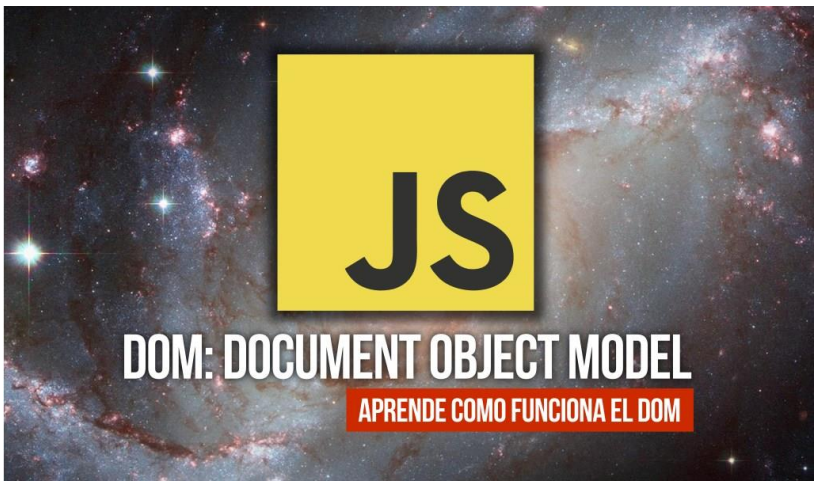
¿Qué es el DOM?

Cuando comenzamos en el mundo del **desarrollo web**, normalmente comenzamos por aprender a escribir etiquetado o **marcado HTML** y además, añadir **estilos CSS** para darle color, forma y algo de interacción. Sin embargo, a medida que avanzamos, nos damos cuenta que en cierta forma podemos estar bastante limitados.

Si únicamente utilizamos HTML/CSS, sólo podremos crear **páginas «estáticas»** (*sin demasiada personalización por parte del usuario*), pero si añadimos Javascript, podremos crear **páginas «dinámicas»**. Cuando hablamos de páginas dinámicas, nos referimos a que podemos dotar de la potencia y flexibilidad que nos da un lenguaje de programación para crear documentos y páginas mucho más ricas, que brinden una experiencia más completa y con el que se puedan automatizar un gran abanico de tareas y acciones.

¿Qué es el DOM?

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*).



En Javascript, cuando nos referimos al **DOM** nos referimos a esta estructura, que podemos modificar de forma dinámica desde Javascript, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc...

Al estar "amparado" por un **lenguaje de programación**, todas estas tareas se pueden automatizar, incluso indicando que se realicen cuando el usuario haga acciones determinadas, como por ejemplo: pulsar un botón, mover el ratón, hacer click en una parte del documento, escribir un texto, etc...

El objeto document

En Javascript, la forma de acceder al DOM es a través de un objeto llamado `document`, que representa el árbol DOM de la página o pestaña del navegador donde nos encontramos. En su interior pueden existir varios tipos de elementos, pero principalmente serán o :

- no es más que la representación genérica de una etiqueta: `HTMLElement`.
- es una unidad más básica, la cuál puede ser o un **nodo de texto**.

Todos los **elementos HTML**, dependiendo del elemento que sean, tendrán un tipo de dato específico. Algunos ejemplos:

Tipo de dato	Tipo específico	Etiqueta	Descripción
`HTMLElement`	`HTMLDivElement`	`<div>`	Capa divisoria invisible (en bloque).
`HTMLElement`	`HTMLSpanElement`	`<span>`	Capa divisoria invisible (en línea).
`HTMLElement`	`HTMLImageElement`	`<img>`	Imagen.
`HTMLElement`	`HTMLAudioElement`	`<audio>`	Contenedor de audio.

Obviamente, existen muchos tipos de datos específicos, uno por cada etiqueta HTML.

API nativa de Javascript

En los siguientes capítulos veremos que **Javascript** nos proporciona un conjunto de herramientas para trabajar de forma nativa con el DOM de la página, entre las que se encuentran:

Capítulo del DOM	Descripción
🔍 **Buscar etiquetas**	Familia de métodos entre los que se encuentran funciones como `.getElementById()`, `.querySelector()` o `.querySelectorAll()`, entre otras.
Crear etiquetas	Una serie de métodos y consejos para crear elementos en la página y trabajar con ellos de forma dinámica.
🔌 **Insertar etiquetas**	Las mejores formas de añadir elementos al DOM, ya sea utilizando propiedades como `.innerHTML` o método como `.appendChild()`, `.insertAdjacentHTML()`, entre otros.
🎨 **Gestión de clases CSS**	Consejos para la utilización de la API `.classList` de Javascript que nos permite manipular clases CSS desde JS, de modo que podamos añadir, modificar, eliminar clases de CSS de un elemento de una forma práctica y cómoda.
🚗 **Navegar entre elementos**	Utilización de una serie de métodos y propiedades que nos permiten «navegar» a través de la jerarquía del DOM, ciñéndonos a la estructura del documento y la posición de los elementos en la misma.

