

«Talento Tech»

Front-End JS

Clase 12



Clase 12: DOM y eventos

1. Manipulación del DOM

- ¿Qué es el DOM?
- La estructura jerárquica del DOM
- Importancia del DOM en el desarrollo web
- Cómo acceder al DOM desde JavaScript
- Métodos para modificar el DOM:
 - getElementById()
 - getElementsByName()
 - querySelector()
 - querySelectorAll()
- Creación dinámica de elementos HTML con createElement()
- Modificar elementos con innerHTML y textContent
- Eliminar elementos con remove()

2. Eventos en JavaScript

- ¿Qué son los eventos en JS?
- La importancia de los eventos en la interacción usuario-sitio
- Tipos de eventos comunes: click, keypress, submit, etc.
- Cómo "escuchar" eventos en el DOM: addEventListener
- Diferencias entre eventos en atributos HTML, propiedades y addEventListener
- Ejemplos prácticos de eventos
- Eventos con funciones flecha y eventos tradicionales
- Delegación de eventos

Objetivo de la clase:

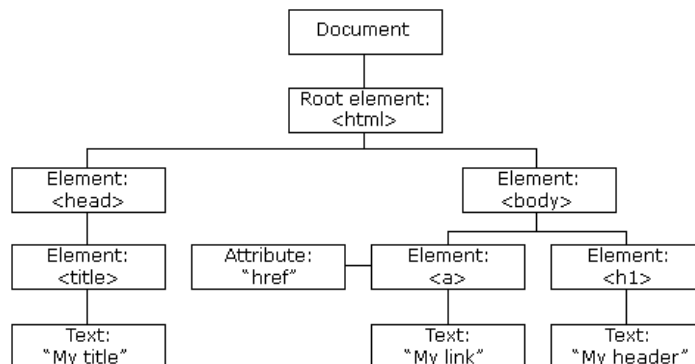
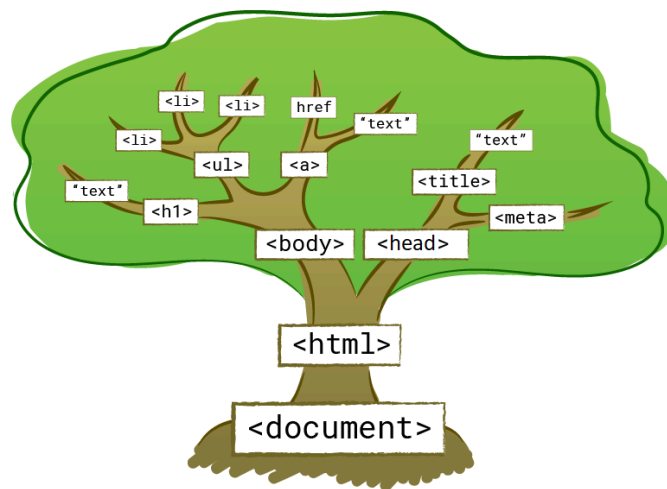
En esta clase aprenderán a interactuar con el contenido de las páginas web mediante la manipulación del DOM, comprendiendo su estructura jerárquica y la forma en que JavaScript puede acceder, modificar o eliminar elementos dinámicamente. Además, explorarán cómo usar eventos para responder a acciones del usuario, construyendo experiencias más interactivas y funcionales. Esto les permitirá sentar bases sólidas para desarrollar aplicaciones web dinámicas y adaptadas a las necesidades reales de los usuarios.

1. Manipulación del DOM

¿Qué es el DOM?

El **DOM (Document Object Model)** es una representación estructurada de la página web. Es la manera en la que el navegador convierte el HTML en una estructura de nodos a los que podemos acceder y modificar mediante JavaScript. Pensá en el DOM como en un árbol donde cada elemento HTML (etiquetas, atributos, textos) es un nodo del mismo.

En resumen, el DOM es la "puerta de entrada" para que JavaScript interactúe con tu página web.



La estructura jerárquica del DOM

El DOM está organizado de forma jerárquica, como un árbol genealógico, en el que los elementos HTML están relacionados entre sí como una familia, siguiendo una línea de herencia. El documento HTML comienza con la etiqueta `<html>`, que tiene como "hijos/hijas" a `<head>` y `<body>`. Dentro de estos, encontramos más elementos anidados (que descienden de otros elementos). Por ejemplo:

```
<!DOCTYPE html>

<html>

  <head>

    <title>Título de la página</title>

  </head>

  <body>

    <h1>Encabezado</h1>

    <p>Este es un párrafo.</p>

  </body>

</html>
```

Aquí, `<html>` es el "madre o padre" de `<head>` y `<body>`, mientras que `<h1>` y `<p>` son "hijos/as" de `<body>`.

