Tema 4.1.
Modelo de objetos del documento. DOM

- Composición del DOM
- Estructura del árbol DOM
- DOM. Document
- Accediendo a los elementos
- DOM. Element
- Recorrer elementos
- Manipulando elementos

Document Object Model (DOM)

- Es el documento cargado en cada una de las pestañas del navegador
- Define la interfaz de programación de aplicaciones, su estructura lógica y la forma en que se accede y se manipulan los documentos. Es la representación interna de un documento de una aplicación y a través de él se pueden crear documentos, navegar por su estructura y agregar, modificar o eliminar elementos y contenido
- Está diseñado para usarse con cualquier lenguaje de programación Java, ECMAScript (JavaScript) y es compatible en todos los navegadores
- Proporciona métodos para acceder a los elementos individuales y a sus propiedades, permitiendo, con ello la creación de contenido dinámico
- Es un modelo lógico que usamos para describir la representación en forma de árbol de un documento. También usamos el término "árbol" cuando nos referimos a la disposición de los elementos de información a los que se puede llegar utilizando métodos de "caminar por el árbol"

Composición del DOM

En HTML DOM (Modelo de Objeto de Documento), todo es un **nodo**:

- **DOM Document**

  Cuando un documento HTML se carga en un navegador web, se convierte en un **objeto de documento**

  El objeto del document es el nodo raíz del documento HTML y el "dueño" de todos los demás nodos

- **DOM Elements**

  En el DOM HTML, el objeto element representa un elemento HTML

  Los objetos element pueden tener nodos secundarios de tipo elemento, nodos de texto, o nodos de comentario.

  Un **objeto NodeList** representa una lista de nodos, como la colección de un elemento HTML de nodos secundarios.

  Los elementos también pueden tener atributos. Los atributos son nodos de atributo

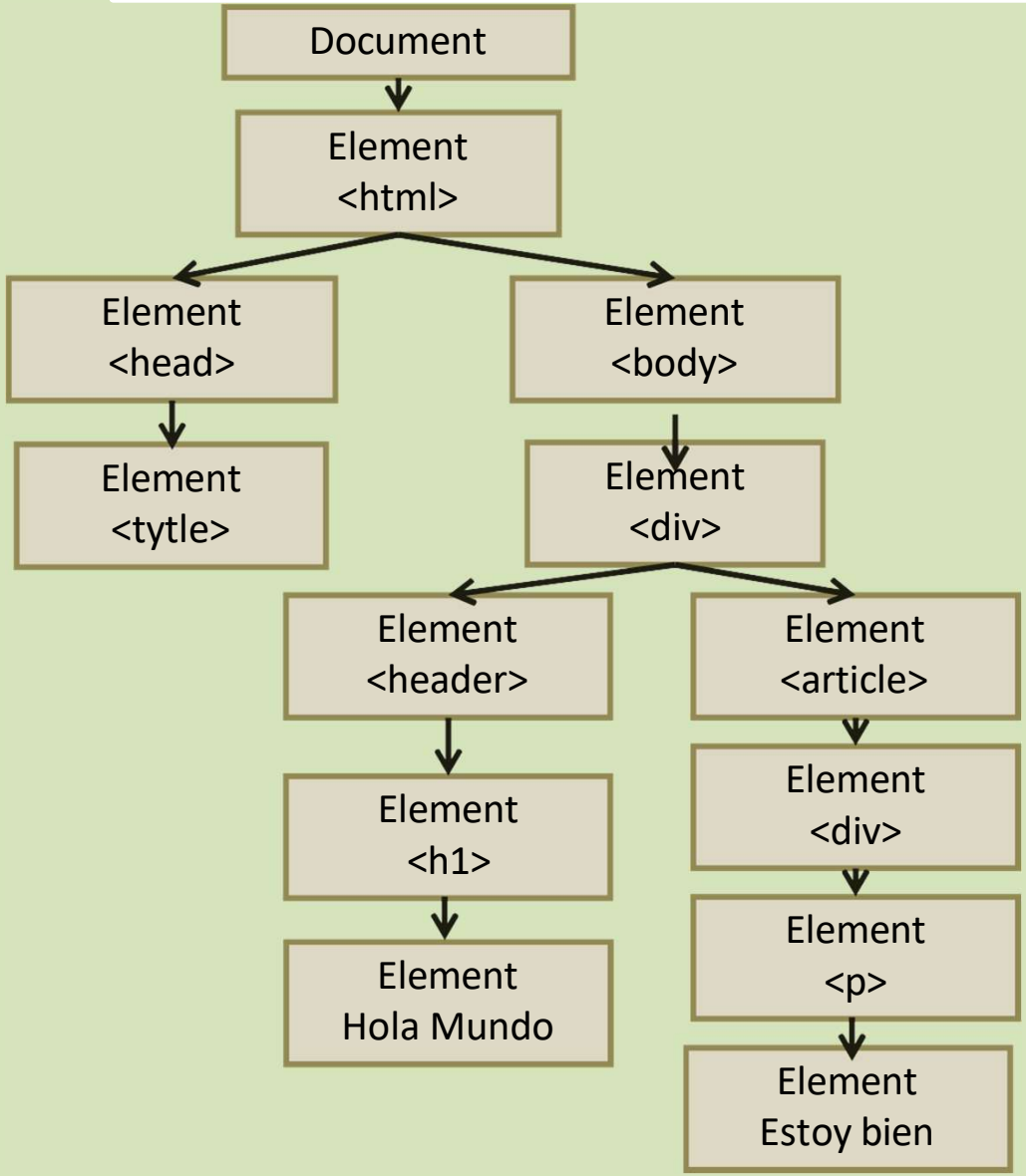
- **DOM Atributtes**

  En el DOM HTML, el **objeto Attr** representa un atributo HTML. Un atributo siempre pertenece a un elemento HTML

