Desarrollo Web en Entorno Cliente

UT2. DOM, Eventos y Formularios

Actualizado Octubre 2022

ÍNDICE DE CONTENIDO

1. DOM 4
   1.1. Document 4
   1.2. Seleccionar elementos del DOM 6
      1.2.1. getElementById() 7
      1.2.2. querySelector(selectorCSS) 7
      1.2.3. querySelectorAll(selectorCSS) 8
   1.3. Crear elementos del DOM 8
      1.3.1. Crear elementos HTML 9
      1.3.2. createElement() 9
   1.4. Atributos HTML de un elemento 11
      1.4.1. API classList 13
   1.5. Insertar elementos en el DOM 16
      1.5.1. Reemplazar contenido 16
      1.5.2. Insertar elementos 18
      1.5.3. Eliminar elementos 20
   1.6. Navegar por elementos del DOM 21
2. EVENTOS 23
   2.1 Eventos mediante HTML 23
   2.2 Eventos mediante propiedades 24
   2.3 Eventos mediante listeners 24
   2.4 Objeto event 26
   2.5 Opciones de addEventListener 27
   2.6 Uso del objeto “this” en la gestión de eventos 28
   2.7 Drag and drop (arrastrar y soltar) 28
   2.8 Principales eventos 33
3. BOM 33
4. FORMULARIOS 35
   4.1 Input, text, textarea 36
   4.2 Radio button (botones de radio) 37
   4.3 Checkbox (casillas de verificación) 37
   4.4 Select y datalist 37
   4.5 Validar un formulario 38
      4.5.1. Validación al introducir información 39
      4.5.2. La interfaz FormData 41
      4.5.3. Deshabilitar enviar un formulario dos veces 42
      4.5.4. Enviar un formulario desde código 42

4.6. Bibliotecas para validación formularios Javascript 42
5. BIBLIOGRAFÍA 42
6. ANEXOS 44
6.1. Código HTML apartado 1.2 44
6.2. Librerías para trabajar con DOM 47
6.3. Evitar ataques inyección de código 48

1. DOM

El llamado DOM (Document Object Model) es un modelo (API) que permite tratar un documento Web XHTML como si fuera XML, navegando por los nodos existentes que forman la página, pudiendo manipular sus atributos e incluso crear nuevos elementos.

Para más información general de DOM:

- https://developer.mozilla.org/es/docs/Web/API/Document_Object_Model/Introduction
- https://es.wikipedia.org/wiki/Document_Object_Model

Usando Javascript para navegar en el DOM podemos acceder a todos los elementos XHTML de una página. Esto nos permite cambiar dinámicamente el aspecto de nuestras páginas Web.

1.1. 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.

```
console.log("********** Elementos del Documento **********");
console.log(window.document);
console.log(document);//es lo mismo que el anterior, no hace falta poner window.
delante de document, está implícito
console.log(document.head);
console.log(document.body);
console.log(document.documentElement);
console.log(document.doctype);
console.log(document.charset);
console.log(document.title);
console.log(document.links);
console.log(document.images);
console.log(document.forms);
console.log(document.styleSheets);
console.log(document.scripts);//Puede que devuelva dos al usar la extensión de Open
Live Server
setInterval(() => {
  console.log(document.getSelection().toString());//Muestra lo que se ha
seleccionado después de 2 segundos
}, 2000);
```

```
********** Elementos del Documento **********                    index.html:12
▼#document                                                        index.html:13
    <!DOCTYPE html>
    <html lang="en">
    ▶<head>…</head>
    ▶<body>…</body>
    </html>
▼#document                                                        index.html:14
    <!DOCTYPE html>
    <html lang="en">
    ▶<head>…</head>
    ▶<body>…</body>
    </html>
                                                                  index.html:15
▼<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstra
    p.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU
    8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
    <title>Document</title>
  </head>
▼<body>                                                           index.html:16
  ▶<script>…</script>
    <!-- Code injected by live-server -->
  ▶<script type="text/javascript">…</script>
  </body>
  <html lang="en">                                                index.html:17
  ▶<head>…</head>
  ▶<body>…</body>
  </html>
  <!DOCTYPE html>                                                 index.html:18
UTF-8                                                             index.html:19
Document                                                          index.html:20
▶HTMLCollection(0)                                                index.html:21
▶HTMLCollection(0)                                                index.html:22
▶HTMLCollection(0)                                                index.html:23
▶StyleSheetList                                                   index.html:24
▶HTMLCollection(2)                                                index.html:25
```

Dentro de document pueden existir varios tipos de elementos, pero principalmente serán element o

node:

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

1.2. 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, aunque también se puede hacer por etiquetas (a, img, p, h2, …)

El siguiente ejemplo de código JS se realiza sobre el código HTML del anexo Código HTML apartado 1.2

```
console.log(document.getElementsByTagName("li"));
	console.log(document.getElementsByClassName("card"));
	console.log(document.getElementsByName("nombre"));
	//Estos tres primeros, ya no se suelen usar y en su defecto se utiliza
querySelector y querySelectorAll
	console.log(document.getElementById("menu"));//este es más rápido que
querySelector
	console.log(document.querySelector("#menu"));//obtine el primero que se
encuentra
	console.log(document.querySelector("a"));
	console.log(document.querySelectorAll("a"));//los obtiene todos
	console.log(document.querySelectorAll("a").length);
	document.querySelectorAll("a").forEach((el) => console.log(el));
	console.log(document.querySelector(".card"));
	console.log(document.querySelectorAll(".card"));
	console.log(document.querySelectorAll(".card")[2]);
	console.log(document.querySelector("#menu li"));
	console.log(document.querySelectorAll("#menu li"));
```

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 array donde tendremos que elegir el elemento en cuestión posteriormente:

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

1.2.1. 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>
```

El resto de métodos han sido sustituidos por los siguientes métodos más modernos que utilizan selectores CSS: .querySelector() y .querySelectorAll()

Método de búsqueda	Descripción
ELEMENT .querySelector(sel)	Busca el primer elemento que coincide con el selector CSS **sel**. Si no, NULL.
ARRAY .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. El único inconveniente es que, por ejemplo, en el caso de .getElemetById(id) este es más rápido.

1.2.2. querySelector(selectorCSS)

Devuelve el primer elemento del documento que coincida con el grupo especificado de selectores:

```
// Ejemplo 1
const textoHeading = document.querySelector('.header__texto h2');
const p =document.querySelector("#miParrafo");
const c= document.querySelector("h2");
console.log(textoHeading);
textoHeading.textContent = 'Nuevo Heading'; // También se puede utilizar .text

// Ejemplo 2
const page = document.querySelector("#page");              // <div id="page"></div>
const info = document.querySelector(".main .info");        // <div class=”main”>
<div class="info"></div></div>
```

1.2.3. querySelectorAll(selectorCSS)

El método devuelve un array que representa una lista de elementos del documento que coinciden con el grupo de selectores indicados.

```
// Ejemplo 1
const enlaces = document.querySelectorAll('.navegacion a');
console.log(enlaces);
console.log(enlaces[0]);

// Algunas operaciones...