Importancia del DOM en el desarrollo web:

El DOM es fundamental porque es lo que le da interactividad a nuestras páginas. Gracias a él, podemos modificar elementos, agregar otros nuevos, o incluso eliminar los que ya no necesitamos, todo en tiempo real y sin recargar la página.

¿Cómo acceder al DOM desde JavaScript?

Para interactuar con el DOM desde JavaScript, siempre empezamos accediendo al objeto `document`, que representa el documento HTML cargado en el navegador. A partir de ahí, podemos usar varios métodos para buscar, seleccionar y modificar elementos.

Métodos para modificar el DOM

Ahora, vamos a profundizar en algunos de los métodos más utilizados para seleccionar y modificar elementos del DOM.

1. getElementById()

Este es uno de los métodos más simples y rápidos para acceder a un elemento específico del DOM. Como su nombre lo indica, selecciona un elemento según su atributo `id`. Recordá que el `id` debe ser único en el documento.

Ejemplo:

```
<h1 id="titulo">Bienvenido a mi sitio</h1>

<script>

    let elemento = document.getElementById("titulo");

    console.log(elemento); // Devuelve el elemento <h1 id="titulo">...</h1>

</script>
```

En este ejemplo, accedemos al `<h1>` usando su `id`. Una vez que tenemos acceso al elemento, podemos modificar su contenido, cambiar estilos o realizar cualquier operación que necesitemos.

Ventaja: Es muy rápido porque los `id` son únicos en la página, por lo que el navegador no tiene que buscar entre muchos elementos.

2. getElementsByClassName()

Este método devuelve una colección de elementos que comparten la misma clase. A diferencia de `getElementById()`, que devuelve un solo elemento, este método devuelve un **HTMLCollection**, lo que significa que podés tener varios elementos con la misma clase y trabajar con todos ellos.

Ejemplo:

```
<div class="caja">Caja 1</div>

<div class="caja">Caja 2</div>

<div class="caja">Caja 3</div>

<script>

    let cajas = document.getElementsByClassName("caja");

    console.log(cajas); // Devuelve una colección con las tres <div
class="caja">...</div>
```

```
// Podemos iterar sobre ellos:

for (let i = 0; i < cajas.length; i++) {

    cajas[i].style.color = "red"; // Cambiamos el color de texto a rojo

}

</script>
```

En este caso, `getElementsByClassName()` selecciona todas las `div` con la clase "caja". Podés luego modificar cualquiera de estos elementos usando índices, ya que es una colección.

3. `querySelector()`

Este es un método mucho más flexible y potente. Nos permite seleccionar el primer elemento que coincida con un selector CSS. Si estás familiarizado con selectores en CSS, te va a resultar muy fácil de usar. **Ejemplo:**

```
<div class="contenedor">

    <p class="texto">Texto 1</p>

    <p class="texto">Texto 2</p>

</div>

<script>

    let primerParrafo = document.querySelector(".contenedor .texto");

    console.log(primerParrafo); // Devuelve el primer <p
class="texto">Texto 1</p>

    // Podemos cambiar su contenido:

    primerParrafo.textContent = "Nuevo texto";

</script>
```

Acá estamos utilizando un selector compuesto para acceder al primer párrafo dentro del `div` con clase "contenedor". Este método es sumamente versátil porque acepta cualquier selector CSS.

4. querySelectorAll()

Este método es similar a `querySelector()`, pero en lugar de devolver sólo el primer elemento que coincide, devuelve todos los elementos que coinciden con el selector, en forma de **NodeList** (una lista de nodos). **Ejemplo:**

```
<div class="contenedor">

  <p class="texto">Texto 1</p>

  <p class="texto">Texto 2</p>

</div>

<script>

  let parrafos = document.querySelectorAll(".contenedor .texto");

  console.log(parrafos); // Devuelve una NodeList con los dos <p
class="texto">...</p>

  // Cambiamos el color de todos los párrafos seleccionados:

  parrafos.forEach(parrafo => {

    parrafo.style.color = "blue";

  });

</script>
```

En este ejemplo, `querySelectorAll()` selecciona todos los párrafos con la clase "texto" dentro del `div` con clase "contenedor", y luego usamos un bucle `forEach` para cambiar el color de todos ellos. A diferencia de `getElementsByClassName()`, que devuelve una colección de HTML, `querySelectorAll()` devuelve una lista de nodos que soporta métodos como `forEach()`.