- **DOM Event**

  Trata los eventos producidos en la página

Estructura del árbol DOM




```
<html>
    <head>
        <tytle>
            Saludo
        </tytle>
    </head>
    <body>
        <div>
            <header>
                <h1>Hola Mundo</h1>
            </header>
            <article>
                <div>
                    <p>Estoy bien</p>
                </div>
            </article>
        </div>
    </body>
</html>
```

Estructura del árbol DOM

El DOM permite un acceso a la estructura de una página HTML mediante el mapeo de los elementos de esta página en un árbol de nodos. Cada elemento se convierte en un nodo y cada porción de texto en un nodo de texto

```
<!DOCTYPE html>
<html class=e>
   <head>
      <title>Pagina de prueba</title>
   </head>
   <body>
      Esto es un comentario
   </body>
</html>
```

```
Document
Doctype:  html
Elemento: class=e
   Elemento:  head
       Elemento:  title
            Texto:  Pagina de prueba
   Elemento:  body
       Texto: Esto es un comentario
```

En el árbol de nodos, al nodo superior (document) se le llama raíz

- Cada nodo, exceptuando el raíz, tiene un padre
- Un nodo puede tener cualquier número de hijos
- Los nodos que comparten el mismo padre, son hermanos
- El árbol de documento se obtiene cuando se ha cargado toda la página, lo sabremos con el evento

Objeto Document

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

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

Todos los **elementos HTML**, dependiendo del elemento que sean, tendrán un tipo de dato específico asociado. Veamos algunos ejemplos:
Existen muchos tipos de datos específicos, prácticamente uno por cada etiqueta HTML.

Tipo de dato específico	**Etiqueta**	**Descripción**
HTMLDivElement		Etiqueta divisoria (en bloque).
HTMLSpanElement		Etiqueta divisoria (en línea).
HTMLImageElement	<img>	Imagen.
HTMLAudioElement	<audio>	Contenedor de audio.

API nativa de Javascript

Javascript nos proporciona un conjunto de herramientas para trabajar de forma nativa con el DOM de la página. Las categorías importantes que deberíamos tener en cuenta son las siguientes:

Buscar etiquetas	Formas y métodos para buscar elementos en el DOM como .querySelector().
Modificar contenido	Acceder y modificar el contenido de una etiqueta HTML del DOM desde Javascript.
Gestionar atributos o clases	Uso de varias de las API de Javascript para manipular clases CSS o atributos HTML.
Crear etiquetas	Métodos y consejos para crear elementos en el DOM y trabajar con ellos.
Insertar etiquetas	Varias de las API de Javascript para insertar elementos en el DOM de una página.

¿Utilizar el DOM?

En la industria de la programación web, se suele optar por **evitar la manipulación del DOM de forma directa**. Esto es así porque trabajar con el DOM requiere destreza, conocimiento y es un proceso lento que necesita experiencia para que no resulte incómodo o se realicen acciones contraproducentes.

Por esta razón, habitualmente, se **delega** la tarea de la manipulación del DOM a una **librería o framework de terceros** (*generalmente a frameworks como React o Vue, por ejemplo*).

El uso de estos frameworks nos simplifica el trabajo a cambio de dos cosas:

- Aprender otras estrategias propias del framework (*habitualmente más sencillas y cómodas*)
- Perder un poco de rendimiento y control sobre el DOM de nuestra aplicación o web

Sin embargo, saber trabajar con el DOM es un skill muy importante que te dotará de habilidades para ser mucho más eficiente a la hora de pensar y resolver problemas relacionados con la estructura de una página.

Métodos de Búsqueda Tradicionales

Existen **4 métodos tradicionales** de Javascript para manipular el DOM. Se denominan tradicionales porque son los que existen en Javascript desde versiones más antiguas.

El método getElementById()

Busca un elemento HTML con el id especificado. 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, ya que no debería existir más de uno:

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

En el caso de no encontrar el elemento indicado, devolverá NULL.
Hoy en día, es más habitual localizar elementos por sus clases, ya que no todos los elementos son únicos o tienen un id establecido.

Métodos de Búsqueda Tradicionales

El método getElementsByClassName()

Permite buscar los elementos que tengan la clase especificada en class.
Puede devolver **varios elementos**, ya que pueden existir varios elementos con la misma clase:

```
<div class="container">
<div class="item">Item 1</div>
<div class="item">Item 2</div>
<div class="item">Item 3</div>
</div>
```

```
const elements = document.getElementsByClassName("item");
console.log(elements); // [div, div, div]
console.log(elements[0]); // Primer item encontrado: <div class="item"></div>
console.log(elements.length); // 3
```

Devuelve un Array de elementos, en concreto un HTMLCOLLECTIONS. En el caso de no encontrar ninguno, devolverá un Array vacío: [].

