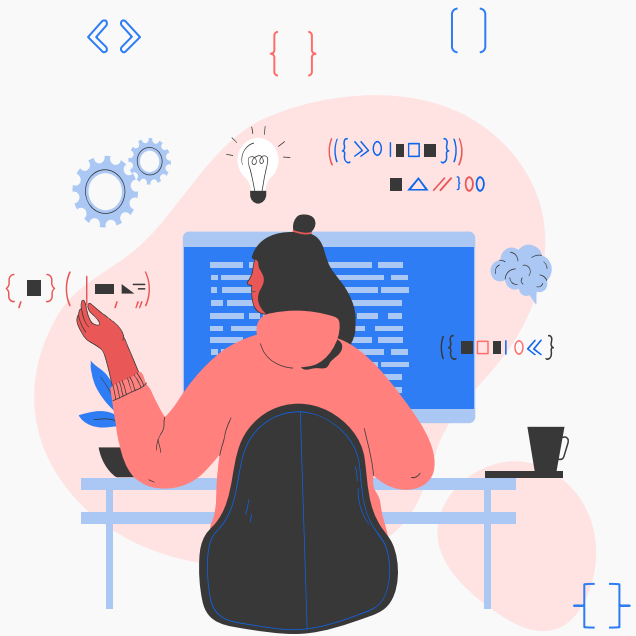
Desarrollo Web en Entorno Cliente

Tema 3 – Interacción con el usuario. DOM y eventos



Marina Hurtado Rosales
marina.hurtado@escuelaartegranada.com

Indice de contenidos

Navegador web. Proceso de renderizado

¿Qué es un navegador web?

Es un programa que permite **visualizar e interactuar con páginas web.**

Interpreta el código HTML, CSS y JavaScript de una página web.

Su función principal es mostrar el contenido de manera comprensible para el usuario.

Cada navegador tiene su **propio motor de renderizado:** Blink (Chrome, Edge), Gecko (Firefox), WebKit (Safari)...

Un navegador **no solo muestra el código HTML y CSS**, sino que lo **analiza, interpreta y transforma** en una estructura manipulable por JavaScript.

Critical Rendering Path

Critical rendering path se refiere al proceso que un navegador web sigue para renderizar una página web en la pantalla del usuario. Es el **proceso que sigue el navegador** para convertir el código fuente (HTML, CSS, JS) en **píxeles visibles en pantalla**.

Este proceso afecta directamente al **rendimiento y la velocidad de carga** de una página.

En este proceso intervienen **múltiples pasos consecutivos**, desde la descarga de recursos hasta la composición final de la imagen.

Critical Rendering Path

Los pasos que se dan para el renderizado son los siguientes:

- **Recuperación de recursos :** Solicita recursos, html, css, javascript, img, fuentes, etc..
- **Análisis del HTML:** Analiza el html para construir el DOM
- **Construcción del árbol de estilos (CSSOM)**
- **Combinación del DOM y el CSSOM**
- **Layout (Diseño):** El navegador calcula la geometría exacta de cada elemento determinando su posición y tamaño en relación con la ventana gráfica del navegador
- **Pintura (Painting):** El navegador pinta los píxeles en la pantalla.
- **Composición (Composition):** El navegador combina las capas de pintura en una sola imagen que se muestra en la pantalla.

JavaScript se ejecuta después de que se haya construido el DOM y el CSSOM, y su ubicación en la página y el uso de atributos específicos pueden influir en cuándo se carga y ejecuta, así como en cómo afecta al proceso de renderización de la página web..