Creación dinámica de elementos HTML con `createElement()`

Una de las mayores ventajas del DOM es que no solo podés modificar lo que ya está en la página, sino que también podés agregar nuevos elementos en tiempo real. Esto lo hacemos con `createElement()`, que crea un nuevo nodo en memoria, pero que no se añade a la página hasta que usamos `appendChild()`. **Ejemplo:**

```
<ul id="lista">

  <li>Elemento 1</li>

  <li>Elemento 2</li>

</ul>

<script>

  let nuevoElemento = document.createElement("li");

  nuevoElemento.textContent = "Elemento 3";

  document.getElementById("lista").appendChild(nuevoElemento);

</script>
```

En este ejemplo, creamos un nuevo elemento de lista (``) y lo agregamos a la lista existente. Esta técnica es perfecta para páginas dinámicas, donde el contenido puede cambiar o crecer dependiendo de las acciones de las personas usuarias.

Modificar elementos con innerHTML y textContent

innerHTML: Este método te permite modificar o insertar contenido HTML dentro de un elemento. Es muy útil cuando querés agregar contenido más complejo, como listas, tablas o incluso secciones enteras con varias etiquetas HTML.

Ejemplo:

```
<div id="contenido">

  <p>Texto inicial</p>

</div>

<script>

  let div = document.getElementById("contenido");

  div.innerHTML = "<h2>Nuevo título</h2><p>Nuevo párrafo
añadido</p>";

</script>
```

En este ejemplo, usamos **innerHTML** para reemplazar el contenido del **div**. Esto elimina el párrafo original y lo reemplaza por un **h2** y un nuevo **p**. Como ves, con **innerHTML** podés insertar HTML completo, no solo texto.

⚠ Aunque **innerHTML** es muy flexible, debés usarlo con cuidado cuando estás manipulando contenido que proviene de los usuarios, porque puede ser una puerta de entrada para ataques de XSS (Cross-Site Scripting).

textContent: A diferencia de **innerHTML**, este método solo modifica el texto dentro de un elemento, sin interpretar etiquetas HTML. Es mucho más seguro si solo estás manejando texto y querés evitar problemas de seguridad o renderizado inesperado.

Ejemplo:

```
<div id="contenido">

  <p>Texto inicial</p>

</div>

<script>

  let div = document.getElementById("contenido");
```

```
div.textContent = "Este es solo texto, sin HTML"; </script>
```

En este ejemplo, `textContent` reemplaza todo el contenido del `div` con texto plano, sin interpretación de etiquetas HTML. Si hubieras intentado insertar HTML (por ejemplo, un `<h1>`), se mostraría tal cual, como texto, sin ser renderizado como un encabezado.

Eliminar elementos con `remove()`

A veces no sólo queremos agregar o modificar elementos, sino que también necesitamos eliminarlos por completo. Para esto, usamos el método `remove()`, que elimina un nodo del DOM.

Ejemplo:

```
<ul>

  <li id="item1">Elemento 1</li>

  <li id="item2">Elemento 2</li>

  <li id="item3">Elemento 3</li>

</ul>

<script>

  let item = document.getElementById("item2");

  item.remove();

</script>
```

En este caso, seleccionamos el elemento con `id="item2"` y lo eliminamos por completo del DOM. Esto es útil cuando trabajas con listas dinámicas o interfaces donde algunos elementos deben desaparecer dependiendo de la acción del usuario o la usuaria.

2. Eventos en JavaScript

¿Qué son los eventos en JS?

Los **eventos** en JavaScript son acciones que ocurren en el navegador y que podemos "escuchar" o capturar para ejecutar algún código. Estos eventos pueden ser disparados por

distintas interacciones como un clic, una tecla presionada, o el envío de un formulario) o bien por el navegador (carga de la página) u otros elementos interactivos.

En otras palabras, un evento es como una señal de que algo ha pasado en el navegador, y podemos responder a esa señal escribiendo código.

La importancia de los eventos en la interacción usuario-sitio

Sin eventos, una página web sería estática e inerte. Gracias a los eventos, podemos interactuar con la usuaria, permitiéndole realizar acciones como enviar formularios, hacer clic en botones, o incluso mover el ratón sobre elementos para mostrar información adicional. Los eventos son esenciales para la interactividad en las aplicaciones web modernas.