// Cambiar el texto
enlaces[0].textContent = 'Nuevo Texto enlace';

// Cambiar el enlace del primer enlace
enlaces[0].href = 'google.com';

// Agregar una clase...
enlaces[0].classList.add('nueva-clase');

// Eliminar una clase...
// enlaces[0].classList.remove('navegacion__enlace');

//Ejemplo 2
// 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");
```

Recuerda que 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.

Al asignar un elemento o nodo del DOM a una variable, se suele hacer como **const** (son objetos, así que recuerda que sus propiedades si se pueden modificar). Además, en algunos entornos, al nombre de estas variables se le antepone un **$** para distinguirlas de variables de la lógica de nuestra aplicación.

1.3. Crear elementos del DOM

Aunque no lo creas, 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.

Vamos a estudiar cómo 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.

1.3.1. Crear elementos HTML

Existen una serie de métodos para crear 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.
NODE .createComment(text)	Crea y devuelve un nodo de comentarios HTML <!-- text -->.
NODE .createTextNode(text)	Crea y devuelve un nodo HTML con el texto text.
NODE .cloneNode(deep)	Clona el nodo HTML y devuelve una copia. deep es false por defecto.
BOOLEAN .isConnected	Indica si el nodo HTML está insertado en el documento HTML.

1.3.2. createElement()

Mediante el método .createElement() podemos crear un element HTML en memoria (¡pero 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 (posteriormente veremos como insertarlo en el DOM). El funcionamiento de .createElement() es muy sencillo: se trata de pasarle el nombre de la etiqueta tag a utilizar.

```
// Ejemplo 1
// Generar HTML desde JavaScript...
const nuevoEnlace = document.createElement('A');
console.log(nuevoEnlace);
// Un enlace tiene una clase...
nuevoEnlace.classList.add('navegacion__enlace');
// Tiene un href
nuevoEnlace.href = 'nuevo-enlace.html';
// Tiene un Texto...
nuevoEnlace.textContent = 'Un Nuevo Enlace';
// console.log(nuevoEnlace);
label.setAttribute("for",valorAleatorio);

// Ejemplo 2Generar HTML desde JavaScript...
```

```
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()

```
const comment = document.createComment("Comentario"); // <!--Comentario-->
```

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.

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 boolean 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).

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.

1.4. Atributos HTML de un elemento

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:

```
//Ejemplo 1
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>
```

Los atributos disponibles a modificar dependerán de cada elemento.

Sin embargo, en algunos casos esto se puede complicar. 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 string separadas por espacio. De esta forma se asignarán múltiples clases. Aun así, recomendamos utilizar la propiedad classList que explicaremos más adelante.

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

```
// 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");        // class="info data dark" y data-number="5"
div.setAttribute("id", "page");   // Vuelve a añadir id="page"
```

Acceder al valor de un atributo mediante su propiedad o mediante el método .getAttribute() **no siempre devuelve el mismo resultado**:

```
<a class="link-dom" data-id="1" data-description="Document Object Model"
href="dom.html">DOM</a>
```

```
console.log(document.querySelector(".link-
dom").href);//http://127.0.0.1:5500/dom.html
console.log(document.querySelector(".link-dom").getAttribute("href"));//dom.html
```

También es posible interactuar con los data-attributes[1]. Recuerda que los atributos data-* permiten almacenar información adicional sobre un elemento HTML cualquiera, sin tener que recurrir a artilugios tales como la utilización de atributos no estándar.

```
<a class="link-dom" data-id="1" data-description="Document Object Model"
href="dom.html">DOM</a>

console.log($linkDOM.getAttribute("data-description"));
console.log($linkDOM.dataset);
console.log($linkDOM.dataset.description);
$linkDOM.setAttribute("data-description", "Modelo de Objeto del Documento");
console.log($linkDOM.dataset.description);
$linkDOM.dataset.description = "Suscríbete a mi canal y comparte";
console.log($linkDOM.dataset.description);
console.log($linkDOM.hasAttribute("data-id"));
$linkDOM.removeAttribute("data-id");
console.log($linkDOM.hasAttribute("data-id"));
```

Puedes acceder a todos **los estilos CSS** de un elemento o nodo con la propiedad **.style**. Además, para acceder a cualquier propiedad CSS a través de style, se utiliza la notación **lowerCamelCase** (background-color pasa a llamarse backgroundColor).

```
<a class="link-dom" data-id="1" data-description="Document Object Model"
href="dom.html">DOM</a>

const $linkDOM = document.querySelector(".link-dom");
console.log($linkDOM.style);//Muestra valores asignados por el usuarios
console.log($linkDOM.getAttribute("style"));
console.log($linkDOM.style.backgroundColor);
console.log($linkDOM.style.color);
```

[1] https://developer.mozilla.org/es/docs/Learn/HTML/Howto/Use_data_attributes

```
console.log(window.getComputedStyle($linkDOM));//Muestra valores por defecto que
asigna el navegador
console.log(getComputedStyle($linkDOM).getPropertyValue("color"));//Se puede
prescindir de window, está implícito
$linkDOM.style.setProperty("text-decoration", "none");
$linkDOM.style.setProperty("display", "block");
$linkDOM.style.width = "50%";
$linkDOM.style.textAlign = "center";
$linkDOM.style.marginLeft = "auto";
$linkDOM.style.marginRight = "auto";
$linkDOM.style.padding = "1rem";
$linkDOM.style.borderRadius = ".5rem";
console.log($linkDOM.style);
console.log($linkDOM.getAttribute("style"));
console.log(getComputedStyle($linkDOM));
//Variables CSS - Custom Properties CSS
const $html = document.documentElement,
  $body = document.body;
let varDarkColor = getComputedStyle($html).getPropertyValue("--dark-color"),
  varYellowColor = getComputedStyle($html).getPropertyValue("--yellow-color");
//Esas variables están en :root en un archivo CSS
//:root {
//      --yellow-color: #F7DF1E;
//      --dark-color: #222;
//    }

console.log(varDarkColor, varYellowColor);
$body.style.backgroundColor = varDarkColor;
$body.style.color = varYellowColor;
$html.style.setProperty("--dark-color", "#000");
varDarkColor = getComputedStyle($html).getPropertyValue("--dark-color");
$body.style.setProperty("background-color", varDarkColor);
```

1.4.1. API 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.

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", value)**. 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**. 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 array[2] (lista) de clases CSS de dicho elemento. Pero además, incorpora una serie de métodos ayudantes que nos harán muy sencillo trabajar con clases CSS:

[2] .classList no es un array, sino 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().

Método	Descripción
ARRAY .classList	Devuelve la lista de clases del elemento HTML.
STRING .classList.item(n)	Devuelve la clase número n del elemento HTML.
UNDEFINED .classList.add(c1, c2, ...)	Añade las clases c1, c2... al elemento HTML.
UNDEFINED .classList.remove(c1, c2, ...)	Elimina las clases c1, c2... del elemento HTML.
BOOLEAN .classList.contains(clase)	Indica si la clase existe en el elemento HTML.
BOOLEAN .classList.toggle(clase)	Si la clase no existe, la añade. Si no, la elimina.
BOOLEAN .classList.toggle(clase, expr)	Si expr es true, añade clase. Si no, la elimina.
BOOLEAN .classList.replace(old, new)	Reemplaza la clase old por la clase new.

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

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

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

```
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.

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"]
```

Ten en cuenta que en .toggle(), al contrario que .add() o .remove(), sólo se puede indicar una clase CSS por parámetro.