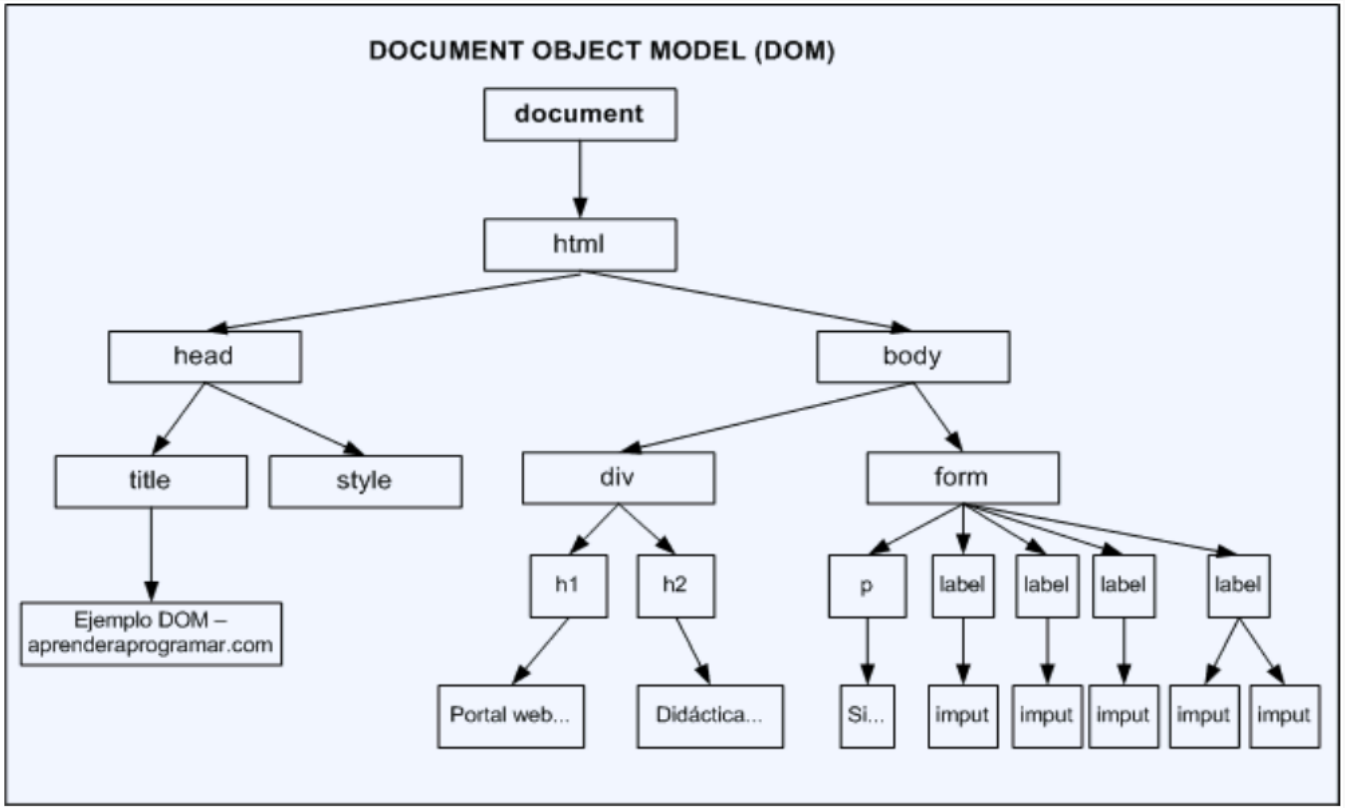
Introducción al DOM

¿Qué es el DOM?

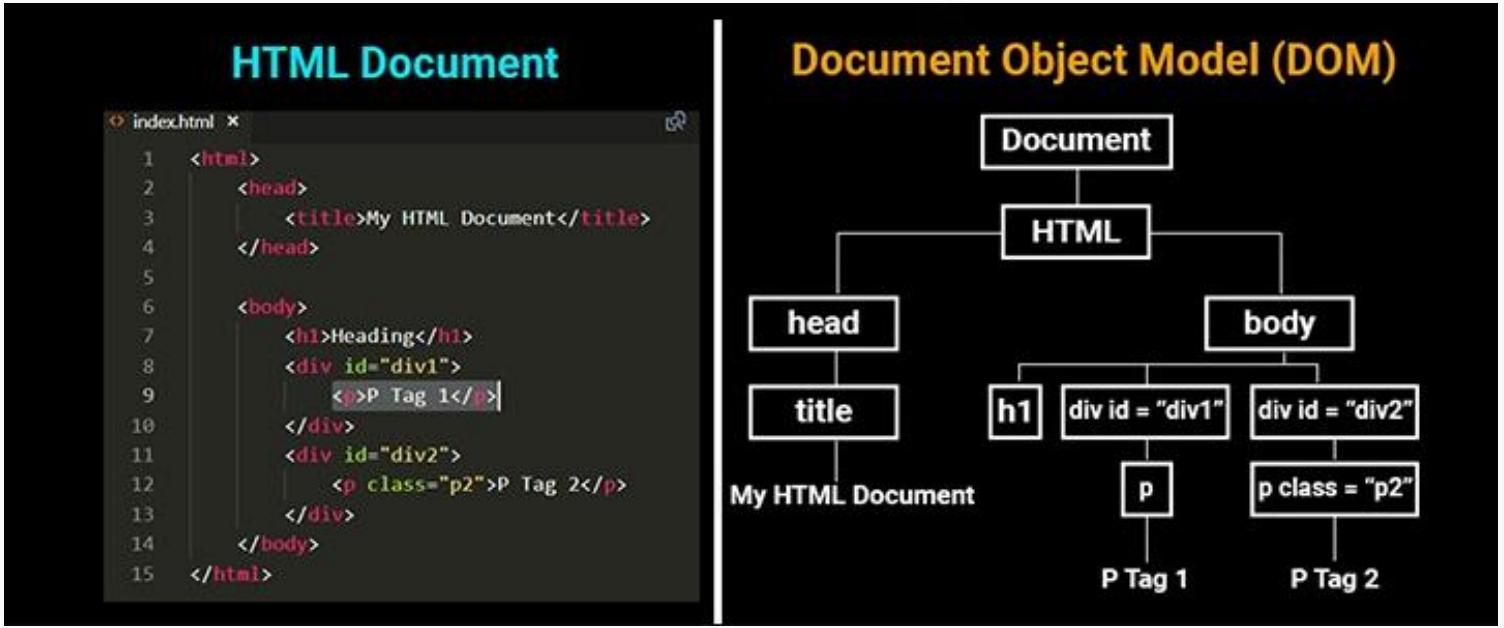
Las siglas **DOM** significan **Document Object Model**, o lo que es lo mismo, la estructura del documento HTML. Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol **DOM**.