Métodos de Búsqueda Tradicionales

El método getElementsByName()

Busca elementos HTML por el valor que tienen en el atributo name, algo muy habitual en **formularios HTML**:

```
const nicknames = document.getElementsByName("nickname");
console.log(nicknames); // [input]
```

Devuelve un Array de elementos (HTMLCOLLECTIONS), podría haber varios elementos HTML con ese mismo name.

El método getElementsByTagName()

Busca elementos por el tipo de etiqueta HTML.

```
const divs = document.getElementsByTagName("div");
console.log(divs); // [div, div, div, div]
```

```
<div class="item">Item 1</div>
<div class="item">Item 2</div>
<div class="item">Item 3</div>
</div>
```

Devuelve un Array (HTMLCOLLECTIONS) con **4 elementos**. En el caso de no encontrar ninguno, nos devolvería un array vacío [].

Métodos de Búsqueda Tradicionales

El tipo HTMLCollection vs Array

Los tres últimos métodos no devuelven realmente un ARRAY, realmente devuelven un HTMLCollection:

```
const elements = document.getElementsByTagName("div");
elements instanceof Array; // false
elements instanceof HTMLCollection; // true
elements.constructor.name; // 'HTMLCollection'
```

Los tipos de datos HTMLCollection funcionan como un array, aunque no son exactamente arrays. Si se ejecuta sobre el array un método .forEach(), .map(), .filter() o similar, se obtiene un error porque no existen.
Para evitarlo, se usa Array.from() o la desestructuración, y convertirlo fácilmente en un ARRAY real:

```
const collection = document.getElementsByTagName("div");
const array = [...document.getElementsByTagName("div")];
collection.constructor.name; // 'HTMLCollection'
array.constructor.name; // 'Array'
```

Métodos de Búsqueda Tradicionales

¿Diferencia entre un ARRAY y un HTMLCollection ?

La estructura HTMLCollection es una colección **viva** de elementos, lo que significa que si se modifican elementos del DOM, se actualizan en la estructura, al contrario que un ARRAY.
Ejemplo:

```
// HTMLCollection
const collection = document.getElementsByTagName("div");
collection.length; // 63
collection[62].remove(); // Eliminamos del DOM el último elemento
collection.length; // 62 (Cómo el elemento no existe en el DOM, se borra)
// Array
const collection = [...document.getElementsByTagName("div")];
collection.length; // 63
collection[62].remove(); // Eliminamos del DOM el último elemento
collection.length; // 63 (Sigue teniendo el mismo número de elementos)
```

Aunque pueda parecer más interesante el primero, lo cierto es que es más frágil y mucho más difícil de predecir una estructura que puede cambiar sus elementos con el paso del tiempo.

Métodos de Búsqueda Modernos

En este grupo tenemos dos métodos que son equivalentes a todos los métodos tradicionales, pero más potentes, pudiendo realizar todo lo que hacíamos con los **métodos tradicionales** e incluso muchas más cosas gracias a **CSS**.

El método querySelector()

Devuelve el primer elemento que encuentra que encaja con el **selector CSS** indicado por parámetro:

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

- El primer ejemplo sería equivalente a utilizar un .getElementById(), sólo que en la versión de .querySelector() indicamos por parámetro un SELECTOR, y en el primero le pasamos un simple STRING. Le indicamos un # porque se trata de un id.
- En el segundo ejemplo, recuperamos el primer elemento con clase info que esté dentro de otro con clase main. En los métodos tradicionales, tendríamos que hacer varias llamadas. Con .querySelector() se hace directamente sólo con una.

El método querySelector() siempre devuelve un solo elemento: el primero que encuentra. En el caso de no coincidir con ninguno, devuelve NULL.

Métodos de Búsqueda Modernos

El método querySelectorAll()

Realiza una búsqueda de elementos en el DOM, y 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");
```

El método .querySelectorAll() siempre nos devolverá un ARRAY de elementos. En el caso de no encontrar ninguna coincidencia, nos devolverá un array de 0 elementos.

Búsquedas acotadas

Al realizar una búsqueda de un elemento particular y guardarlo en una variable o constante, podemos volver a realizar una nueva búsqueda posteriormente sobre este elemento, en lugar del DOM íntegro document:

```
const menu = document.querySelector("#menu");
const links = menu.querySelectorAll("a");
```

Sin embargo, si controlamos un poco de CSS, y nos interesa únicamente el elemento final, puede ser mucho más sencillo realizar lo siguiente:

```
const links = document.querySelectorAll("#menu a");
```

Métodos de Búsqueda Modernos

El tipo NodeList vs HTMLCollection

Aunque hemos dicho que el método querySelectorAll() devuelve siempre un ARRAY, realmente devuelve un tipo de dato NodeList, que es muy similar a un ARRAY , pero no es exactamente lo mismo.

