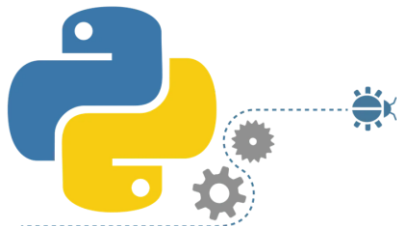




Curso FullStack Python

Codo a Codo 4.0



Clase 14

JavaScript Parte 5



Páginas estáticas vs. Páginas dinámicas

Cuando comenzamos en el mundo del **desarrollo web**, normalmente empezamos a 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»** (no responden al comportamiento 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.

La forma de crear páginas dinámicas es dotar de código de programación a nuestro documento HTML, a través de la **manipulación de sus componentes**, lo que se conoce como **DOM** (*Document Object Model*)

Más información: <https://lenguajejs.com/javascript/dom/que-es/>

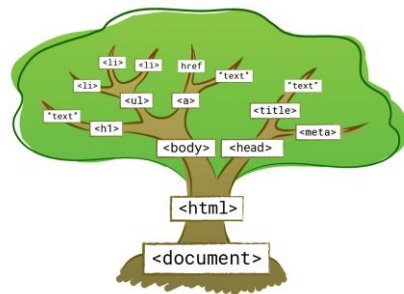
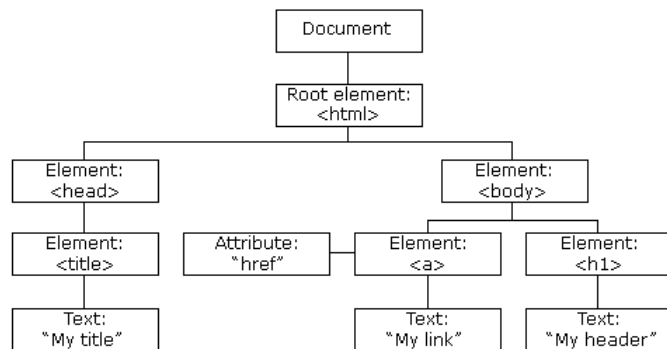
DOM

DOM: Document Object Model, es 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).

Desde Javascript podemos modificar esta estructura de forma dinámica, añadiendo nuevas etiquetas, modificando o eliminando otras, cambiando sus atributos HTML, añadiendo clases, cambiando el contenido de texto, etc... Estas tareas pueden automatizarse, 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...

Recordemos que la estructura HTML está compuesta por etiquetas y que las etiquetas están anidadas dentro de otras (*ejemplo: dentro del body puedo tener un div y adentro del div otro div y a su vez un párrafo*). A esto se lo denomina el **árbol DOM o simplemente DOM**. Es un árbol porque hay una **jerarquía**, hay ramas que se abren, que las vemos cuando utilizamos Visual Studio Code que nos permite ver la estructura, las indentaciones, las tabulaciones.

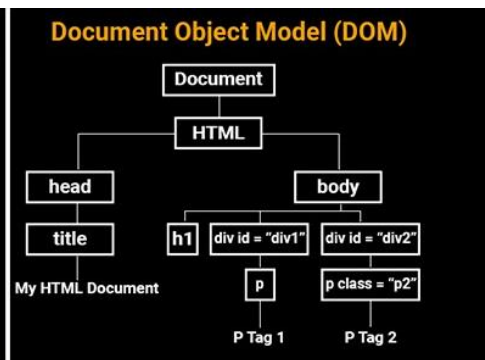
DOM: Estructura jerárquica



Estructura jerárquica del árbol DOM

HTML Document

```
1 <html>  
2   <head>  
3     <title>My HTML Document</title>  
4   </head>  
5  
6   <body>  
7     <h1>Headings</h1>  
8     <div id="div1">  
9       <p>P Tag 1</p>  
10    </div>  
11    <div id="div2">  
12      <p class="p2">P Tag 2</p>  
13    </div>  
14  </body>  
15 </html>
```