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

¿Qué es el DOM?


DOCUMENT OBJECT MODEL (DOM)
document
html
head
body
title
style
div
form
Ejemplo DOM – aprenderaprogramar.com
h1
h2
p
label
label
label
label
Portal web...
Didáctica...
Si...
imput
imput
imput
imput
imput

¿Qué es el DOM?


HTML Document
index.html
1 <html>
2 <head>
3 <title>My HTML Document</title>
4 </head>
5
6 <body>
7 <h1>Heading</h1>
8
9 P Tag 1
10
11
12 P Tag 2
13
14 </body>
15 </html>
Document Object Model (DOM)
Document
HTML
head
body
title
h1
div id = "div1"
div id = "div2"
My HTML Document
p
p class = "p2"
P Tag 1
P Tag 2

¿Qué es el DOM?

El **DOM está formado por nodos**, que representan las diferentes partes de un documento HTML.

Cada etiqueta, texto, atributo o comentario es un nodo distinto.

Estos nodos se estructuran en forma de árbol, donde cada uno puede tener una relación de padre, hijo, hermano con otros nodos.

Existen diferentes tipos de nodos en el DOM, los más comunes son:

- **Document** -> representa el documento completo (document)
- **Element** -> representa una etiqueta HTML (<p>, …)
- **Text** -> representa el texto dentro de una etiqueta (“Hola mundo”)
- **Attribute** -> un atributo de una etiqueta (id=“titulo”)
- **Comment** -> un comentario HTML (<!– nota -->)

Un **nodo** es una etiqueta HTML sobre la que vamos a poder realizar operaciones de lectura y escritura.

Propiedades básicas de los nodos

Todos los nodos comparten ciertas propiedades:

- **nodeType** -> indica el tipo de nodo

Valor	Tipo de nodo
1	Elemento
2	Atributo
3	Texto
8	Comentario

Valor	Tipo de nodo
9	Documento
10	Tipo de documento
11	Fragmento de documento

- **nodeName** -> indica el nombre del nodo (por ejemplo “p”)
- **nodeValue** -> contenido del nodo (solo para nodos de texto o comentario

Estas propiedades permiten **identificar el tipo de nodo** que estamos leyendo o manipulando.

El objeto document

Es el **punto de entrada** al árbol DOM.

Representa **todo el documento HTML** cargado en la página.

Nos permite **acceder a los elementos** y consultar o modificar su contenido.

Las propiedades más usadas de este objeto son:

- **document.documentElement** -> devuelve el elemento <html>
- **document.head** -> devuelve el <head>
- **document.body** -> devuelve el <body>

Recorrer el árbol del DOM

Cada nodo conoce sus **relaciones con otros nodos**:

- **parentNode** -> devuelve el nodo **padre**
- **childNodes** -> devuelve una lista con **todos los nodos hijos**, incluyendo textos, saltos de línea y comentarios
- **children** -> devuelve una lista con los nodos **hijos** que son **elementos HTML**
- **firstChild** -> devuelve el primer hijo
- **lastChild** -> devuelve el último hijo
- **nextSibling** -> devuelve el siguiente nodo hermano
- **previousSibling** -> devuelve el anterior nodo hermano

Podemos **recorrer el árbol DOM** igual que una estructura de datos, navegando entre padres, hijos y hermanos.