Librerías de terceros

En muchos casos, el rendimiento no es lo suficientemente importante como para justificar trabajar a bajo nivel, por lo que se prefiere utilizar algunas **librerías de terceros** que nos facilitan el trabajo a costa de reducir minimamente el rendimiento, pero permitiéndonos programar más rápidamente.

Si es tu caso, puedes utilizar alguna de las siguientes librerías para abstraerte del DOM:

Librería	Descripción	GitHub
RE:DOM	Librería para crear interfaces de usuario, basada en DOM.	@redom/redom
Voyeur.js	Pequeña librería para manipular el DOM	@adriancooney/voyeur.js
HtmlJs	Motor de renderización de HTML y data binding (MVVM)	@nhanfu/htmljs
DOMtastic	Libraría moderna y modular para DOM/Events	@webpro/DOMtastic
UmbrellaJS	Librería para manipular el DOM y eventos	@franciscop/umbrella
SuperDOM	Manipulando DOM como si estuvieras en 2018	@szaranger/superdom

Muchas veces, también se eligen frameworks de Javascript para trabajar, que en cierta forma también te abstraen de tener que gestionar el DOM a bajo nivel, y lo cambian por realizar otras tareas o estrategias relacionadas con el framework escogido.

Seleccionar elementos del 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>
```

Recuerda que en el caso de no encontrar el elemento indicado, devolverá .

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
```

Estos métodos devuelven siempre un con todos los elementos encontrados que encajen con el criterio. En el caso de no encontrar ninguno, devolverán un vacío: `[]`.

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

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

OJO: Aunque en esta documentación se hace referencia a , realmente los métodos de búsqueda generalmente devuelven un tipo de dato `HTMLCollection` o `NodeList`, que aunque actúan de forma muy similar a un , no son arrays, y por lo tanto pueden carecer de algunos métodos, como por ejemplo `.forEach()`.

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 de elementos. Depende de los elementos que encuentre mediante el , nos devolverá un 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.

Crear elementos en el DOM

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'
```

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 como crear elementos HTML con Javascript, pero no hemos visto como 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`.

Es posible asignar a la propiedad `className` varias clases en un separadas por espacio. De esta forma se asignarán múltiples clases. Aún así, recomendamos utilizar la propiedad `classList` que explicaremos más adelante en el capítulo manipulación de clases CSS.

Aunque la forma anterior es la más rápida, tenemos algunos métodos para utilizar en un elemento HTML y añadir, modificar o eliminar sus atributos:

Métodos	Descripción

Métodos	Descripción
`hasAttributes()`	Indica si el elemento tiene atributos HTML.
`hasAttribute(attr)`	Indica si el elemento tiene el atributo HTML `attr`.
`getAttributeNames()`	Devuelve un con los atributos del elemento.
`getAttribute(attr)`	Devuelve el valor del atributo `attr` del elemento o si no existe.
`removeAttribute(attr)`	Elimina el atributo `attr` del elemento.
`setAttribute(attr, value)`	Añade o cambia el atributo `attr` al valor `value`.
`getAttributeNode(attr)`	Idem a `getAttribute()` pero devuelve el atributo como **nodo**.
`removeAttributeNode(attr)`	Idem a `removeAttribute()` pero devuelve el atributo como **nodo**.
`setAttributeNode(attr, value)`	Idem a `setAttribute()` pero devuelve el atributo como **nodo**.

Estos métodos son bastante autoexplicativos y fáciles de entender, aún así, vamos a ver 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)
```

Los tres últimos métodos mencionados: `getAttributeNode()`, `removeAttributeNode()` y `setAttributeNode()` son versiones idénticas a sus homónimos, sólo que devuelven el afectado, útil si queremos guardarlo en una variable y seguir trabajando con él.

Recuerda que hasta ahora hemos visto como 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. En el siguiente capítulo abordaremos ese tema.

Insertar elementos en el DOM

En el capítulo anterior hemos visto como crear elementos en el DOM, pero dichos elementos **se creaban en memoria** y los almacenábamos en una variable o constante. No se conectaban al DOM o documento HTML de forma automática, sino que debemos hacerlo manualmente, que es justo lo que veremos en este artículo: como insertar elementos en el DOM, así como eliminarlos.