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

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

```
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)
```

1.5. Insertar elementos en el DOM

Al crear elementos, estos se crean en memoria y los almacenamos en una variable o constante. No se conectaban al DOM o documento HTML de forma automática, sino que debemos hacerlo manualmente.

1.5.1. 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
STRING .nodeName	Devuelve el nombre del nodo (etiqueta si es un elemento HTML). Sólo lectura.
STRING .textContent	Devuelve el contenido de texto del elemento. Se puede asignar para modificar.
STRING .innerHTML	Devuelve el contenido HTML del elemento. Se puede usar asignar para modificar.
STRING .outerHTML	Idem a .innerHTML pero incluyendo el HTML del propio elemento HTML.
STRING .innerText	Versión no estándar de .textContent de Internet Explorer con diferencias. **Evitar**.
STRING .outerText	Versión no estándar de .textContent/.outerHTML de Internet Explorer. **Evitar**.

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"
```

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.

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).

Si podéis usar tanto .innerHTML como .textContent, se aconseja este último, por eficiencia (https://developer.mozilla.org/es/docs/Web/API/Node/textContent):

1.5.2. 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.

Para añadir elementos al documento HTML (conectarlos al DOM) se pueden utilizar estos métodos:

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

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:

```
//Ejemplo 1
const img = document.createElement("img");
img.src = "https://lenguajejs.com/assets/logo.svg";
img.alt = "Logo Javascript";

document.body.appendChild(img);
//Ejemplo 2
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>
```

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 element

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

En las tres versiones, debemos indicar por parámetro un string 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 ejemplo, si la etiqueta HTML fuera * foo *, quedaría así:

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Por otro lado, notar que tenemos tres versiones en esta familia de métodos, una que actúa sobre elementos HTML (la que hemos visto), pero otras dos que actúan 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>
```

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:

- null, insertando newnode después del último nodo hijo. Equivalente a .appendChild().
- node, insertando newnode antes de dicho node de referencia.

1.5.3. 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.

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
```

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

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

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)[3]");    // <div
class="item">2</div>
document.body.removeChild(div); // Desconecta el segundo .item
```

[3] Representa el 2º elemento hermano

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);
```

1.6. 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.

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

Propiedades de elementos HTML	Descripción
ARRAY children	Devuelve una lista de elementos HTML hijos.
ELEMENT parentElement	Devuelve el padre del elemento o NULL si no tiene.
ELEMENT firstElementChild	Devuelve el primer elemento hijo.
ELEMENT lastElementChild	Devuelve el último elemento hijo.
ELEMENT previousElementSibling	Devuelve el elemento hermano anterior o NULL si no tiene.
ELEMENT nextElementSibling	Devuelve el elemento hermano siguiente o NULL si no tiene.

En primer lugar, tenemos la propiedad **children** que 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.

- 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 null.

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>
```

```
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.

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

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

```
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;               // p
a.nextSibling;                   // null
```

Cuidado con los espacios en el documento HTML al usar previusSibling y nextSibling, pues los detectan y devuelven #text.

2. EVENTOS

En Javascript existe un concepto llamado evento, que no es más que una notificación de que alguna característica interesante acaba de ocurrir. Dichas características pueden ser de múltiples tipos y muy variadas: desde un click de ratón hasta la pulsación de una tecla de teclado o la reproducción de un audio o video, entre muchas otras.

Nuestro objetivo es preparar el código para que cuando ocurra un evento con ciertas características, se ejecute un código asociado. Para ello, podemos gestionar los eventos de varias formas (veamos un ejemplo con el evento click):

Ejemplo	Descripción
`<button onClick="..."></button>`	Mediante un atributo HTML precedido de **on** que llama a la función.
`.onclick = ...`	Mediante propiedades precedidas de **on** que contienen una función.
`.addEventListener("click", ...)`	Mediante la escucha de eventos a través de listeners.

Cada una de estas opciones se puede utilizar para gestionar eventos en Javascript de forma equivalente, pero cada forma tiene sus ventajas y sus desventajas. Veamos detalladamente sus características.

2.1 Eventos mediante HTML

Probablemente, la forma más básica y sencilla es la de crear Eventos HTML a través de un atributo HTML en una etiqueta, como por ejemplo <button>. Se trata de una aproximación simple y sencilla, que para casos muy básicos puede ser más que suficiente:

```
<button onClick="alert('Hello!')">Saludar</button>
```

En este ejemplo, cuando el usuario haga click con el ratón en el botón Saludar, se dispara el evento

click sobre ese elemento HTML, y al tener un atributo onClick (cuando hagas click), se ejecutará el código que tenemos en el valor de dicho atributo, en este caso un alert(), que no es más que un mensaje emergente con el texto indicado. Sin embargo, crear eventos mediante HTML aunque es buen punto de partida inicial, tiene ciertas desventajas:

1. En primer lugar, estamos mezclando código de marcado HTML con código Javascript, cuando lo ideal sería tenerlo bien separado.
2. En segundo lugar, si el código del onClick es muy extenso, perjudica la legibilidad del mismo.
3. En tercer lugar, en el onClick podríamos hacer referencia al nombre de una función Javascript, solucionando el punto 2, pero seguiríamos con el mismo problema del punto 1. Además, a la larga, sería más complejo de mantener porque existe una dependencia del nombre de la función.

2.2 Eventos mediante propiedades

Esta opción se realiza desde Javascript. La idea es la misma que en la anterior, pero haciendo uso de una propiedad especial precedida por on a la que asignaremos la función con el código deseado.

```
const button = document.querySelector("button");
button.onclick = function() {
  alert("Hello!");
}
```

Observa que en este caso, lo que hacemos es asignar la función (con el código que queremos ejecutar cuando ocurra el evento) a la propiedad .onclick del elemento en cuestión (en nuestro caso, el botón que hemos seleccionado previamente). Ten en cuenta que los eventos como .onclick siempre van en minúsculas, ya que se trata de una propiedad Javascript y es sensible a mayúsculas y minúsculas.

2.3 Eventos mediante listeners

Quizás, la forma más interesante y versátil de gestionar los eventos de Javascript sea utilizando el método **.addEventListener()**. Este método pone en escucha al elemento, para que cuando se dispare el evento indicado en el string del primer parámetro, se ejecute la función indicada en el segundo parámetro:

Método	Descripción
.addEventListener(STRING event, FUNCTION func)	Añade una función al evento.
.addEventListener(STRING event, FUNCTION func, OBJECT options)	Idem, con opciones.
.removeEventListener(STRING event, FUNCTION func)	Elimina una función del evento.

Veamos un ejemplo sencillo del método .addEventListener() en acción:

```
const button = document.querySelector("button");
button.addEventListener("click", function() {
  alert("Hello!");
```

```
});
```

Es posible que veas mucho más organizado esto, si sacamos la función y la guardamos en una constante previamente, para luego hacer referencia a ella desde el .addEventListener():

```
const button = document.querySelector("button");
const action = function() {
  alert("Hello!");
};
button.addEventListener("click", action);
```

Dicho método .addEventListener() permite asociar múltiples funciones a un mismo evento, cosa que aunque no es imposible, es menos sencillo e intuitivo en las modalidades anteriores. Además, también es posible eliminar un listener previamente añadido, haciendo uso del método .removeEventListener() y teniendo la función externalizada en una constante:

```
const button = document.querySelector("button");
const funcName = function() {
  alert("Hello!");
};

button.addEventListener("click", funcName);       // Add listener
button.removeEventListener("click", funcName);    // Delete listener
```

Más ejemplos:

```
// Eventos en JavaScript...