Fuente: <https://javadesde0.com/introduccion-a-document-object-model-dom/>

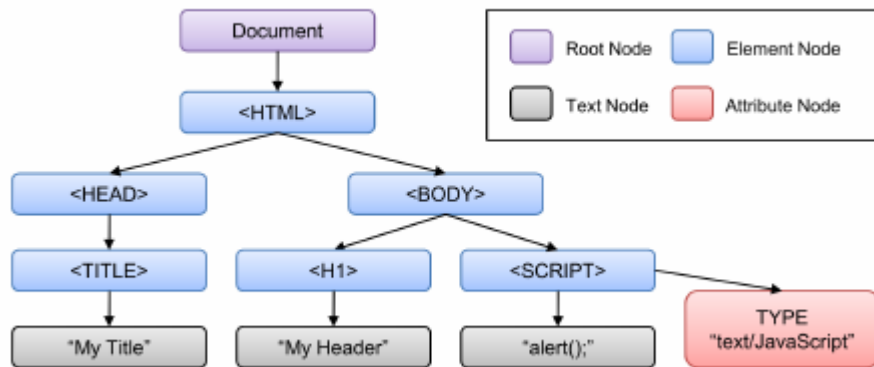
Manipulando el DOM

Para **manipular el DOM** vamos a hacerlo a través de objetos, llamando al objeto **document** (algo habíamos visto con *document.write*).

En el interior del DOM vamos a tener **elementos** y **nodos**. El **elemento** es la representación de una etiqueta HTML y el **nodo** es una unidad más básica todavía que el elemento propiamente dicho.

Hay formas de manipular el DOM a través de la *API nativa de JS*, que es lo que nos permite comunicarnos con el documento. Esta API nos proporciona un conjunto de métodos que van a permitirme modificar o acceder a datos del DOM.

Hay métodos más **antiguos**, que ya vienen con versiones anteriores de JavaScript y tenemos algunos más **modernos** que nos van a permitir hacer esto de forma mucho más sencilla.



Manipulando el DOM: 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, principalmente serán **Element** o **Node**:

- **Element** no es más que la representación genérica de una etiqueta: HTML.
- **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. Algunos ejemplos:

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

Modificar elementos

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 identificamos al elemento a través de alguno de sus atributos más utilizados, generalmente el **id** o la **clase**.

Métodos tradicionales

Uno de los métodos tradicionales para realizar búsquedas de elementos en el documento es **getElementById()**, que busca un documento específico. También tenemos **otros tres métodos** que nos devolverán un array donde tendremos que elegir el elemento en cuestión posteriormente:

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

Métodos tradicionales

Los métodos que comienzan con la palabra clave **get** nos van a devolver un valor. El **set** va a modificar un valor.

Los **gets** (getters) van a obtener un valor en *función de un criterio de búsqueda*. Si yo a un elemento le digo “*tráeme un elemento según el ID*” le tengo que pasar el ID. El ID era la propiedad que podía tener cada elemento del documento HTML. Puedo localizarlo a través de ese ID y si no lo encuentra devuelve **null**.

Lo que traigo por ID es un **objeto** que lo puedo almacenar en una variable.

Con **innerHTML** puedo pasarle un HTML para que reemplace su contenido por lo que le estoy pasando. Podemos utilizar comillas invertidas ` (se coloca con ALT+96) que permiten que los saltos de línea sean considerados.

También lo puedo buscar por su **nombre de clase** si un elemento HTML pertenece una clase de CSS. Pero el **getElementsByClassName** me va a devolver un array de elementos (*array de objetos*) porque podemos tener más de un elemento con la misma clase aplicada, con el ID solo se podía aplicar a un único elemento. Entonces podré acceder a una posición de ese array de elementos que es un único elemento.