Acceso y manipulación del DOM

Funciones del DOM

Utilizando esta representación y JavaScript, podemos modificar parte de la página web a través del código. Algunas de las funciones que podemos realizar accediendo al DOM son la siguientes:

- Cambiar todos los elementos HTML que se encuentran en la página.
- Cambiar todos los atributos de aquellos elementos que deseemos que estén en la página que se ha cargado.
- Modificar la representación visual, es decir, los estilos de los elementos que están en la página.
- Eliminar elementos HTML que se encuentre en la página.
- Añadir nuevos elementos a dicha página.
- Reaccionar a eventos existentes en la página.
- Crear nuevos eventos en la página.

Todos los cambios que realicemos sobre el documento utilizando el DOM se visualizarán o surtirán efecto en el momento, de modo que no es necesario volver a cargar la página.

Funciones del DOM: Acceso al DOM

Para modificar el contenido de la página, primero **debemos acceder a los elementos** del DOM.

JavaScript ofrece varios métodos para seleccionar elementos concretos del documento:

- **getElementById(“identificador”)**: devuelve el elemento HTML cuyo atributo id coincide con el parámetro indicado en la función.
- **getElementsByClassName(“clase”)**: devuelve todos los elementos de la página HTML que tienen esa clase que se pasa como parámetro a la función.
- **getElementsByTagName(“etiqueta”)**: obtiene todos los elementos de la página HTML cuya etiqueta sea igual que el parámetro que se le pasa a la función.
- **getElementsByName(“nombre)**: es similar al anterior, pero en este caso se buscan los elementos cuyo atributo name sea igual al parámetro proporcionado.

Funciones del DOM: Selectores CSS

Hoy en día se recomienda utilizar los métodos basados en **selectores CSS**, por su flexibilidad:

- **querySelector(“selector”)**: devuelve el **primer elemento** que cumple la condición. Si no existe el elemento, el valor retornado es null.
- **querySelectorAll(“selector”)**: devuelve un objeto con **todos los elementos** que coincidan con el selector

Diferencia entre HTMLCollection y NodeList

Los métodos de acceso pueden devolver dos tipos de colecciones diferentes:

- **HTMLCollection**: colección de elementos HTML. Se trata de una **colección viva**, la cual se actualiza si el DOM cambia. Se obtiene principalmente a través de propiedades como *document.forms, document.images, document.links* y métodos como *document.getElementsBy…*

- **NodeList**: colección de nodos (etiquetas HTML, nodos texto, comentarios, etc…). Se trata de una **colección estática**, la cual no cambia automáticamente. Se obtiene mediante métodos generales del DOM como *querySelector…, childNodes, parentNode,* etc, entre otras operaciones que devuelven nodos relacionados con el DOM.

```
const coleccion = document.getElementsByTagName("p");
const lista = document.querySelectorAll("p");

console.log(coleccion.length); // cambia si añadimos un <p>
console.log(lista.length); // no cambia
```

Diferencia entre HTMLCollection y NodeList

Tanto HTMLCollection como NodeList no se consideran Arrays de JavaScript, por lo tanto no admiten propiedades y métodos de un Array.

Solución:

Convertir tanto un **NodeList** como un **HTMLCollection** en un array utilizando **Array.from()** o el **operador spread**.

Una vez que se convierten en arrays, se podrán usar métodos como forEach, map, etc…

NOTA: se puede utilizar **for…of** para recorrer tanto **HTMLCollection** como **NodeList**.

```
const coleccion = document.getElementsByTagName("p");
const arrayColeccion = Array.from(coleccion);

arrayColeccion.forEach(element => {
  console.log(element);
});
```

Funciones del DOM: Modificar contenido

Una vez accedemos a un elemento, podemos **leer o modificar** su contenido:

- **elemento.textContent**: accede o modifica el contenido de texto de un elemento

```
p.textContent = "Nuevo texto";
```

- **elemento.innerHTML**: accede o modifica el contenido HTML de un elemento

```
div.innerHTML = "<b>Hola</b>";
```

- **elemento.outerHTML**: devuelve el HTML completo de un elemento

```
console.log(div.outerHTML);
```

Funciones del DOM: Atributos de los elementos

Podemos manipular los atributos de los elementos con los siguientes métodos:

- **elemento.getAttribute(“nombre”)**: obtiene el valor de un atributo

```
img.getAttribute("src");
```