```
const elements = document.querySelectorAll("div");
elements.map // undefined
const elements = [...document.querySelectorAll("div")];
elements.map // ƒ map() { [native code] }
```

//Para recorrer todos los elementos de elementos se usaría el **for …. of**

En el primer caso, la estructura devuelva (*es un NodeList*) no tiene el método .map(), que utilizamos para transformar elementos de un array.

En el segundo caso hemos hecho una desestructuración de arrays para sacar todos los elementos de la estructura y reestructurarla en un ARRAY. Por esta razón en el primer caso no podemos usar .map() y en el segundo caso, sí.

Si no vas a realizar tareas donde necesites estas estructuras de datos, se recomienda no convertirlo en un array, pero si necesitas trabajar con métodos como .map() o .filter(), por ejemplo, puedes destructurarlo o convertirlo mediante un Array.from().

Métodos de Búsqueda Modernos

Navegar a través de elementos

Las propiedades que veremos a continuación devuelven información de otros elementos relacionados con el elemento en cuestión. En este caso estamos trabajando con el tipo ELEMENT :

- Propiedad **children** nos ofrece un ARRAY 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.
- Propiedad **firstElementChild** sería un acceso rápido a children[0]
- Propiedad **lastElementChild** sería un acceso rápido al último elemento hijo.
- Propiedades **previousElementSibling** y **nextElementSibling** nos devuelven los elementos hermanos anteriores o posteriores, respectivamente.
- Propiedad **parentElement** nos devolvería el padre del elemento en cuestión. En el caso de no existir alguno de estos elementos, nos devolvería NULL .

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

Métodos de Búsqueda Modernos

```
// Trabajando sobre <body>
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.
En esta modalidad, cuando hablamos de ARRAY trabajamos a nivel de HTMLCOLLECTION.

Métodos de Búsqueda desde otros elementos

Existen algunos métodos interesantes para encontrar elementos desde otros elementos. Los métodos que mencionamos son los siguientes:

El método matches()

Nos permite comprobar si un elemento coincide con un selector CSS indicado por parámetro. El método devuelve true si el elemento coincide con el selector indicado. Si no lo encuentra, devuelve false.

```
const container = document.querySelector(".container");
const sandbox = document.querySelector(".sandbox");
container.matches(".container"); // true
sandbox.matches(".container .sandbox"); // true (detecta ancestros)
```

```
<div class="container">
    <div class="sandbox">
    </div>
</div>
```

El método closest()

Nos permite encontrar el primer elemento más cercano que coincide con un selector CSS indicado por parámetro. Si no lo encuentra devuelve NULL.

```
const item3 = document.querySelector(".item-3");
item3.closest(".container"); // <div class="container"></div>
item3.closest(".subitem"); // null
```

Busca elementos en dirección hacia el padre, devolviendo el más cercano, pero nunca hacia los elementos hijos.

```
<div class="container">
    <div class="sandbox">
        <div class="item item-1"></div>
        <div class="item item-2"></div>
        <div class="item item-3">
            <div class="subitem"></div>
        </div>
    </div> </div>
```

Navegar a través de nodos

Si queremos hilar más fino, podemos trabajar a nivel de elementos de tipo NODE , utilizando las siguientes propiedades, que son equivalentes a las anteriores, pero trabajando con la unidad más básica: **nodos**.
Estas propiedades suelen ser más interesantes cuando queremos trabajar sobre nodos de texto, ya que incluso los espacios en blanco entre elementos HTML influyen.

Propiedades de nodos HTML	Descripción	Si está vacío...
.childNodes	Devuelve una lista de nodos hijos (incluye nodos de texto y comentarios).	NULL
.parentNode	Devuelve el nodo padre del nodo.	NULL
.firstChild	Devuelve el primer nodo hijo.	NULL
.lastChild	Devuelve el último nodo hijo.	NULL
.previousSibling	Devuelve el nodo hermano anterior.	NULL
.nextSibling	Devuelve el nodo hermano siguiente.	NULL

Navegar a través de nodos

```
// Trabajando sobre <body>

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
```

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

OJO!!Los intros o espacios en blanco entre las etiquetas generan un NODE (como texto).

Con todo esto, ya tenemos suficientes herramientas para trabajar a bajo nivel con las etiquetas y nodos HTML de un documento HTML desde Javascript.
En esta modalidad, cuando hablamos de ARRAY trabajamos a nivel de NODELIST .

Acceder al contenido del DOM

La propiedad .nodeName

Con la propiedad .nodeName podemos obtener el nombre de la etiqueta en cuestión. Veamos un ejemplo para entenderlo mejor:

```
// <div class="container"></div>
const element = document.querySelector("div");
element.nodeName; // DIV
```

Esta propiedad nodeName nos devuelve el nombre del nodo, 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.

Si no tenemos una etiqueta, sino un nodo de texto o un comentario, devolvería #text o #comment.

Acceder al contenido del DOM

La propiedad .textContent

Es la propiedad recomendada para hacer modificaciones de texto. Nos devuelve el **contenido de texto** de un elemento HTML concreto. Es muy útil para obtener (*o modificar*) **sólo el texto** de un elemento, obviando el marcado o etiquetado HTML.

Ejemplo:

```
<div class="container">
        <div class="parent">
        <p>Hola a todos.</p>
        <p class="message">Mi nombre es
<strong>Mario</strong>.</p>
        </div>
</div>
```

```
const element = document.querySelector(".message");
element.textContent; // "Mi nombre es Mario."
element.textContent = "Hola a todos"; // Modificamos el contenido de texto
element.textContent; // "Hola a todos"
```

Podemos utilizar la propiedad .textContent para acceder a la información de texto que contiene y para reemplazar su contenido.

Observa que en el HTML tenemos una etiqueta . 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.