Tipos de eventos comunes: click, keypress, submit, etc.

Algunos de los eventos más comunes en JavaScript incluyen:

- **click**: Se dispara cuando se hace clic en un elemento.
- **keypress**: Se activa cuando se presiona una tecla.
- **submit**: Ocurre cuando un formulario es enviado.
- **mouseover**: Se dispara cuando el ratón pasa por encima de un elemento.
- **load**: Ocurre cuando un recurso (como una página o una imagen) se ha cargado completamente.

¿Cómo "escuchar" eventos en el DOM: addEventListener?

Para "escuchar" estos eventos y ejecutar alguna acción cuando ocurren, usamos `addEventListener`. Este método asocia una función a un evento para que se ejecute cada vez que el mismo ocurra.

Ejemplo:

```
<button id="miBoton">Hacé clic</button>

<script>

  let boton = document.getElementById("miBoton");

  boton.addEventListener("click", function() {

    alert(";Hiciste clic en el botón!");

  });
```

```
</script>
```

En este caso, cuando se haga clic en el botón, aparecerá una alerta. El método `addEventListener` nos permite definir de manera limpia cómo reaccionar a distintos eventos.

Diferencias entre eventos en atributos HTML, propiedades y `addEventListener`

Hay varias formas de manejar eventos en JavaScript:

Atributos HTML: Podés definir eventos directamente en el HTML, pero no es recomendable porque mezcla lógica con estructura.


Ejemplo:

```
<button onclick="alert(';Hiciste clic!')">Hacé clic</button>
```

Propiedades de los elementos: Podés asignar una función directamente a la propiedad del evento del elemento en JavaScript.

Ejemplo:

```
boton.onclick = function() {  
    alert(";Clic detectado!");  
};
```

 **`addEventListener`:** Es la forma recomendada, ya que permite adjuntar múltiples eventos al mismo elemento sin sobrescribir funciones anteriores. Además, es más flexible.

Eventos con funciones flecha y eventos tradicionales

En JavaScript, `this` es una palabra reservada que **apunta al objeto que está ejecutando el código**.

En los eventos del DOM, normalmente *apunta al elemento* que dispara el evento (por ejemplo un botón).

Ejemplo con función tradicional:

```
boton.addEventListener("click", function() {  
  
    console.log(this); // 'this' se refiere al elemento que disparó el  
    evento  
  
});
```

Acá usás una función **tradicional**.

- Cuando hacés click, la función se ejecuta **con `this` apuntando al botón** que disparó el evento.
- O sea, `this` significa “el botón que fue clickeado”.

Por eso podés hacer, por ejemplo:

```
this.style.backgroundColor = "red";
```

Ejemplo con función flecha:

```
boton.addEventListener("click", () => {  
  
    console.log(this); // 'this' se refiere al contexto exterior, no al  
    botón  
  
});
```

Acá usás una **función flecha**.

- Las funciones flecha **no** crean su propio `this`.
- En cambio, **heredan el `this` del contexto donde fueron definidas** (por ejemplo, la ventana global o el objeto exterior).

Por eso, cuando hacés click, `this` *no apunta al botón*, sino al contexto exterior (por ejemplo `window` o el objeto que contiene la función flecha).

Si dentro de una función flecha hacés:

```
this.style.backgroundColor = "red";
```

probablemente te dé error, porque `this` ya no es el botón.

🔴 Si necesitas que `this` apunte al elemento que disparó el evento (por ejemplo un botón), usá funciones tradicionales. Evitá funciones flecha en eventos si vas a trabajar con `this`.

Delegación de eventos

La **delegación de eventos** es una técnica que nos permite escuchar eventos de varios elementos hijos utilizando un solo "listener" en el padre. Es especialmente útil cuando los elementos hijos son generados dinámicamente.

Imaginá esto:

- Tenés muchos elementos (por ejemplo, muchos `` dentro de una lista ``)
- Podrías ponerle un `addEventListener` a cada uno, pero si son muchos (¡o si se agregan dinámicamente!) se vuelve un lío.

En lugar de eso, **le ponés UN SOLO listener** al elemento padre (por ejemplo el ``) y ahí adentro verificás **a quién** le hicieron click.