// Hay muchos eventos ocurriendo en tus sitios y aplicaciones web, cuando un usuario
 da click, cuando dan scroll, al presionar en un botón, enviar un formulario, pero u
no de los más comunes es esperar a que el documento este listo...

console.log('1');
window.addEventListener('load', function() { // Cuando el Archivo JS y los archivos
dependientes han cargado como es el HTML, CSS, las imágenes...
    console.log('2');
});

window.onload = function() {
    console.log('3')
};

document.addEventListener('DOMContentLoaded', function() { // Este se ejecuta cuando
 el HTML ha sido descargado pero no espera por CSS o imagenes...
    console.log('4');
});

console.log('5');
```

```
// Estos closures también pueden ser con una función aparte...
function imprimir (){
    console.log("Imprimir");
}

window.addEventListener('load',imprimir);//al llamar a la función no se pone ()

// Eventos con Click...

const btnEnviar = document.querySelector('.formulario input[type=submit]');
console.log(btnEnviar);

btnEnviar.addEventListener('click', function() { // callback o closure
     console.log('click');
 });
btnEnviar.addEventListener('click', function(evento) { // evento incluye mucha infor
mación
     console.log(evento);
 });
```

2.4 Objeto event

En algunos casos, nos puede interesar profundizar en detalles relacionados con el evento. Por ejemplo, si estamos escuchando un evento de click de ratón, nos puede interesar saber con qué botón del ratón se ha hecho click, o en qué punto concreto de la pantalla. Estos detalles son posibles obtenerlos a través del objeto event.

En los ejemplos anteriores, la función (callback) que ejecuta la acción al dispararse el evento no tiene parámetros. Sin embargo, podemos indicarle un nombre a un primer parámetro, que será el que contendrá la información del evento:

```
const button = document.querySelector("button");
button.addEventListener("click", (event) => console.log(event));
```

Entre otras propiedades, el objeto event contiene lo siguiente:

```
// Objeto MouseEvent
{
  altKey: false,      // ¿La tecla ALT estaba presionada?
  clientX: 43,        // Posición en eje X donde se hizo click
  clientY: 16,        // Posición en eje Y donde se hizo click
  ctrlKey: false,     // ¿La tecla CTRL estaba presionada?
  detail: 1,          // Contador de veces que se ha hecho click
  ...                 // Otros...
}
```

- **type**: dice el tipo de evento que es (“click”, “mouseover”, etc...). Devuelve el nombre del evento tal cual, sin el “on”. Es útil para hacer una función que maneje varios eventos.

- **key**: en eventos de teclado, almacena el código de tecla de la tecla afectada por el evento. (keyCode está deprecated)
- **screenX / screenY**: en eventos del ratón, devuelve las coordenadas X e Y donde se encontraba el ratón, tomando como referencia la pantalla del ordenador.

En este caso, se trata de un objeto MouseEvent porque el evento que estamos escuchando es un evento de ratón. Sin embargo, si utilizáramos otro evento, probablemente tendríamos un objeto diferente.

```
function mostrarMensaje(evento) {
  if (evento.type === "keyup") {
    alert(evento.key);
  } else if (evento.type === "click") {
    alert(evento.clientX + " " + evento.clientY);
  }
}
document.getElementById("miObjeto").addEventListener("click", mostrarMensaje);
document.getElementById("miObjeto").addEventListener("keyup", mostrarMensaje);
document.getElementById("miObjeto").addEventListener("dblclick", function () {
  alert("Codigo metido directamente");
});
```

2.5 Opciones de addEventListener

Al utilizar el método .addEventListener(), su tercer parámetro es un object opcional, en el cual podemos indicar alguna de las siguientes opciones para modificar alguna característica de los listeners que escuchan un evento:

Opción	Descripción
BOOLEAN capture	El evento se dispara al inicio (*capture*), en lugar de al final (*bubble*).
BOOLEAN once	Sólo ejecuta la función la primera vez. Luego, elimina listener.
BOOLEAN passive	La función nunca llama a `.preventDefault()` (mejora rendimiento).

- **capture** nos permite modificar la modalidad en la que escuchará el evento (capture/bubbles, ver más adelante). Esto, básicamente, lo que hace es modificar en qué momento se procesa el evento.
- **once** nos permite indicar que el evento se procesará solo la primera vez. Internamente, lo que hace es ejecutarse la primera vez y llamar al .removeEventListener(), eliminando el listener una vez ha sido ejecutado por primera vez.
- **passive** nos permite crear un evento pasivo en el que indicamos que nunca llamaremos al método .preventDefault() para alterar el funcionamiento del evento. Esto puede ser muy interesante en temas de rendimiento (por ejemplo, al hacer scroll en una página), ya que los eventos pasivos son mucho menos costosos.

2.6 Uso del objeto "this" en la gestión de eventos

Cuando ejecutamos código dentro de un evento, existe la posibilidad de utilizar el objeto "this". Este objeto es una referencia al elemento DOM donde se ha producido el evento. Por ejemplo, si el evento se ha producido al hacer clic a una imagen con id="miImagen", el objeto "this" referencia a esa imagen, es decir, sería equivalente a referenciar el objeto usando document.getElementById("miImagen");

```
// // Eventos Scroll...
window.onscroll = function(e) {
     console.log('scrolling...');

     console.log(this.scrollY);
 }
```

2.7 Drag and drop (arrastrar y soltar)

Además de los eventos típicos tratados, existe un proceso (en el que entran en juego varios eventos) que es útil para el desarrollo de aplicaciones Web, el "Drag and Drop" (Arrastrar y soltar).

En W3Schools podéis obtener teoría y ejemplos de esta técnica https://www.w3schools.com/html/html5_draganddrop.asp

Es importante asignar la propiedad draggable a true para que el elemento se pueda arrastrar.

Ejemplo:

```
HTML
<div class="caja"></div>
<div class="contenedor" ></div>
CSS
.caja{
    width: 100px;
    height: 100px;
    background-color: blue;

}

.contenedor{
    width: 200px;
    height: 200px;
    border:1px solid black;
    border-radius: 20px;
    padding: 20px;
    margin-top: 20px;
}
JS
let caja = document.querySelector (".caja");
caja.draggable=true; //hacemos el elemento arrastrable
```