- **elemento.setAttribute(“nombre”, “valor”)**: crea o modifica el valor de un atributo

```
img.setAttribute("alt", "Foto");
```

- **elemento.removeAttribute(“nombre”)**: elimina un atributo

```
img.removeAttribute("title");
```

- **elemento.hasAttribute(“nombre”)**: comprueba si existe un atributo con ese nombre

```
img.hasAttribute("src");
```

Funciones del DOM: estilos y clases

También podemos cambiar el aspecto visual de un elemento:

- **elemento.style.propiedad = “valor”**: modifica el estilo de un elemento

```
elemento.style.color = "blue";
elemento.style.backgrounColor = "lightgray";
```

- **elemento.classList.add(“clase”)**: agrega una clase CSS a un elemento
- **elemento.classList.remove(“clase”)**: elimina una clase CSS de un elemento
- **elemento.classList.toggle(“clase”)**: agrega o quita una clase CSS según su estado comprueba si existe un atributo con ese nombre
- **elemento.classList.contains(“clase”)**: devuelve true/false dependiendo de si el elemento tiene la clase o no

```
elemento.classList.add("activo");
elemento.classList.remove("oculto");
elemento.classList.toggle("resaltado");
elemento.classList.contains("activo");
```

Funciones del DOM: crear y eliminar elementos

También podemos añadir o eliminar elementos HTML desde JavaScript:

- **document.createElement(“etiqueta”)**: crea un nuevo elemento
- **elemento.appendChild(nuevoElemento)**: agrega un elemento como hijo de otro
- **elemento.removeChild(elementoHijo)**: elimina un elemento hijo existente
- **elemento.remove()**: elimina el propio elemento
- **elemento.insertBefore(nuevoElemento, referencia)**: inserta un nuevo elemento antes que el elemento de referencia

```
const lista = document.querySelector("ul");
const nuevo = document.createElement("li");
nuevo.textContent = "Nuevo elemento";
lista.appendChild(nuevo);
```

Funciones del DOM: insertar adyacentes

Para insertar trozos de HTML sin convertirlo a elementos en el DOM utilizando JavaScript, puedes utilizar las siguientes funciones y métodos:

- **insertAdjacentHTML():** puedes especificar la posición de inserción utilizando los siguientes valores como argumento:
  - **beforebegin**: antes del elemento en sí
  - **afterbegin**: dentro del elemento, al comienzo
  - **beforeend**: dentro del elemento, al final
  - **afterend**: después del elemento

```
const element = document.getElementById("myElement");
element.insertAdjacentHTML("beforebegin", "<p>Antes del elemento</p>");
```

Funciones del DOM: insertar adyacentes

- **insertAdjacentElement():** similar a insertAdjacentHTML(), pero esta función permite insertar un elemento adyacente a otro elemento en el DOM. Puedes usar un elemento HTML como argumento.

```
const element = document.getElementById("myElement");
const newElement = document.createElement("p");
newElement.textContent = "Nuevo párrafo";
element.insertAdjacentElement("beforebegin", newElement);
```

Funciones del DOM: insertar adyacentes

- **insertAdjacentText():** Esta función te permite insertar texto adyacente a un elemento en el DOM en una posición específica.

```
const element = document.getElementById("myElement");
element.insertAdjacentText("beforebegin", "Texto antes del elemento");
```

- Asegúrate de reemplazar `"myElement"` con el ID del elemento al que deseas insertar contenido y selecciona la posición de inserción adecuada según tus necesidades. Estos métodos son útiles cuando necesitas agregar código HTML, elementos o texto adyacente a un elemento existente en el DOM de manera fácil y flexible

Riesgo de inyección de código (XSS)

Utilizar métodos como **write, writeln, innerHTML, outerHTML** o **insertAdjacentHTML()** puede dar lugar a lo que se conoce como **inyección de código**.

XSS o Cross-Site Scripting es un tipo de ataque en el cual los actores maliciosos logran inyectar un script malicioso en un sitio web para luego ser procesado y ejecutado.

Si insertamos contenido HTML que proviene del usuario el navegador **podría ejecutar malicioso**. Cualquier texto que venga del usuario o de una fuente externa no debe insertarse directamente, porque el navegador podría ejecutarlo como si fuera código.

¿CÓMO LO EVITAMOS?

- Usando **.textContent** para mostrar texto proveniente del usuario.
- Validando o limpiando los datos antes de insertarlos.