En este artículo vamos a centrarnos en tres categorías:

- **Reemplazar** contenido de elementos en el DOM
- **Insertar** elementos en el DOM
- **Eliminar** elementos del DOM

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.
`.innerText`	Versión no estándar de `.textContent` de Internet Explorer con diferencias. **Evitar**.
`.outerText`	Versión no estándar de `.textContent`/`.outerHTML` de Internet Explorer. **Evitar**.

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 cuál 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 `</strong>` 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*).

Las propiedades `.innerText` y `.outerText` son propiedades **no estándar** de Internet Explorer. Se recomienda sólo utilizarlas con motivos de fallbacks o para dar soporte a versiones antiguas de Internet Explorer. En su lugar debería utilizarse `.textContent`.

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`.
`.insertBefore(new, node)`	Inserta el nodo `new` antes de `node` y como hijo del nodo actual.

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>
```

El método insertBefore()

Por último, el método `insertBefore(newnode, node)` es un método más específico y menos utilizado en el que se puede especificar exactamente el lugar a insertar un nodo. El parámetro `newnode` es el nodo a insertar, mientras que `node` puede ser:

- ; insertando `newnode` después del último nodo hijo. Equivalente a `.appendChild()`.
- o ; insertando `newnode` antes de dicho `node` de referencia.

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 `<div class="deleteme">` 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);
```

Manipular clases CSS (classList)

En CSS es muy común utilizar múltiples **clases CSS** para asignar estilos relacionados dependiendo de lo que queramos. Para ello, basta hacer cosas como la que veremos a continuación:

```
<div class="element shine dark-theme"></div>
```

- La clase `element` sería la clase general que representa el elemento, y que tiene estilos fijos.
- La clase `shine` podría tener una animación CSS para aplicar un efecto de brillo.
- La clase `dark-theme` podría tener los estilos de un elemento en un tema oscuro.

Todo esto se utiliza sin problema de forma estática, pero cuando comenzamos a programar en Javascript, buscamos una forma **dinámica**, práctica y cómoda de hacerlo desde Javascript, y es de lo que tratará este artículo.

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`.

OJO: Recuerda que el objeto `.classList` aunque parece que devuelve un no es un array, sino un elemento que actúa como un array, por lo que puede carecer de algunos métodos o propiedades concretos. Si quieres convertirlo a un array real, utiliza `Array.from()`.

Veamos un ejemplo de uso de cada método de ayuda. Supongamos que tenemos el siguiente elemento HTML en nuestro documento. Vamos a acceder a el 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 por elementos del DOM

En algunas ocasiones en las que conocemos y controlamos perfectamente la estructura del código HTML de la página, nos puede resultar más cómodo tener a nuestra disposición una serie de propiedades para **navegar por la jerarquía** de elementos HTML relacionados.

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
```

Estas son las propiedades más habituales para navegar entre elementos HTML, sin embargo, tenemos otra modalidad un poco más detallada.

Navegar a través de nodos

La primera tabla que hemos visto nos muestra una serie de propiedades cuando trabajamos con . Sin embargo, si queremos hilar más fino y trabajar a nivel de , podemos utilizar las siguientes propiedades, que son equivalentes a las anteriores:

Propiedades de nodos HTML	Descripción
`childNodes`	Devuelve una lista de nodos hijos. Incluye nodos de texto y comentarios.
`parentNode`	Devuelve el nodo padre del nodo o si no tiene.
`firstChild`	Devuelve el primer nodo hijo.
`lastChild`	Devuelve el último nodo hijo.
`previousSibling`	Devuelve el nodo hermano anterior o si no tiene.
`nextSibling`	Devuelve el nodo hermano siguiente o si no tiene.

Estas propiedades suelen ser más interesantes cuando queremos trabajar sobre nodos de texto, ya que incluso los espacios en blanco entre elementos HTML influyen. Volvamos a trabajar sobre el documento HTML anterior, pero ahora utilizando este grupo de propiedades basadas en :

```
document.body.childNodes.length;  // 3
document.body.childNodes;         // [text, div#app, text]
```

```
document.body.parentNode;          // <html>

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

app.childNodes;                    // [text, div.header, text, p, text,
a, text]
app.firstChild.textContent;        // "                                "
app.lastChild.textContent;         // "                                "

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

a.previousSibling;                 // #text
a.nextSibling;                     // #text
```

Con todo esto, ya tenemos suficientes herramientas para trabajar a bajo nivel con las etiquetas y nodos HTML de un documento HTML desde Javascript.