```
let contenedor = document.querySelector (".contenedor");

contenedor.addEventListener("dragenter", e=>{
  console.log("dragenter");//el elemento draggable empieza a entrar cuanto entra
el puntero del ratón
});
contenedor.addEventListener("dragleave", e=>{
  console.log("dragleave");//el elemento draggable empieza a salir cuanto sale el
puntero del ratón
});
contenedor.addEventListener("dragover", e=>{
  e.preventDefault();//para que ejecute el drop
  console.log("dragover");
});

contenedor.addEventListener("drop", e=>{
  console.log("drop");
  contenedor.appendChild(caja);
});

caja.addEventListener("dragstart",e=>{
  console.log("dragstart");
});
caja.addEventListener("dragend",e=>{
  console.log("dragend");
});
caja.addEventListener("drag",e=>{
//este evento se ejecuta constantemente
console.log("drag");
});
```

Ejemplo drag-drop selectivo:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Drag and Drop (2)</title>
    <style>
        body {
            font-family: Arial, Verdana, sans-serif;
        }

        #piezas {
```

```
        display: flex;
        flex-wrap: wrap;
        width: 400px;
        margin: auto;
        justify-content: center;
    }

    #puzzle {
        border: 1px solid black;
        width: 318px;
        height: 495px;
        display: flex;
        flex-wrap: wrap;
        margin: auto;
    }

    .pieza {
        width: 106px;
        height: 165px;
        background-size: cover;
        margin: 10px;
    }

    .placeholder {
        background-color: #F2F2F2;
        outline: 1px solid #333;
        width: 106px;
        height: 165px;
        transition: 1s;
    }

    .placeholder.hover {
        background-color: orange;
    }

    .placeholder .pieza {
        margin: 0;
    }

    #mensaje {
        color: black;
        text-align: center;
        display: none;
    }

    .ganaste {
        background-color: #B3D67C;
```

```
        }

        .ganaste #mensaje {
            display: block;
        }

        .ganaste .placeholder {
            outline: none;
        }

        .ganaste #piezas {
            display: none;
        }

        .container {
            display: flex;
            flex-wrap: wrap;
        }
    </style>
</head>
<body>
    <div class="container">
        <div id="puzzle"></div>
        <div id="piezas"></div>
    </div>
    <h1 id="mensaje">¡Ganaste!</h1>


    <script>
        const imagenes = [
            'imagen-0', 'imagen-1', 'imagen-2',
            'imagen-3', 'imagen-4', 'imagen-5',
            'imagen-6', 'imagen-7', 'imagen-8'
        ];

        const puzzle = document.getElementById('puzzle');
        const piezas = document.getElementById('piezas');
        const mensaje = document.getElementById('mensaje');

        let terminado = imagenes.length;

        while (imagenes.length) {
            const index = Math.floor(Math.random() * imagenes.length);
            const div = document.createElement('div');
            div.className = 'pieza';
            div.id = imagenes[index];
            div.draggable = true;
```

```
            div.style.backgroundImage = `url("recursos/${imagenes[index]}.jpg")`;
            piezas.appendChild(div);
            imagenes.splice(index, 1);
        }

        for (let i = 0; i < terminado; i++) {
            const div = document.createElement('div');
            div.className = 'placeholder';
            div.dataset.id = i;
            puzzle.appendChild(div);
        }


        piezas.addEventListener('dragstart', e => {
            e.dataTransfer.setData('id', e.target.id);
        });

        puzzle.addEventListener('dragover', e => {
            e.preventDefault();
            e.target.classList.add('hover');
        });

        puzzle.addEventListener('dragleave', e => {
            e.target.classList.remove('hover');
        });

        puzzle.addEventListener('drop', e => {
            e.target.classList.remove('hover');

            const id = e.dataTransfer.getData('id');
            const numero = id.split('-')[1];

            if (e.target.dataset.id === numero) {
                e.target.appendChild(document.getElementById(id));

                terminado--;

                if (terminado === 0) {
                    document.body.classList.add('ganaste');
                }
            }
        });

    </script>
</body>
</html>
```

Para dispositivos móviles podéis encontrar información en: https://medium.com/@deepakkadarivel/drag-and-drop-dnd-for-mobile-browsers-fc9bcd1ad3c5

2.8 Principales eventos

A continuación, mostramos un listado de los principales eventos existentes en Javascript:

- **onfocus**: al obtener un foco.
- **onblur**: al salir del foco de un elemento.
- **onchange**: al hacer un cambio en un elemento.
- **onclick**: al hacer un click en el elemento.
- **ondblclick**: al hacer doble click en un elemento.
- **onkeydown**: al pulsar una tecla (sin soltarla).
- **onkeyup**: al soltar una tecla pulsada.
- **onkeypress**: al pulsar una tecla.
- **onload**: al cargarse una página.
- **onunload**: al descargarse una página (salir de ella).
- **onmousedown**: al hacer clic de ratón (sin soltarlo).
- **onmouseup**: al soltar el botón del ratón previamente pulsado.
- **onmouseover**: al entrar encima de un elemento con el ratón.
- **onmouseout**: al salir de encima de un elemento con el ratón.
- **onsubmit**: al enviar los datos de un formulario.
- **onreset**: al resetear los datos de un formulario.
- **onselect**: al seleccionar un texto.
- **onresize**: al modificar el tamaño de la página del navegador.

3. BOM

El **BOM** (Browser Object Model) permite acceder y modificar propiedades de las ventanas del navegador (https://developer.mozilla.org/es/docs/Web/API/Window). Existen una serie de **objetos**, **propiedades** y **eventos** que nos permiten interactuar con el BOM.

A continuación, tienes algunos ejemplos con eventos y propiedades:

```
window.addEventListener("resize", (e) => {
  console.clear();
  console.log("********** Evento Resize **********");
  console.log(window.innerWidth);//parte interna visible
  console.log(window.innerHeight);//parte interna visible
  console.log(window.outerWidth);
  console.log(window.outerHeight);
  console.log(e);
});
window.addEventListener("scroll", (e) => {
  console.clear();
  console.log("********** Evento Scroll **********");
```

```
  console.log(window.scrollX);
  console.log(window.scrollY);
  console.log(e);
});
window.addEventListener("load", (e) => {
  console.log("********** Evento Load **********");
  console.log(window.screenX);
  console.log(window.screenY);
  console.log(e);
});
document.addEventListener("DOMContentLoaded", (e) => {
  console.log("********** Evento DOMContentLoaded **********");
  console.log(window.screenX);
  console.log(window.screenY);
  console.log(e);
});
```

El evento **DOMContentLoaded** es disparado cuando el documento HTML ha sido completamente cargado y parseado[4], sin esperar hojas de estilo, imágenes y subtramas para finalizar la carga. El evento **load** se dispara cuando se ha detectado la carga completa de la página.

Aquí tienes más ejemplos:

```
<h3 style="width: 2000px;">Manejo del BOM</h3>
  <button id="abrir-ventana">Abrir Ventana</button>
  <br><br>
  <button id="cerrar-ventana">Cerrar Ventana</button>
  <br><br>
  <button id="imprimir-ventana">Imprimir Ventana</button>


//window.alert("Alerta");
//window.confirm("Confirmación");
//window.prompt("Aviso");
const $btnAbrir = document.getElementById("abrir-ventana"),
  $btnCerrar = document.getElementById("cerrar-ventana"),
  $btnImprimir = document.getElementById("imprimir-ventana");
let ventana;
$btnAbrir.addEventListener(
  "click",
  (e) => (ventana = window.open("https://https://iesmartinezm.es/"))
);
$btnCerrar.addEventListener("click", (e) => {
  //window.close();
  ventana.close();
```

[4] Peticiones asíncronas pausan el parseo del DOM

```
});
$btnImprimir.addEventListener("click", (e) => window.print());
```

También es posible acceder a información del objeto **URL**, el **historial** o el **navegador**:

```
console.log("********** Objeto URL (location) **********");
console.log(location);
console.log(location.origin);
console.log(location.protocol);
console.log(location.host);
console.log(location.hostname);
console.log(location.port);
console.log(location.href);
console.log(location.hash);
console.log(location.search);
console.log(location.pathname);
//location.reload();
console.log("********** Objeto Historial (history) **********");
console.log(history);
console.log(history.length);
//history.forward(1);
//history.go(-3);
//history.back(2);
console.log("********** Objeto Navegador (navigator) **********");
console.log(navigator);
console.log(navigator.connection);
console.log(navigator.geolocation);
console.log(navigator.mediaDevices);
console.log(navigator.mimeTypes);
console.log(navigator.onLine);
console.log(navigator.serviceWorker);
console.log(navigator.storage);
console.log(navigator.usb);
console.log(navigator.userAgent);
```

4. FORMULARIOS

En este apartado trataremos algunas de las operaciones más habituales en los formularios. En primer lugar, veremos cómo acceder a elementos de formularios de distinto tipo. También veremos cómo validar los campos de un formulario antes de su envío.