Acceder al contenido del DOM

La propiedad .innerText

Es muy similar a .textContent, pero tiene una diferencia clave: accede al texto **renderizado visualmente por el navegador**.

```
<div class="container">
        <p>Hola a todos.</p>
        <p>Me llamo <strong>Mario</strong>. <mark style="display: none">New message</mark></p>
        <p hidden>Esto es un mensaje posterior oculto semánticamente.</p>
        <details> <summary>Más información</summary>
                <div>Esto es un desplegable que está colapsado.</div>
        </details>
</div>
```

El contenido de la etiqueta <mark> está oculto mediante CSS.

- El contenido de la tercera etiqueta <p> está oculto mediante hidden.
- El contenido de la etiqueta dentro del <details> está oculto porque el acordeón está colapsado.

Si consultamos el contenido de texto del .container mediante .textContent nos devolvería todo el contenido de texto de todas las etiquetas. Con .innerText nos devolvería **sólo lo renderizado visualmente**:

```
const element = document.querySelector(".container");
element.innerText; // "Hola a todos. // // Me llamo Mario. // // Más información" (Y con formato)
```

Acceder al contenido del DOM

La propiedad .outerText

Funciona exactamente igual que .innerText para obtener información. Sin embargo, tiene una pequeña diferencia si la utilizamos para modificar texto:

- En el caso de utilizar .innerText, modificamos el contenido de la etiqueta.
- En el caso de utilizar .outerText, hacemos lo mismo, pero incluyendo la etiqueta en sí.

```
// <section><div class="container"></div></section>
const element = document.querySelector(".container");
element.innerText = "Me llamo Mario"; // <section><div class="container">Me llamo Mario</div></section>
element.outerText = "Me llamo Mario"; // <section>Me llamo Mario</section>
```

Observa que en el primer caso, ha añadido el texto en el interior de la etiqueta . En el segundo caso, está reemplazando la propia etiqueta , quedándose sólo con el texto.

Acceder al contenido del DOM

Insertando contenido HTML

Un aspecto muy importante a tener en cuenta es que las propiedades que hemos visto sirven sólo para **contenido de texto**. Si intentamos insertar contenido HTML de esta forma:

```
const element = document.querySelector(".container");
element.textContent = "<h1>Hola</h1>";
```

Obtendremos que lo que ocurre es que se inserta literalmente el texto <h1>Hola</h1>, en lugar de renderizarse una etiqueta <h1> como encabezado, con el texto Hola. Si lo que queremos es renderizar contenido lo tenemos que hacer con innerHTML.
Para añadir contenido HTML a las etiquetas usaremos los métodos:

Propiedades	Descripción
.nodeType	Devuelve un número del tipo de elemento. Sólo lectura.
Contenido HTML	
.innerHTML	Devuelve (o cambia) el contenido HTML del interior del elemento.
.outerHTML	Idem a .innerHTML pero incluye también la propia etiqueta HTML.
.setHTMLUnsafe(html)	Alternativa moderna a innerHTML. Permite Shadow DOM declarativo.

Acceder al contenido del DOM

La propiedad .nodeType

La propiedad .nodeType no tiene relación directa con este apartado, pero es una forma excelente de identificar el tipo de elemento con el que estamos trabajando:

```
const element = document.querySelector("div");
element.nodeType; // 1
```

Ese valor numérico identifica el tipo de elemento con el que estamos trabajando. Los principales y más importantes, son estos tres, aunque existen algunos más:

Valor	Tipo de Elemento	Descripción	Ejemplo
1	Element.ELEMENT_NODE	Un elemento HTML.	Hello!
3	Element.TEXT_NODE	Un nodo de texto.	Hello!
8	Element.COMMENT_NODE	Un comentario HTML.	<!-- Hello! -->

Acceder al contenido del DOM

La propiedad .innerHTML

Nos permite acceder al contenido de un elemento, pero en lugar de devolver su contenido de texto como lo hace .textContent, esta propiedad nos devuelve su **contenido HTML**, es decir, su marcado HTML. Esto tiene varias implicaciones que explicaremos más adelante.

Ejemplo con la diferencia entre .textContent y .innerHTML:

```
const element = document.querySelector(".message");
element.innerHTML; // "Mi nombre es <strong>Mario</strong>."
element.textContent; // "Mi nombre es Mario."
```

.textContent se enfoca en obtener el contenido de texto, mientras que .innerHTML se enfoca en el contenido HTML.

De la misma forma que .textContent, también podemos usar .innerHTML para modificar el contenido.

```
element.innerHTML = "<strong>Importante</strong>"; // Se lee "Importante" (en negrita)
element.textContent = "<strong>Importante</strong>"; // Se lee "<strong>Importante</strong>"
```

Acceder al contenido del DOM

Parseo de marcado HTML

La propiedad .innerHTML comprueba y parsea el marcado HTML escrito (*corrigiendo si hay errores*) antes de realizar la asignación.

Ejemplo:

```
const container = document.querySelector(".container");
container.innerHTML = "<strong>Mario";
container.innerHTML // "<strong>Mario</strong>"
```

Aunque hemos insertado el HTML incompleto con .innerHTML, si examinamos el contenido, podemos ver que está completo. Esto ocurre porque el navegador parsea e intenta que el código HTML sea correcto en todo momento.