Ejemplo:

```
<ul id="lista">

  <li>Elemento 1</li>

  <li>Elemento 2</li>

  <li>Elemento 3</li>

</ul>

<script>

  let lista = document.getElementById("lista");

  lista.addEventListener("click", function(event) {

    if (event.target.tagName === "LI") {

      alert("Hiciste clic en " + event.target.textContent);

    }

  })

}
```

```
});  
  
</script>
```

Este código:

1. Agrega un solo **listener** al ``.
2. Cada vez que hacés click en algo dentro de esa lista, el evento sube al ``.
3. Usando `event.target`, podés ver **en qué elemento realmente se hizo clic**.
4. Si el `target` es un ``, mostrás el alert.

🔴 ¿Por qué se llama delegación?

Porque el elemento padre *delega su capacidad de escuchar* a todos sus hijos. El evento sube (propaga) hasta el padre, y ahí decidís si te interesa capturarlo o no.

- **Menos código.**
- Más fácil de mantener
- Si agregás nuevos `` con JS después, ¡no hace falta volver a ponerles listeners uno por uno!

Por eso se usa mucho con elementos dinámicos.

💬 Storytelling

¡Dinamizamos la web mostrando productos y agregando interactividad con JavaScript!

Talento⁷
Lab

Tomás (Desarrollador Senior)



¡Vamos avanzando muy bien! Ahora es momento de llevar a la práctica la manipulación dinámica del contenido con JavaScript y hacer que tu sitio web sea más interactivo para los usuarios.

Ejercicio práctico #1:

Mostrar la lista de productos en la página

Lucía (Product Owner)



nos pidió que mostremos los productos que generamos la clase pasada, pero esta vez directamente en la página web. La idea es que cada producto aparezca como una tarjeta con su nombre y precio, para que se vean ordenados y claros.

Pasos a seguir:

1. En tu archivo HTML, crea un contenedor para las tarjetas de productos. Por ejemplo, un `<div>` con un id como `contenedor-productos`.
2. Decide cómo quieres que se muestren las tarjetas:
 - Puedes crear un botón para mostrar la lista al hacer clic.
 - O hacer que las tarjetas se muestren automáticamente al cargar la página.
3. En tu archivo JavaScript, accede al contenedor creado en el HTML usando un selector (`getElementById` o similar).
4. Recorre el array de productos que generaste en la clase anterior.
5. Por cada producto del array:
 - Crea un nuevo elemento `div` que servirá como tarjeta y asígnale una clase (por ejemplo, `.card`).
 - Dentro de esa tarjeta, agrega elementos que contengan el nombre del producto (por ejemplo, un `<h3>`) y el precio (por ejemplo, un `<p>`).
 - No agregues la descripción por ahora.
6. Inserta cada tarjeta dentro del contenedor en el HTML usando un método adecuado (como `appendChild`).

Tips de Tomás:



Usá `createElement()` para crear cada nuevo elemento HTML (como `divs`, `h3`, `p`).

Para agregar cada tarjeta al contenedor, utilizá `appendChild()`.

Recordá que manipular el DOM es más sencillo si trabajás paso a paso: primero creás el elemento, luego le agregás contenido, y por último lo insertás en el documento.

Ejercicio práctico #2:

Agregar descripción dinámica con botón

Lucía (Product Owner)



Ahora necesitamos que cada tarjeta tenga un botón que permita al usuario ver la descripción completa del producto, pero solo cuando quiera.

Pasos a seguir:

1. Al crear cada tarjeta, agrega un botón pequeño dentro de ella con el texto “Ver descripción”.
2. Para cada botón, agrega un listener de eventos (`addEventListener`) para detectar cuando se hace clic.
3. En la función que maneja el clic:
 - Verifica si la descripción ya está visible en la tarjeta para evitar que se duplique.
 - Si la descripción no está visible, crea un nuevo elemento `<p>` con el texto de la descripción del producto.
 - Agrega ese párrafo dentro de la tarjeta (usando `appendChild` o similar).
 - Si la descripción ya está visible, puedes decidir ocultarla o simplemente ignorar el clic.

Tips de Tomás:



Para agregar eventos usá siempre `addEventListener()` en lugar de poner el evento directo en el HTML.

Antes de crear y agregar la descripción, chequeá si ya existe para no duplicar contenido.

Pensá en la experiencia del usuario: que el botón tenga una acción clara y no genere elementos repetidos.

Practicá crear y manejar elementos dinámicamente para ganar fluidez con el DOM y los eventos.



Buenos Aires
aprende
Agencia de Habilidades para el Futuro

BA Buenos
Aires
Ciudad