Para simular el comportamiento del submit de un formulario puedes usar el servicio que nos ofrece https://formsubmit.co/, enviando correos.

Puedes revisar este enlace si no recuerdas muy bien cómo se hacía un formulario: https://lenguajehtml.com/html/formularios/crear-un-formulario/

Se puede acceder a cualquier elemento del formulario a través del propio formulario y el valor de la propiedad name del elemento.

```
<form id="responsive-tester">
        <input name="direccion" type="url" placeholder="URL" required>
        <br>
        <input name="ancho" type="text" placeholder="Ancho" required>
        <br>
        <input name="alto" type="text" placeholder="Alto" required>
        <br>
…
    const $form =d.getElementById(form);
    $form.direccion.value;
    $form.ancho.value;
    $form.alto.value
```

4.1 Input, text, textarea

Para acceder al valor de un input de tipo texto, simplemente debemos referenciar el atributo “value”.

```
//HTML
<input type="text" id="nombre">
//JS 1
const elemento=document.getElementById("nombre");
alert(elemento.value);
//JS 2
const nombre = document.querySelector("#nombre");
nombre.addEventListener('input', leerTexto());
function leerTexto (e){
    console.log(e.target.value);
}
//JS 3
const datos ={
    nombre:'',
    email:"",
    mensaje:""
};

const nombre = document.querySelector("#nombre");
nombre.addEventListener('input', leerTexto());
function leerTexto (e){
    datos[e.target.id]=e.target.value;
    console.log(datos);
}
```

4.2 Radio button (botones de radio)

Los “Radio button” son elementos del formulario que ante varias entradas te permiten seleccionar sólo una de ellas. Se agrupan teniendo un “name” común.

Para acceder a ellos, se accede como un array, donde se tiene el atributo “value” y el atributo “checked” que es true si está seleccionado, false en caso contrario.

```
//HTML
<input type="radio" id="preguntaSI" name="pregunta" value="si" />
<label for="preguntaSI">SI</label>
<input type="radio" id="preguntaNO" name="pregunta" value="no" />
<label for="preguntaNO">NO</label>
//JS
const elementos=document.querySelector('input[name="pregunta"]');
for(let i=0;i<elementos.length;i++){
  if(elementos[i].checked===true)
    alert("Valor del elemento marcado "+elementos[i].value);
}
```

4.3 Checkbox (casillas de verificación)

Similar a los “radio button”, salvo que permite que haya más de un elemento marcado.

```
//HTML
<input type="checkbox" id="preguntaAS" name="pregunta" value="asc" />
<label for="preguntaAS">Piso con ascensor</label>
<input type="checkbox" id="preguntaAM" name="pregunta" value="amb" />
<label for="preguntaAM">Piso amueblado</label>
//JS
const elementos=document.querySelector('input[name="pregunta"]');
for(let i=0;i<elementos.length;i++){
  if(elementos[i].checked===true){
    alert("Valor del elemento marcado "+elementos[i].value);
  }
}
```

4.4 Select y datalist

El select es un elemento que muestra un desplegable “cerrada” y nos permite elegir una opción del mismo. Aquí destaca el atributo “options”, que es un atributo que contiene un array con las opciones disponibles y el atributo “selectedIndex” que contiene (y se puede modificar) la posición del array “options” seleccionada actualmente (o la primera si se permite multiselección) o -1 si no está seleccionada ninguna opción (o queremos des-seleccionarlas).

Dentro de cada “options”, “value” almacena el valor y “text” el texto mostrado.

```
//HTML
<select id="aprobar" >
    <option value="10">Saco 10 en DWEC</option>
```

```
    <option value="9" selected>Saco 9 en DWEC</option>
    <option value="8">Saco 8 en DWEC</option>
  </select>
//JS
const elemento=document.querySelector("#aprobar");
//console.log(elemento);
Array.from(elemento.options).forEach(element => { //transformamos elemento.options en
array para poder usar foreach
  alert(`Valor de la opción ${element.text} es ${element.value}`);
});
let sel=elemento.selectedIndex;
alert("El valor de la opción seleccionada es "+elemento.options[sel].value+" y el
texto asociado es "+elemento.options[sel].text);
// Cambiamos el índice seleccionado
elemento.selectedIndex=2;
```

El datalist es un elemento que permite crear una lista “abierta”, donde el usuario puede seleccionar opciones sugeridas o indicar la suya propia escribiéndola manualmente. Funciona prácticamente igual que un <select>, conteniendo las opciones posibles en etiquetas <option> y utilizando un input para el elemento “seleccionado”.

```
//HTML
  <input type="text" list="items" name="ndata"/>
  <datalist id="items">
    <option value="1">Opción 1</option>
    <option value="2">Opción 2</option>
    <option value="3">Opción 3</option>
  </datalist>
//JS
const elemento=document.querySelector("#items");
const input = document.querySelector('input[name="ndata"]');
console.log(elemento);
Array.from(elemento.options).forEach(element => { //transformamos elemento.options
en array para poder usar foreach
  alert(`Valor de la opción ${element.text} es ${element.value}`);
});
console.log(`El valor introducido en el input del data lista es ${input.value}`);
```

4.5 Validar un formulario

Al crear un formulario en HTML, debemos ser conscientes de un detalle ineludible: los usuarios se equivocan al rellenar un formulario. Ya sea por equivocación del usuario, ambigüedad del formulario, o error del creador del formulario, el caso es que debemos estar preparados y anticiparnos a estos errores, para intentar que los datos lleguen correctamente a su destino y evitar cualquier tipo de moderación o revisión posterior.

Para evitar estos casos, se suele recurrir a un tipo de proceso automático llamado validación, en el cuál, establecemos unas pautas para que, si el usuario introduce alguna información incorrecta, deba modificarla o en caso contrario no podrá continuar ni enviar el formulario correctamente.

En un caso ideal, un formulario debe tener validación en el front-end, y si lo supera, vuelve a pasar otro proceso de validación en el back-end.