Propiedad .outerHTML

Es muy similar a .innerHTML. Mientras este último devuelve el código HTML del **interior** de un elemento HTML, .outerHTML devuelve el código HTML **desde el exterior**, es decir, incluyendo al propio elemento implicado:

```
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>"
```

Aributos del DOM

Las etiquetas HTML tienen ciertos atributos que definen el comportamiento de la etiqueta. Existen atributos comunes a todas las etiquetas HTML, y atributos que sólo existen para determinadas etiquetas HTML. Además, un atributo puede tener un valor o ser un atributo , es decir, simplemente estar presente y no tener ningún valor indicado.

Acceder a atributos HTML

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

En algunos casos como el del último ejemplo, se indica className en lugar de class. Esto ocurre porque es una palabra reservada para las clases de Javascript.

Aunque es posible asignar a la propiedad className varias clases en un separadas por espacio, se recomienda utilizar la propiedad classList para manipular clases CSS (más adelante)

Atributos del DOM

Obtener atributos HTML

Aunque la forma anterior es la más rápida, tenemos algunos métodos para obtener los atributos HTML de forma clara y literal, sin problemas como los de className:
Se puede usar **hasAttributes**() o **hasAttribute(**attr) para saber qué atributos HTML tiene definidos una etiqueta.
El método **getAttributeNames**() nos devuelve la lista de atributos que tiene una etiqueta, y el método **getAttribute(attr**) nos da el valor que tiene un atributo HTML específico.

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

Javascript:

```
const element = document.querySelector("#page");
element.hasAttributes(); // true (tiene 3 atributos)
element.hasAttribute("data-number"); // true (data-number existe)
element.hasAttribute("disabled"); // false (disabled no existe)
element.getAttributeNames(); // ["id", "data-number", "class"]
element.getAttribute("id"); // "page"
```

Atributos del DOM

Modificar o eliminar atributos HTML

A partir del siguiente código HTML:

Vamos a modificar sus atributos HTML utilizando **setAttribute**(), que puede servir tanto para añadir nuevos atributos que no existían como para modificar los que ya existen, y **removeAttribute**() para eliminar atributos:

```
const element = document.querySelector("#page");
element.setAttribute("data-number", "10"); // Cambiar data-number a 10
element.removeAttribute("id"); // Elimina el atributo id
element.setAttribute("id", "page"); // Vuelve a añadirlo
```

Sin embargo, hay un caso especial que es digno de mención, como son los atributos booleanos.

Atributos del DOM

Caso especial: Atributos booleanos

Hay que un caso especial, cuando los atributos HTML son BOOLEANOS , es decir, que no tienen indicado ningún valor.
Si esto lo hacemos con el método setAttribute() y le indicamos un booleano, no tendremos exactamente lo que buscamos.

```
const button = document.querySelector("button");
button.setAttribute("disabled", true); // ✗ <button disabled="true">Clickme!</button>
button.disabled = true; // ✓ <button disabled>Clickme!</button>
button.setAttribute("disabled", ""); // ✓ <button disabled>Clickme!</button>
```

La forma correcta de establecerlos es indicar un vacío. Automáticamente, el navegador sabrá que una cadena de texto vacía es un booleano y ocultará su valor. También se puede hacer mediante una propiedad Javascript, asignándole un booleano, y entonces añadirá el atributo HTML automáticamente.
El método .**toggleAttribute(attr**, force) es más sencillo para estos casos. Añade el atributo que le pasas por parámetro si no existe, y lo elimina si ya existe:

```
button.toggleAttribute("disabled"); // Como ya existe "disabled", lo elimina
button.toggleAttribute("hidden"); // Como no existe "hidden", lo añade
```

Si se le proporciona el force, si es verdadero: añade el atributo, si es falso: lo elimina

Atributos del DOM

Acceso a las clases o estilos CSS

Las formas principales de modificar las clases o los estilos CSS mediante el DOM son las siguientes:
Veamos detalladamente cada uno de ellos, pero ahora vamos a centrarnos en el primero.

La propiedad .className

Con dicha propiedad podemos acceder al valor del atributo HTML class como un , tanto para obtenerlo como para cambiarlo o reemplazarlo.
La propiedad .className viene a ser la modalidad directa, rápida y equivalente a utilizar:

- El getter .getAttribute("class") que devolvería el valor del atributo HTML class.
- El setter .setAttribute("class", value) que cambiaría el valor del atributo HTML class.

```
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 = "element shine light-theme";
div.setAttribute("class", "element shine light-theme");
```

Atributos del DOM

Trabajar con .className es sencillo, simple y rápido, pero tiene una limitación cuando trabajamos con elementos HTML con **múltiples clases CSS**, que es la mayoría de los casos. Si queremos realizar una manipulación sólo en una clase CSS concreta, dejando las demás intactas, se puede hacer, pero .className se vuelve poco práctico ya que hay que escribir más código.

En este ejemplo donde vamos a cambiar la clase dark-theme por light-theme:

```
const div = document.querySelector(".element"); // Obtener clases CSS
div.className; // "element shine dark-theme"
const classnames = div.className.split(" "); // ["element", "shine", "dark-theme"] classnames[2] =
"light-theme";
div.className = classnames.join(" "); // "element shine light-theme"
```

Resulta un poco incómodo. La forma más interesante de manipular clases CSS desde Javascript es mediante la propiedad .classList, que tiene un conjunto de métodos muy útiles y prácticos.