Métodos tradicionales

```
<script>
function myFunction() {
  var x = document.getElementsByClassName("example");
  x[0].innerHTML = "Hello World!";
}
</script>
```

JS

En este caso modifica el elemento 0 de la colección de elementos cuya clase sea "example".

Ver ejemplo:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_getelementsbyclassname

En este ejemplo se obtienen los elementos de la clase="example" y se guardan en un array al cual luego se accede por su posición (0) y con innerHTML se cambia en el documento HTML.

getElementById()

El primer método, **.getElementById(id)** busca un elemento HTML con el **id especificado** en id por parámetro. En principio, un documento HTML bien construido 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> JS
```

*En el caso de no encontrar el elemento indicado, devolverá **null**.*

Ver ejemplo [aquí](#)

getElementsByClassName()

El método **.getElementsByClassName(class)** permite buscar los elementos con la **clase** especificada en **class**. Es importante darse cuenta de que el método 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
```

JS

*Estos métodos devuelven siempre un **array** con todos los elementos encontrados que encajen con el criterio. En el caso de no encontrar ninguno, devolverán un **array** 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");
```

JS

Ver ejemplos [getElementsByName\(name\)](#) y [getElementsByTagName\(tag\)](#)

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
ELEMENT <code>.querySelector(sel)</code>	Busca el primer elemento que coincide con el selector CSS <code>sel</code> . Si no, <code>NULL</code> .
ARRAY <code>.querySelectorAll(sel)</code>	Busca todos los elementos que coinciden con el selector CSS <code>sel</code> . Si no, <code>[]</code> .

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 **null**:

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

JS

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 **selector**, y en el primero le pasamos un simple string. 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 con este **All()** devolverá un **array** con todos los elementos que coinciden con el **selector** 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");
```

JS

En este caso **.querySelectorAll()** siempre nos devolverá un **array** de elementos. Depende de los elementos que encuentre mediante el **selector**, nos devolverá un **array** 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.*

Métodos modernos

Vamos a buscar elementos a través de los nombres de los selectores. Si tengo un selector por ID buscaré con **querySelector** la primera aparición del elemento que coincida con ese selector, si hay más devolverá el primero y si no puedo obtener un array de elementos, siempre y cuando coincidan con ese selector.

Si tengo una clase que aplica a más de un elemento HTML el array me va a devolver todas las clases. En cambio, si utilizo un selector de tipo ID solo devolverá ese único elemento o la primera aparición.

Ver ejemplo: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_queryselector_class

*En este ejemplo lo que hacemos es localizar la clase **.example** y modificar el color de fondo a rojo, al **primer elemento** que va a aparecer (en este caso el elemento h2). Lo busco, lo traigo y le hago la modificación de la nueva propiedad del backgroundColor.*

Con **querySelectorAll** puedo encontrar todas las apariciones de la clase .example:

Ver ejemplo:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_queryselectorall_class

Para hacer esto aplica un for, itera sobre los elementos que le devuelve el método y a todos les modifica el estilo.

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
Element .createElement(tag, options)	Crea y devuelve el elemento HTML definido por el String tag. Ej: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_createelement2
Nodo .createComment(text)	Crea y devuelve un nodo de comentarios HTML <!-- text -->.
Node .createTextNode(text)	Crea y devuelve un nodo HTML con el texto text. Ej: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_createTextNode2
Node .cloneNode(deep)	Clona el nodo HTML y devuelve una copia. deep es false por defecto. Ej: https://www.w3schools.com/jsref/met_node_clonenode.asp
Boolean .isConnected	Indica si el nodo HTML está insertado en el documento HTML. Ej: https://developer.mozilla.org/en-US/docs/Web/API/Node/isConnected

Create element

Mediante el método **.createElement()** podemos crear un elemento HTML en memoria. 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. En este ejemplo se crea un elemento de tipo botón:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_createelement2

Explicación del ejemplo:

Le pedimos que cree (*document.createElement*) un elemento de qué tipo de tipo **botón**, eso me devuelve un objeto, un elemento que es asignado a la variable **btn** y esa variable va a contener la información del elemento. Entonces al tener la variable hago una referencia al objeto con `var btn`.