Hay dos tipos diferentes de validación por parte del cliente que encontrarás en la web:

- La validación de formularios incorporada utiliza características de validación de formularios HTML5 (https://lenguajehtml.com/html/formularios/validaciones-html5/ y https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation), Esta validación generalmente no requiere mucho JavaScript. La validación de formularios incorporada tiene un mejor rendimiento que JavaScript, pero no es tan personalizable como la validación con JavaScript. Además, es fácilmente modificable por cualquier usuario.
- La validación con JavaScript se codifica utilizando JavaScript. Esta validación es completamente personalizable, pero debes crearlo todo (o usar una biblioteca).

En este apartado vamos a centrarnos en el segundo caso.

La validación se puede realizar principalmente en dos momentos, mientras se van introduciendo los datos (por ejemplo con el evento “keyup”) o al enviar el formulario (evento “submit”).

Para mostrar los errores, se pueden usar las pseudoclases de CSS :valid e :invalid.

4.5.1. Validación al introducir información

Dependiendo del elemento del formulario, debemos usar un evento u otro:

Elemento	Evento
input	keyup/change
radio button/checkbox	click/change
select	change
datalist	change

Si un manejador de un evento devuelve true (o no devuelve nada), se realiza el evento asociado. Si el manejador devuelve false, se cancela el evento.

Existe un evento asociado a un formulario completo llamado “onsubmit”.

Aprovechando todo esto, podemos a nuestro antojo permitir el envío de información al servidor mediante el formulario devolviendo true o cancelarlo devolviendo false.

Para la validación de formularios en JS es imprescindible el uso del método **.preventDefault()** del evento, para “evitar” el comportamiento normal de un evento y para asignarle el que se quiera, en este caso, evitar el envío del formulario en el evento onsubmit y realizar una validación.

Una manera de trabajar es utilizar este método justo al principio del manejador del evento que dispare el submit del formulario:

```
if(nombre.value==""){
  e.preventDefault();
```

```
  document.getElementById("errorNombre").contenText= "Debes rellenar el campo";
  document.getElementById("errorNombre").setAttribute("style","color:red;");
}else{
  //podríamos escribir si quisiéramos document.formulario.submit() y forzaríamos el
envío pero es innecesario porqué se hará igualmente.
              console.log("el formulario se ha enviado");
}
```

```
const btnEnviar = document.querySelector(".formulario input[type=submit]");
const formulario = document.querySelector(".formulario")
console.log(btnEnviar);
formulario.addEventListener("submit", function (e) {
  e.preventDefault();
  //obtener valores
  let nombre = this.nombre.value;
  let email = this.email.value;
  let mensaje = this.mensaje.value;

  // Validar el Formulario...
  if(nombre === '' || email === '' || mensaje === '' ) {
    console.log("Al menos un campo esta vacio");
    mostrarError("Todos los campos son obligatorios");
    return; // Detiene la ejecución de esta función
  }
  console.log("Todo bien...");
  mostrarMensaje("Mensaje enviado correctamente");
});
function mostrarError(mensaje) {
  const alerta = document.createElement("p");
  alerta.textContent = mensaje;
  alerta.classList.add("error");
  formulario.appendChild(alerta);
  setTimeout(() => {
    alerta.remove();
  }, 3000);
}
function mostrarMensaje(mensaje) {
  const alerta = document.createElement("p");
  alerta.textContent = mensaje;
  alerta.classList.add("correcto");
  formulario.appendChild(alerta);
  setTimeout(() => {
    alerta.remove();
  }, 3000);
}
```

Sin embargo, en otros escenarios se utiliza **.preventDefault()** cuando se detecta el error y se va a mostrar el mismo, para que no envíe el formulario. P.e.:

```
form.addEventListener('submit', function (event) {
    // si el campo de correo electrónico es válido, dejamos que el formulario se
envíe

    if(!email.validity.valid) {
      // Si no es así, mostramos un mensaje de error apropiado
      showError();
      // Luego evitamos que se envíe el formulario cancelando el evento
      event.preventDefault();
    }
  });
```

4.5.2. La interfaz FormData

La interfaz FormData[5] (https://developer.mozilla.org/es/docs/Web/API/FormData) proporciona una manera sencilla de construir un conjunto de parejas clave/valor que representan los campos de un formulario y sus valores, que pueden ser enviados fácilmente. Están destinados principalmente para el envío de los datos del formulario, pero se pueden utilizar de forma independiente con el fin de transmitir los datos tecleados.

- FormData.get() Devuelve el primer valor asociado con una clave dada en un objeto FormData.
- FormData.entries() Devuelve un iterator que permite recorrer todas las parejas clave/valor contenidas en este objeto.
- FormData.values() Devuelve un iterator que permite recorrer todos los valores contenidos en este objeto.

```
const formulario = document.querySelector('#formulario')
…
const datos = new FormData(formulario)

    console.log('campo Email', datos.get('emailCampo'))
    console.log('campo Password', datos.get('passCampo'))
    console.log('campo Checkbox', datos.get('checkCampo'))
```

```
formulario.addEventListener("submit", (e) => {
  e.preventDefault();
  console.log("funciona");

  const inputs = new FormData(formulario);
  console.log(inputs.get("userName"));
  console.log(inputs.get("userEmail"));

  for (let campo of inputs.values()) {
```

[5] FormData puede dar problemas con Chrome

```
        console.log(campo);
    }

    for (let campo of inputs.entries()) {
        console.log(campo);
    }
});
```

4.5.3. Deshabilitar enviar un formulario dos veces

A veces un usuario pulsa enviar un formulario más de una vez por error.

Si queremos evitar esto, podemos usar “this.disabled=true”;

```
<form onsubmit="this.disabled=true;">
```

4.5.4. Enviar un formulario desde código

En algunas aplicaciones por motivos estéticos o de funcionalidad es deseable que el “enviar un formulario” no se haga desde un botón “submit”, sino desde cualquier otro evento que permita la ejecución de código. Esto se puede hacer recogiendo el elemento del formulario y aplicando el método submit().

```
<form id="formulario">
```

```
const elemento=document.getElementById("formulario");
elemento.submit();
```

4.6. Bibliotecas para validación formularios Javascript

Algunas de las bibliotecas más populares para la validación de formularios Javascript son:

- Joi: https://joi.dev/
- Validator.js https://github.com/validatorjs/validator.js
- Validate.js: http://rickharrison.github.io/validate.js/

5. BIBLIOGRAFÍA

[1] Javascript Mozilla Developer https://developer.mozilla.org/es/docs/Web/JavaScript

[2] Javascript ES6 W3C https://www.w3schools.com/js/js_es6.asp

[3] Referencia DOM
https://developer.mozilla.org/es/docs/Referencia_DOM_de_Gecko/Introducci%C3%B3n

[4] DOM: https://lenguajejs.com/javascript/dom/que-es/

[5] Referencia eventos Javascript http://www.w3schools.com/jsref/dom_obj_event.asp

[6] Valida formularios: https://www.delftstack.com/es/howto/javascript/javascript-form-submit/ y https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation#validar_formularios_utilizando_javascript