Riesgo de inyección de código (XSS)

EJEMPLO INSEGURO

```
div.innerHTML = "<p>" + userInput + "</p>"; // peligro
```

EJEMPLO SEGURO

```
div.textContent = userInput; // muestra texto sin ejecutar código
```

Eventos del DOM. Interacción con el usuario

¿Qué es un evento?

Un **evento** es una acción o suceso que ocurre en la página web.

Puede ser **provocado por el usuario** (si hace clic en la página, pulsa una tecla…) o **por el navegador** (carga, error, scroll…).

JavaScript permite **detectar estos eventos y responder** ante ellos ejecutando una función.

¿Cómo funcionan los eventos?

Mecanismo de los eventos

El navegador detecta una acción del usuario (por ejemplo, un clic).

Se genera **un objeto de evento** que contiene información sobre lo ocurrido.

Si el elemento tiene asociada una **función manejadora**, esta se ejecuta.

```
<button id="btn">Haz clic</button>
<script>
const boton = document.getElementById("btn");
boton.onclick = () => {
    alert("Has hecho clic en el botón");
};
</script>
```

Tipos de eventos

Los tipos de eventos más comunes son:

- **Eventos del ratón** (MouseEvent): mousedown, mouseup, click, dblclick, mousemove, mouseover, mousewheel, mouseout, contextmenu
- **Eventos del teclado** (KeyboardEvent): keydown, keypress, keyup
- **Eventos de la ventana**: scroll, resize, load, unload
- **Eventos de formularios**: focus, blur, change, submit

Tipos de eventos. Eventos del mouse

Nombre del evento	Descripción del evento
click / dbclick	Cuando hacemos **click / doble click de ratón** sobre un elemento de la página (mouse down / mouseup en el mismo elemento). **Botón izquierdo.**
mouseenter / mouseleave	Cuando el cursor del ratón entra o sale de un elemento.
mouseover / mouseout	Cuando el cursor del ratón entre o sale de un elemento **o de alguno de sus hijos**.
mousedown / mouseup	Cuando presiono / libero **cualquier botón del ratón, da igual que no sea en el mismo elemento**
drag / dragstart / dragend / dragenter / dragleave / dragover	Eventos relacionados con el movimiento de elementos “arrastrables” (draggable = true) a lo largo de la pantalla
drop	Cuando “libero” el elemento “arrastrable” (draggable) en su destino.

Tipos de eventos. Eventos del teclado

Nombre del evento	Descripción del evento
keypress / keyup	Cuando el usuario **presiona / libera** una tecla.
keydown	Cuando el usuario **mantiene una tecla presionada**.

Tipos de eventos. Eventos del navegador

Nombre del evento	Descripción del evento
load	Cuando la página **se ha acabado de cargar** (si se asocia a <body>).
unload	Cuando **cerramos una página o nos dirigimos a otra** (si se asocia a <body>).
beforeunload	Justo **antes de cerrar la página o dirigirnos a otra** (si se asocia a <body>).
resize	Cuando **se redimensiona la ventana** del navegador.
scroll	Se activa cuando se realiza un **desplazamiento (scroll) en la página**.
DOMContentLoaded	Cuando el DOM se ha cargado completamente.

Captación de eventos

Formas de capturar eventos

Existen tres maneras principalmente de asociar eventos a un elemento:

- Desde el **HTML (inline)**: forma antigua y no recomendada.

```
<div onclick="miFuncion(event)"></div>

<script>
    function miFuncion(e) {
        console.log("Has hecho click");
    }
</script>
```

Formas de capturar eventos

- Desde el **Script** con **Método Propio**: forma clásica.

```
<div id="caja"></div>

<script>
    function miFuncion(e) {
        console.log("Has hecho click");
    }
    let caja = document.getElementById("caja");
    caja.onclick = miFuncion;
</script>
```

Formas de capturar eventos

- Desde el **Script** con **addEventListener(…) / removeEventListener ()**: forma moderna y recomendada.

```html
<div id="caja"></div>

<script>
    function miFuncion(e) {
        console.log("Has hecho click");
    }
    let caja = document.getElementById("caja");
    caja.addEventListener("click", miFuncion);
</script>
```

```html
<div id="caja"></div>

<script>
    function miFuncion(e) {
        console.log("Has hecho click");
    }
    let caja = document.getElementById("caja");
    // Para deshabilitar un manejador de eventos
    caja.removeEventListener("click", miFuncion);
</script>
```

Esta manera tiene varias ventajas:

- Permite varios manejadores para un mismo evento.
- Separa HTML de JavaScript.
- Facilita eliminar eventos con removeEventListener()

El objeto EVENT

Cada vez que ocurre un evento, el navegador crea un objeto *event* con información sobre el suceso.