Atributos del DOM

El método checkVisibility()

Con él, podemos comprobar si el elemento en cuestión es potencialmente visible al usuario en la página o no. Una comprobación que, sin duda, es muy útil para los desarrolladores.

```
const element = document.createElement("div");
element.checkVisibility(); // false (no está en el DOM)
document.body.append(element); // Lo añadimos al <body>
element.checkVisibility(); // true (está en el DOM)
element.setAttribute("hidden", ""); // Le añadimos el atributo `hidden`
element.checkVisibility(); // false (está oculto por hidden)
element.removeAttribute("hidden");
element.className = "hide"; // Le añadimos una clase que tiene display: none
element.checkVisibility(); // false (está oculto por display: none)
```

Es capaz de comprobar si tiene asociado un display a none, si se usa content-visibility a hidden, si tiene una propiedad opacity a 0, si tiene una propiedad visibility a hidden, si está desconectado del DOM o aún no se ha añadido.

Atributos del DOM

Es posible añadir un objeto de opciones a checkVisibility(), este objeto puede tener varias propiedades:

Propiedad	Descripción	Valor por defecto
contentVisibilityAuto	Comprueba si content-visibility está (o hereda) a auto.	false
opacityProperty	Comprueba si opacity está (o hereda) a 0 y está oculto visualmente.	false
visibilityProperty	Comprueba si visibility está (o hereda) a hidden o collapse.	false

Las propiedades checkOpacity y checkVisibility están disponibles porque eran los nombres anteriores de las dos últimas.

```
const element = document.createElement("div"); // Creamos un <div>
document.body.append(element); // Lo añadimos a <body>
element.style.opacity = 0; // Lo ocultamos (añadimos CSS)
element.checkVisibility(); // true (no comprueba opacity)
element.checkVisibility({ opacityProperty: true }); // false (comprueba opacity)
```

Atributos del DOM

La propiedad u objeto .classList, nos devolverá un ARRAY (en concreto *un DOMTokenList)* de clases CSS de dicho elemento.

 //

Acceder a clases CSS

Podemos acceder a la lista de clases con .classList y al número de clases con .classList.length. Podemos acceder a la propiedad .classList.values para obtener un String como lo haría .className:

```
const element = document.querySelector("#page"); // ¿Qué clases tiene?
element.classList; // ["info", "data", "dark"] (DOMTokenList)
element.classList.value; // "info data dark" (String)
element.classList.length; // 3 // Convertirlas a array
Array.from(element.classList) // ["info", "data", "dark"] (Array)
[...element.classList]; // ["info", "data", "dark"] (Array) // Consultarlas
element.classList.item(0); // "info"
element.classList.item(1); // "data"
element.classList.item(3); // null
```

El objeto .classList no devuelve un array, por lo que puede carecer de algunos métodos o propiedades concretos de arrays. Si quieres convertirlo a un array real, utiliza Array.from() o desestructuración con [...div.classList].

El método .classList.item() que nos devuelve un String con la clase específica en esa posición. Si no existe una clase en esa posición, nos devolverá NULL.

Atributos del DOM

Añadir y eliminar clases CSS

Los métodos .classList.add() y .classList.remove() permiten indicar una o múltiples clases CSS a añadir o eliminar. Ejemplo:

```
const element = document.querySelector("#page");
element.classList.add("uno", "dos");
element.classList; // ["info", "data", "dark", "uno", "dos"]
element.classList.remove("uno", "dos");
element.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, no ocurrirá nada.

Comprobar si existen clases CSS

Con .classList.contains() podemos comprobar si existe una clase en un elemento HTML, ya que nos devuelve un Booleano indicándonos si está presente o no:

```
const element = document.querySelector("#page");
element.classList; // ["info", "data", "dark"]
element.classList.contains("info"); // Devuelve `true` (existe esa clase)
element.classList.contains("warning"); // Devuelve `false` (no existe esa clase)
```

Esto puede resultar interesante en algunas situaciones, donde queremos averiguar mediante Javascript si existe una clase.

Atributos del DOM

Conmutar o alternar clases CSS

El método .classList.toggle(), **añade o elimina 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 element = document.querySelector("#page");
element.classList; // ["info", "data", "dark"]
element.classList.toggle("info"); // Como "info" existe, lo elimina. Devuelve "false" element.classList; // ["data", "dark"]
element.classList.toggle("info"); // Como "info" no existe, lo añade. Devuelve "true" element.classList; // ["info", "data", "dark"]
```

.toggle() devuelve un Booleano que sindicará si, tras la operación, la clase sigue existiendo o no. Al contrario que .add() o .remove(), sólo se puede indicar una clase CSS por parámetro.

Reemplazar una clase CSS

El método .classList.replace() nos permite reemplazar la primera clase indicada por parámetro, por la segunda.

```
const element = document.querySelector("#page");
element.classList; // ["info", "data", "dark"]
element.classList.replace("dark", "light"); // Devuelve `true` (se hizo el cambio)
element.classList.replace("warning", "error"); // Devuelve `false` (no existe warning)
```

Atributos del DOM

Hay dos formas principales de que un elemento HTML tenga estilos CSS: Asociándole una **clase HTML** con una colección de estilos, o añadiéndole un **atributo style** con un conjunto de propiedades y valores.
Lo recomendable suele ser utilizar las **clases**, ya que es más organizado y fácil de mantener, pero hay algunas excepciones donde podemos utilizar los **estilos CSS** y pueden ser especialmente interesantes.

