripción
load	Se produce en el objeto window cuando la página se carga por completo. En el elemento <img> cuando se carga por completo la imagen. En el elemento` cuando se carga el objeto
unload	Se produce en el objeto window cuando la página desaparece por completo (al cerrar la ventana del navegador por ejemplo). En el elemento <object> cuando desaparece el objeto.
abort	Se produce en un elemento <object> cuando el usuario detiene la descarga del elemento antes de que haya terminado
error	Se produce en el objeto window cuando se produce un error de JavaScript. En el elemento <img> cuando la imagen no se ha podido cargar por completo y en el elemento <object> cuando el elemento no se carga correctamente
select	Se produce cuando se seleccionan varios caracteres de un cuadro de texto (<input> y <textarea>)
change	Se produce cuando un cuadro de texto (<input> y <textarea>) pierde el foco y su contenido ha variado. También se produce cuando varía el valor de un elemento <select>

submit	Se produce cuando se pulsa sobre un botón de tipo submit (<input type="submit">)
reset	Se produce cuando se pulsa sobre un botón de tipo reset (<input type="reset">)
resize	Se produce en el objeto window cuando se redimensiona la ventana del navegador
scroll	Se produce en cualquier elemento que tenga una barra de scroll, cuando el usuario la utiliza. El elemento <body> contiene la barra de scroll de la página completa
focus	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento obtiene el foco
blur	Se produce en cualquier elemento (incluido el objeto window) cuando el elemento pierde el foco

Uno de los eventos más utilizados es el evento load, ya que todas las manipulaciones que se realizan mediante DOM requieren que la página esté cargada por completo y por tanto, el árbol DOM se haya construido completamente.

El elemento <body> define las propiedades scrollLeft y scrollTop que se pueden emplear junto con el evento scroll.