Propiedad	Descripción
.type	Tipo de evento.
.target / .currentTarget	Elemento que lanzó el evento / cuyos manejadores lanzaron el evento.
.stopPropagation() / .stopImmediatePropagation()	Para la propagación / Para también otros listeners que puedan estar capturando dicho evento.
.pageX / .pageY	Posición X / Y de donde se produjo el evento.
clientX / clientY	Coordenadas del ratón.
.preventDefault()	Evita el comportamiento por defecto.

Ejemplos

El evento **window.onload** se ejecuta cuando toda la página se ha cargado, incluidos estilos y scripts.

```
<script>
    function load() {
        alert("Se ha cargado la página completamente");
    }
    window.onload = load;
</script>
```

Ejemplos

Asignar un evento por **tag**

```
<script>

    window.onload = function(){
        let parrafo = document.getElementsByTagName('p');
        parrafo[0].addEventListener('click', showMessage);
    };

    const showMessage = () => {
        alert("Un mensaje");
    }
</script>
<body>
    <p>Esto es un parrafo</p>
</body>
```

Ejemplos

Asignar un evento por **clase**

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', showMessage);
    };

    const showMessage = () => {
        alert("Un mensaje");
    }
</script>
<body>
    <p class="parrafo">Esto es un parrafo</p>
</body>
```

Ejemplos

Cambiar el texto de un **tag**

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', changeMessage);
    };

    const changeMessage = function () {
        this.innerText = 'Un mensaje';
    }
</script>
<body>
    <p class="parrafo">Esto es un parrafo</p>
</body>
```

Ejemplos

Modificar código **HTML**

```
<script>
    window.onload = function(){
        let container = document.getElementsByClassName('container');
        container[0].addEventListener('click', changeMessage);
    };
    const changeMessage = function () {
        this.innerHTML = '<h1>Un mensaje</h1>';
    }
</script>
<body>
    <div class="container">
        <p class="parrafo">Esto es un parrafo</p>
    </div>
</body>
```

Ejemplos

Añadir código **HTML**

```
<script>
    window.onload = function(){
        let container = document.getElementsByClassName('container');
        container[0].addEventListener('click', addMessage);
    };
    const addMessage = function () {
        let h1 = document.createElement("h1");
        h1.innerText ="Un mensaje";
        this.append(h1);
    }
</script>
<body>
    <div class="container">
        <p class="parrafo">Esto es un parrafo</p>
    </div>
</body>
```

Ejemplos

Añadir código **HTML**

```
<script>
    window.onload = function(){
        let container = document.getElementsByClassName('container');
        container[0].addEventListener('click', addMessage);
    };
    const addMessage = function () {
        let div = document.createElement("div");
        let h1 = document.createElement("h1");
        h1.innerText ="Un mensaje";
        div.append(h1);
        let h2 = document.createElement("h2");
        h2.innerText ="Un mensaje más pequeño";
        div.append(h2);
        this.append(div);
    }
</script>
```

Ejemplos

Añadir código **HTML**

```
<script>
    window.onload = function(){
        let container = document.getElementsByClassName('container');
        container[0].addEventListener('click', addMessage);
    };
    const addMessage = function () {
        let div = document.createElement("div");
        let h1 = document.createElement("h1");
        h1.innerText ="Un mensaje";
        div.append(h1);
        let h2 = document.createElement("h2");
        h2.innerText ="Un mensaje más pequeño";
        div.append(h2);
        this.append(div);
    }
</script>
```

Ejemplos

Cambiar el estilo de un elemento

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', changeColor);
    };
    const changeColor = function () {
        this.style.color = 'red';
    }
</script>
<body>
    <div class="container">
        <p class="parrafo">Esto es un parrafo</p>
    </div>
</body>
```

Ejemplos

Cambiar el estilo de un elemento (se recomienda usar classList mejor)

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', changeColor);
    };
    const changeColor = function () {
        this.style.color = 'red';
    }
</script>
<body>
    <div class="container">
        <p class="parrafo">Esto es un parrafo</p>
    </div>
</body>
```

Ejemplos

Añadir una clase

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', addClass);
    };
    const addClass = function () {
        this.classList.add('red');
    }
</script>
<body>
    <div class="container">
        <p class="parrafo">Esto es un parrafo</p>
    </div>
</body>
```

Ejemplos

Quitar una clase