6. ANEXOS

6.1. Código HTML apartado 1.2

```
<style>
      :root {
        --yellow-color: #f7df1e;
        --dark-color: #222;
      }

      .cards {
        border: thin solid var(--dark-color);
        padding: 1rem;
      }

      .card {
        display: inline-block;
        background-color: var(--dark-color);
        color: var(--yellow-color);
      }

      .card figcaption {
        padding: 1rem;
      }

      .rotate-45 {
        transform: rotate(45deg);
      }

      .rotate-135 {
        transform: rotate(135deg);
      }

      .opacity-80 {
        opacity: 0.8;
      }

      .sepia {
        filter: sepia(1);
      }

      .eventos-flujo div {
        padding: 4rem;
        font-size: 2rem;
```

```
        text-align: center;
      }

      .uno {
        background-color: yellow;
      }

      .dos {
        background-color: gold;
      }

      .tres {
        background-color: lightyellow;
      }
    </style>

<body>
    <h1>WEB APIs</h1>
    <h2>DOM: Document Object Model</h2>
    <h2>BOM: Browser Object Model</h2>
    <h2>CSSOM: CSS Object Model</h2>
    <h2>WEB APIs</h2>
    <ul>
      <li>Eventos</li>
      <li>Forms</li>
      <li>AJAX - Fetch</li>
      <li>History</li>
      <li>Web Storage</li>
      <li>Geolocation</li>
      <li>Drag & Drop</li>
      <li>Indexed DB</li>
      <li>Canvas</li>
      <li>MatchMedia</li>
      <li>etc..</li>
    </ul>
    <br />
    <hr />
    <br />
    <h3>Manejo del DOM</h3>
    <p id="que-es">
      Lorem ipsum dolor sit amet consectetur adipisicing elit. Quidem,
      laboriosam cum quod, perferendis eveniet pariatur facere doloribus
      adipisci consequatur totam est dolorum vero nisi voluptate porro explicabo
      quo necessitatibus blanditiis.
    </p>
    <nav id="menu">
      <ul>
```

```
      <li><a href="#">Sección 1</a></li>
      <li><a href="#">Sección 2</a></li>
      <li><a href="#">Sección 3</a></li>
      <li><a href="#">Sección 4</a></li>
      <li><a href="#">Sección 5</a></li>
    </ul>
  </nav>
  <input type="text" name="nombre" placeholder="Nombre" />
  <a
    class="link-dom"
    data-id="1"
    data-description="Document Object Model"
    href="dom.html"
    style="background-color: #f7df1e; color: #222"
    >DOM</a
  >
  <section class="cards">
    <figure class="card">
      <img src="https://placeimg.com/200/200/tech" alt="Tech" />
      <figcaption>Tech</figcaption>
    </figure>
    <figure class="card">
      <img src="https://placeimg.com/200/200/animals" alt="Animals" />
      <figcaption>Animals</figcaption>
    </figure>
    <figure class="card">
      <img src="https://placeimg.com/200/200/arch" alt="Architecture" />
      <figcaption>Architecture</figcaption>
    </figure>
    <figure class="card">
      <img src="https://placeimg.com/200/200/people" alt="People" />
      <figcaption>People</figcaption>
    </figure>
    <figure class="card">
      <img src="https://placeimg.com/200/200/nature" alt="Nature" />
      <figcaption>Nature</figcaption>
    </figure>
  </section>
  <template id="template-card">
    <figure class="card">
      <img />
      <figcaption></figcaption>
    </figure>
  </template>
  <h3>Eventos en JavaScript</h3>
  <h4>Manejadores de Eventos</h4>
  <button onclick="holaMundo()">Evento con atributo HTML</button>
```

```
    <br /><br />
    <button id="evento-semantico">Evento con manejador semántico</button>
    <br /><br />
    <button id="evento-multiple">Evento con manejador múltiple</button>
    <br /><br />
    <button id="evento-remover">
      Removiendo eventos con manejadores múltiples
    </button>
    <h4>Flujo de Eventos</h4>
    <section class="eventos-flujo">
      <div class="uno">
        1
        <div class="dos">
          2
          <div class="tres">
            3
            <br />
            <a href="https://jonmircha.com/" target="_blank" rel="noopener"
              >jonmircha.com</a
            >
          </div>
        </div>
      </div>
    </section>
    <h3 style="width: 2000px">Manejo del BOM</h3>
    <button id="abrir-ventana">Abrir Ventana</button>
    <br /><br />
    <button id="cerrar-ventana">Cerrar Ventana</button>
    <br /><br />
    <button id="imprimir-ventana">Imprimir Ventana</button>
    <br /><br />
    <script src="js/dom.js"></script>
  </body>
```

6.2. Librerías para trabajar con DOM

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 mínimamente 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

6.3. Evitar ataques inyección de código

Dos de los ataques más peligrosos que atentan contra la seguridad de un sitio web son: la Inyección de código SQL y XSS (Cross-site scripting).

- Inyección SQL es la infiltración de código intruso dentro del código SQL de las bases de datos generalmente usando las entradas de los formularios. El objetivo es alterar el funcionamiento del programa y lograr que se ejecute el código "invasor" incrustado. Es una técnica poderosa con la que se puede manipular direcciones URL de cualquier forma, en entradas de búsquedas, formularios o registros de email, para inyectar el código.
- XSS es la inyección de código JavaScript en aplicaciones web, páginas web y hasta en el navegador web ya sea en código insertado (script, iframes) en HTML, aprovechando las variables que se pasan entre páginas usando una URL o robando las cookies. Ataca principalmente las aplicaciones que procesan datos obtenidos de una entrada sin ningún tipo de chequeo o validación.

Usar captchas en los formularios para validar las peticiones de los usuarios

Captcha es la abreviatura en ingles de "Prueba de Turing pública y automática para diferenciar máquinas y humanos".

Es una prueba para comprobar que quien introduce la información es una persona y no una máquina.

Se utilizan para impedir que se pueda tener acceso a la función de un script de forma automática.

Para eso se emplean pequeñas imágenes que representan caracteres, en ocasiones distorsionados, los que el usuario tiene que introducir en un cuadro manualmente y de coincidir, se ejecuta la función solicitada.

Estas imágenes se escriben mediante PHP, creando un pequeño archivo con solo unas líneas de código y vincularlo al formulario y viceversa.

Actualmente usar captchas en formularios es desaconsejado, porque crea repulsión en los usuarios y es poco práctico.

La tendencia actual es usar el método de reCAPTCHA de Google, en la que el lector solo debe marcar una casilla y Google se encarga de comprobar que se trata de un humano y no una máquina. Aquí puedes ver como integrar reCAPTCHA en una página.

Crear limitaciones y reglas en los formularios

En cualquier formulario es necesario usar ciertas reglas para limitar las acciones de acuerdo a su uso:

- Se debe utilizar como método en los formularios POST en vez de GET.
- En las entradas para introducir contraseñas, en ver de <input type="text"> se debe emplear: <input type="password">
- En las entradas de texto se puede limitar la cantidad de caracteres usando el atributo "maxlength".
- En los formularios usados para subir archivos como imágenes, fotos, etc. puede limitarse el tipo de archivo a subir, basado en su extensión, así como regular su tamaño.

Comprobaciones desde el lado Servidor

Generalmente las comprobaciones para evitar los dos ataques anteriormente mencionados se realizan desde el lado del servidor.

En el caso de PHP puedes revisar este enlace: https://norfipc.com/inf/como-proteger-formularios-web-evitar-inyeccion-codigo-sql.php

En el caso de NodeJS puede ver estos otros: https://ajaxhispano.com/ask/prevencion-de-la-inyeccion-sql-en-el-nodojs-36652/ y https://github.com/mysqljs/mysql#escaping-query-values