Con **innerHTML** le voy a modificar el texto, en este caso el texto que aparece en el botón, mientras que con **appendChild** agrego ese elemento al final del body.

Entonces al objeto lo creamos, pero luego tenemos que asociarlo al body, al cuerpo de mi documento HTML.

innerHTML establece o devuelve el contenido de un elemento. appendChild agrega un nuevo nodo hijo, a un elemento, como último nodo hijo.

Create element

También podremos crear **nodos de texto**:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_document_createtextnode2

Explicación del ejemplo:

1. Creo el elemento H1 (etiqueta para un título)
2. Creo el texto del nodo H1
3. Le agrego con *appendChild* el texto que ya contiene la variable t, que es el texto del nodo.
4. Finalmente agrego al body ese nodo H1 (variable h, que ya contiene el texto).

Tip: Cambiar en la última línea del ejemplo la variable **h** por **t**: ¿por qué me lo crea al lado del botón y el texto es más pequeño?

Create element

Creación de listas clonando un elemento (*cloneNode*):

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_node_clonenode

Explicación del ejemplo:

1. Dentro del documento HTML creamos dos listas, cada una con un ID asociado.
2. Al botón le asociamos la función *myFunction* que obtiene el elemento según el ID. Esa función tiene dos variables. La variable ***itm*** obtiene el último elemento (*lastChild*) de la lista "myList2" y la variable ***cln*** lo que hace es una copia de ese elemento itm obtenido.
3. Con *appendChild* agregamos ese elemento "clonado" de la lista 2 a la lista 1.

Modificar atributos 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 que 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>
```

JS

En este ejemplo tengo un `<div>` creado con un **`createElement`**, lo que creo son los tags, las etiquetas **`div`**, que quedan almacenado en esa constante.

Luego a la constante le digo que modifique el atributo **ID**. Al `div` ahora lo voy a ver con el nuevo id que es "page", por lo cual estoy modificando una propiedad del elemento HTML pero desde JavaScript, es decir que ya no lo tengo que hacer manualmente en HTML como lo veníamos haciendo, sino que es JavaScript el que se encarga de modificar el atributo del elemento HTML.

Lo mismo sucede con el atributo **class**, donde le agrego el nombre de la clase y lo mismo si quiero agregarle estilo desde HTML.

Entonces no solo vamos a modificar el texto, sino que podemos modificar también los atributos de un elemento HTML.

Reemplazar contenido



Podemos reemplazar el contenido de una etiqueta HTML. Las propiedades son las siguientes:

Propiedades	Descripción
.textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar. Ej: https://www.w3schools.com/jsref/prop_node_textcontent.asp
.innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar. Ej: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_element_innerhtml

La propiedad **textContent** 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"
```