```
<script>
    window.onload = function(){
        let parrafo = document.getElementsByClassName('parrafo');
        parrafo[0].addEventListener('click', removeClass);
    };
    const removeClass = function () {
        this.classList.remove('red');
    }
</script>
<body>
    <div class="container">
        <p class="parrafo red">Esto es un parrafo</p>
    </div>
</body>
```

Propagación de eventos

Cuando ocurre un evento en un elemento, este no se queda solo en ese elemento.

El evento se **propaga** a través del árbol DOM en 3 fases:

- **Captura** -> el evento baja desde *window* hasta el elemento objetivo, **de fuera hacia dentro**.
- **Objetivo** -> el evento llega al elemento donde ocurrió.
- **Burbujeo** -> el evento sube desde el objetivo hasta arriba (padre, abuelo…), **de dentro hacia fuera**.

Por defecto, JavaScript maneja los eventos en **fase de burbujeo**.

Fases de propagación

Para escuchar un evento en fase de **captura** debemos usar el tercer parámetro de *addEventListener*. Si este parámetro se pone como **true** el evento se escucha en captura.

```
elemento.addEventListener("click", funcion, true);
```

Por defecto, JavaScript escucha los eventos en fase de **burbujeo**, por lo que no habría que añadir ningún parámetro a *addEventListener*. Sin embargo, podemos poner el tercer parámetro en **false** para conseguir lo mismo.

```
elemento.addEventListener("click", funcion);
elemento.addEventListener("click", funcion, false);
```

Ejemplo del orden real de ejecución

Estructura

```html
<div id="padre">
    <button id="hijo">Haz clic</button>
</div>
```

Código

```
padre.addEventListener("click", () => console.log("PADRE (captura)"), true);
hijo.addEventListener("click", () => console.log("HIJO (objetivo)"));
padre.addEventListener("click", () => console.log("PADRE (burbujeo)"));
```

Resultado al hacer clic en el botón

```
PADRE (captura)
HIJO (objetivo)
PADRE (burbujeo)
```

Detener la propagación

Para impedir que el evento continúe hacia arriba en el DOM, usamos:

```
event.stopPropagation();
```

Esto hace que **los elementos padre no reciban el evento**.

Uso típico:

- Menús desplegables
- Evitar que un clic cierre un modal
- Contenedores con listeners globales

Delegación de eventos

La delegación consiste en que un **solo listener**, colocado en el **elemento padre**, gestione los eventos de **todos los elementos hijos**, aunque no existan todavía.

```
lista.addEventListener("click", function(e) {
  if (e.target.tagName === "li") {
    e.target.classList.toggle("seleccionado");
  }
});
```

Esto tiene diversas ventajas:

- Menos listeners implica **mayor rendimiento**.
- Funciona con **elementos creados dinámicamente**.
- Código más limpio y estable.

Ejemplo de delegación

```
<ul id="lista">
<li>Elemento 1</li>
<li>Elemento 2</li>
</ul>

<script>
const lista = document.getElementById("lista");

lista.addEventListener("click", function(e) {
    if (e.target.tagName === "li") {
    e.target.classList.toggle("activo");
    }
});
</script>
```

¡A practicar!

Crea una página web que contenga una lista no ordenada vacía y un botón. Al pulsar el botón debe crearse un nuevo elemento de la lista con un texto que indique: *Elemento X* (donde X es un contador que empieza en 1).
Coloca **un único listener** en el la lista para gestionar todas las acciones de sus elementos:

- **Hacer clic en un elemento de la lista se debe cambiar su color**
  Alterna una clase como .seleccionado
  Esa clase debe cambiar el fondo o el color del texto

- **Hacer doble clic en un elemento se debe eliminar**
  Usa remove() sobre event.target

- **Todo debe funcionar también con los elementos creados después (dinámicamente)**

¡A practicar!

Crea una página web que contenga **una imagen principal grande** y **un contenedor con varias miniaturas** (imágenes pequeñas). La imagen principal deberá actualizarse según la miniatura seleccionada.

Coloca **un único listener** en el contenedor de miniaturas para gestionar todas las acciones sobre ellas:

- **Al hacer clic en una miniatura:**
  - La imagen principal debe mostrar esa imagen (puedes cambiar el atributo src de la imagen principal usando event.target.src).
  - La miniatura seleccionada debe cambiar su estilo:
    - Alterna una clase como .activa
    - Esa clase debe modificar visualmente la miniatura (borde, sombra, etc.)

Todo debe funcionar también con miniaturas añadidas posteriormente (si las hubiera), gracias a la delegación de eventos.