Los estilos en línea de un elemento

Ejemplo de cómo se añaden **estilos en línea** a un elemento HTML:

```
<div id="title" style="background-color: indigo; padding: 25px; color: var(--color, #000);"> <h1 style="--color: white">El objeto style</h1> </div>
```

Es mucho mejor crear una clase con esta lista de estilos y añadirle la clase a la etiqueta HTML. Hay una excepción donde es realmente útil estos estilos en línea: con las variables CSS.
Puedes ver que el elemento <h1> en lugar de establecerle una propiedad CSS, le hemos establecido una variable CSS. Esto suele ser muy flexible, y nos permite indicar estilos dependiendo del elemento HTML donde se encuentra.

Atributos del DOM

Acceso a los estilos CSS

Si queremos acceder a los estilos en línea definidos, aunque podríamos hacerlo con .getAttribute('style'), lo haremos a través del objeto especial .style:

```
const title = document.querySelector("#title");
title.getAttribute("style") // "background-color: indigo; padding: 25px; ..."
title.style // CSSStyleDeclaration { 0: ..., 1: ..., ... }
```

- Con .getAttribute("style") nos devuelve un con el texto literal del atributo.
- Accediendo al objeto .style, nos devuelve un objeto especial CSSStyleDeclaration.

Este objeto especial contiene todas las propiedades CSS asignables en ese elemento, y podremos acceder a ellas como un objeto normal .

Atributos del DOM

El objeto .style

Con el objeto style podemos acceder a propiedades especificas, para obtener su valor.
Nos mostrará que el objeto tiene muchísimas propiedades (*una por cada propiedad CSS*).
Sin embargo, las propiedades definidas son las que tendrán que tendrán valores:

```
title.style.backgroundColor; // "indigo"
title.style.color; // "var(--color, #000)"
title.style.padding; // "25px"
title.style.paddingLeft; // "25px"
title.style.paddingTop; // "25px"
title.style["paddingTop"] // "25px"
title.style["padding-top"] // "25px"
title.style.fontFamily // ""
```

Observa que si la propiedad CSS se llama padding-top, en Javascript debes acceder a ella en camelCase, es decir, como paddingTop. Si accedes mediante la nomenclatura de corchetes, puedes acceder de ambas formas. Si un valor no está definido, devuelve cadena vacía.
Aunque podemos acceder de esta forma a las propiedades CSS definida en los estilos en línea, esta es una forma legacy y antigua, que aunque sigue funcionando, ofrece menos ventajas que utilizar la familia de las siguientes funciones:

Atributos del DOM

El método getPropertyValue()

Este método sirve tanto para propiedades CSS como para variables CSS (*que no están soportadas de la forma anterior*):

```
title.style.getPropertyValue("background-color"); // "indigo"
title.style.getPropertyValue("color"); // "var(--color, #000)"
title.style.getPropertyValue("padding"); // "25px"
title.style.getPropertyValue("padding-left"); // "25px"
title.style.getPropertyValue("padding-top"); // "25px"
title.style.getPropertyValue("font-family"); // ""
const h1 = title.querySelector("h1");
h1.style.getPropertyValue("--color"); // "white"
```

De esta forma eliminamos el inconveniente de tener que estar traduciendo a camelCase, además de tener soporte con variables CSS.

El método setProperty()

Modifica las variables CSS.

```
const h1 = title.querySelector("h1");
h1.style.setProperty("--color", "gold");
h1.style.getPropertyValue("--color"); // "gold"
```

Atributos del DOM

El método global getComputedStyle()

Obtiene los **estilos computados** de un elemento, que son los estilos CSS finales que ya han sido procesados y calculados por el navegador, después de aplicar herencia, cálculos, reglas, etc.

Si necesitamos obtener el valor CSS final de alguna propiedad que no ha sido definida a través de estilos en línea, utilizar getComputedStyle() puede ser una buena opción. Al igual que en el objeto style, el método getComputedStyle() devuelve un objeto CSSStyleDeclaration, que contiene propiedades con nombre de la propiedad CSS y el valor asociado:

```
const element = document.querySelector(".element");
getComputedStyle(element); // CSSStyleDeclaration { 0: ..., 1: ..., ... }
// Elemento tiene una clase con background: #222
getComputedStyle(element).background // "rgb(34, 34, 34) none repeat scroll 0% 0% / auto
padding-box border-box"
getComputedStyle(element).backgroundColor // "rgb(34, 34, 34)"
```

Observa que aunque los estilos originales tengan un valor de #222, los estilos computados devuelven rgb(34, 34, 34), ya que han sido procesados por el navegador, por lo que no debes esperar que se devuelva el valor exactamente igual.

Atributos del DOM

Acceso a los pseudoelementos

Con getComputedStyle(), también podemos acceder a los estilos CSS de los pseudoelementos, como por ejemplo ::after y ::before, los cuales no es posible recuperarlos directamente con querySelector(), por ejemplo:

```
const element = document.querySelector(".element");
getComputedStyle(element, "::after"); // CSSStyleDeclaration { 0: ..., 1: ..., ... }
```

Manipulación del DOM

EJERCICIOS DOM 1