JS

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"
```

JS

Reemplazar contenido

Se logra con innerHTML:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_node_textcontent2

Explicación del ejemplo:

A esta función la estoy llamando con un evento onclick y lo que hace es obtener el contenido del texto de la lista (*textContent*) y con el **innerHTML** copio el texto de esa lista, guardado en la variable **x** y lo inserto dentro del párrafo que tiene el id "demo".

Otro ejemplo:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_element_innerhtml

Explicación del ejemplo:

En este caso al evento **onclick** se lo damos a un párrafo. Es decir que cuando hagamos clic en él se va a modificar el texto a través del id "demo", se dispara ese evento.

Este ejemplo se puede ver en los posts de Facebook cuando uno toca en "ver más" y se despliega el resto del texto.

Insertar una imagen (ejemplo)

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

En este ejemplo creo un elemento de tipo imagen (*createElement*) y lo guardo en la constante *img*. Luego le cambio el source (*src*) y el texto alternativo (*alt*) obligatorios en una etiqueta de imagen y lo agrego al body (*appendChild*). Append → Adjuntar.

https://www.w3schools.com/jsref/met_node_appendchild.asp






Eliminar el contenido

Con el método *remove()* quitamos un elemento del documento:

https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_node_remove

API nativa de Javascript

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
 <u>Buscar etiquetas</u>	Familia de métodos entre los que se encuentran funciones como <code>.getElementById()</code> , <code>.querySelector()</code> o <code>.querySelectorAll()</code> , entre otras.
 <u>Crear etiquetas</u>	Una serie de métodos y consejos para crear elementos en la página y trabajar con ellos de forma dinámica.
 <u>Insertar etiquetas</u>	Las mejores formas de añadir elementos al DOM, ya sea utilizando propiedades como <code>.innerHTML</code> o método como <code>.appendChild()</code> , <code>.insertAdjacentHTML()</code> , entre otros.
 <u>Gestión de clases CSS</u>	Consejos para la utilización de la API <code>.classList</code> 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.
 <u>Navegar entre elementos</u>	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.

Manipulación del DOM

Más info:

- https://www.w3schools.com/jsref/dom_obj_all.asp
- https://www.w3schools.com/js/js_htmldom_methods.asp
- <https://javadesde0.com/introduccion-a-document-object-model-dom/>

Eventos en JS

Los eventos son acciones que realiza el usuario (consciente o inconscientemente), y que como desarrolladores, tenemos que prever y preparar en el código para saber manejarlos y decirle a nuestra página o aplicación como debe actuar cuando ocurra uno. Estos eventos nos permitirán interactuar con el usuario, por ejemplo cuando hace clic en un botón.

Existen tres formas de definir eventos en nuestro código:

Estrategia	Ejemplo
A través de un atributo HTML donde asocias la función.	<code><tag onclick="..."></code>
A través de una propiedad JS donde asocias la función.	<code>tag.onclick = ...</code>
A través del método <code>addEventListener</code> donde añades la función.	<code>tag.addEventListener("click", ...)</code>

Desde atributos HTML

Probablemente, la forma más sencilla. En ella, definimos un evento a través de un atributo HTML. Los atributos, cuando son eventos, siempre comienzan por **onClick**, y en el valor se indica la función que se quiere ejecutar cuando se dispare dicho evento:

```
<button onClick="sendMessage()"> Press me!</button>  
<script>  
  const sendMessage = () => alert("Hello!");  
</script>
```

HTML

Desde propiedades JS

Otra forma interesante que podemos contemplar, es haciendo uso de las propiedades de Javascript. Por cada evento, existe una propiedad disponible en el elemento en cuestión:

```
<button>Press me!</button>  
<script>  
  const button = document.querySelector("button");  
  button.onclick = () => alert("Hello!");  
</script>
```

HTML

Ejemplo:

La sintaxis básica de un evento permite modificar un atributo (ejemplo: *onclick*) e invocar o llamar a una función:

https://www.w3schools.com/js/tryit.asp?filename=tryjs_event_onclick3

Explicación del ejemplo:

En este caso llama a la función ***changeText*** y al texto original lo va a modificar por el texto **“Ooops!”**. Pero ¿cómo sabe que tiene que modificar este H1 y no otro? porque le estoy pasando el propio objeto H1. No tengo que buscar el H1 por ID, sino que con el ***this*** cuando yo reciba el id va a ser ***el id de este elemento HTML***, le pasa la referencia propia a ese elemento HTML (el H1).

Entonces el *this* hace referencia al propio objeto, en este caso al propio elemento HTML. Con *changetext (this)* lo que hago es modificar ***ese mismo*** elemento HTML.

Con esta función podría modificar otro objeto porque es una propiedad compartida entre objetos, podría agregar un objeto H2 y sucedería lo mismo.

Para ampliar: https://www.w3schools.com/js/js_htmldom_events.asp