

La **Manipulación del Document Object Model (DOM)** es un pilar fundamental en el desarrollo web moderno. Permite a **JavaScript** interactuar con la estructura, el contenido y los estilos de una página HTML o XML, transformándola de un documento estático en una experiencia dinámica y responsiva para el usuario.

## 1. Definición y Estructura del DOM

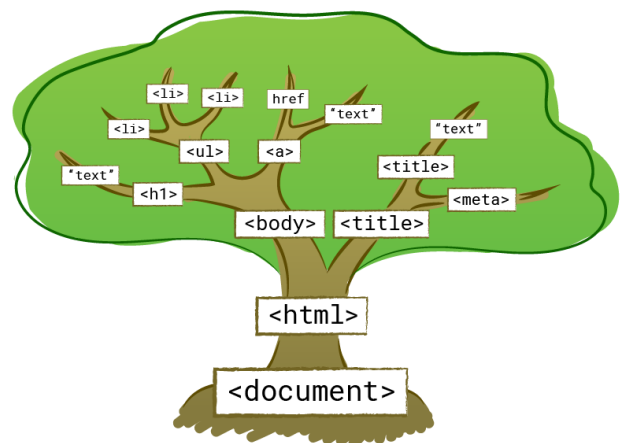
El **Document Object Model (DOM)** es una interfaz de programación de aplicaciones (API) que representa la estructura lógica de los documentos HTML y XML. Proporciona una forma estándar para que programas como JavaScript accedan, modifiquen y gestionen el contenido web. Cuando un navegador carga una página web, construye una representación en memoria de esa página en forma de un **árbol de nodos**. Cada parte del documento (elementos, atributos, texto, comentarios) se convierte en un **objeto** (o nodo) dentro de este árbol, creando una jerarquía que refleja la anidación del HTML.

## 1.1 Estructura en Árbol y Tipos de Nodos Fundamentales

La organización del DOM es jerárquica, similar a un árbol genealógico.

Cada nodo puede tener:

- **Un nodo padre:** El elemento que lo contiene directamente.
- **Cero o más nodos hijos:** Los elementos o texto que contiene directamente.
- **Nodos hermanos:** Nodos que comparten el mismo padre



Tipo de Nodo	Descripción	Representación en JS	Ejemplo HTML
Documento	La raíz de todo el árbol. Representa la página HTML completa.	document	<!DOCTYPE html>
Elemento	Corresponde a las etiquetas HTML. Contienen otros nodos (hijos) y atributos.	HTMLElement (ej. div)	<div>, <p>, <img>
Texto	Representa el contenido textual puro dentro de un elemento.	Text	Hola mundo (dentro de un <p>)
Atributo	Representa los atributos de las etiquetas HTML (ej., id, class, src, href). Se accede a través de nodos de elemento.	Attr	id="miElemento"
Comentario	Nodos que representan comentarios HTML.	Comment	<!-- Mi comentario -->

## Ejemplo de Estructura:

```
2  <html lang="es">
3  <head>
4  |   <title>Pagina Web</title>
5  </head>
6  <body>
7  |   <div id="contenedor">
8  |   <h1>Título Principal</h1>
9  |   <p class="introduccion">Este es un párrafo.</p>
10 |   <!-- Esto es un comentario -->
11 </div>
12 </body>
```

En este ejemplo:

- `document` es el padre de `<html>`.
- `<div id="contenedor">` es un nodo de elemento, hijo de `<body>`.
- `<h1>` y `<p>` son hijos de `<div id="contenedor">` y hermanos entre sí.
- "Título Principal" y "Este es un párrafo." son nodos de texto.
- `id="contenedor"` y `class="introduccion"` son nodos de atributo.

## 2. Selección de Elementos del DOM

Para manipular un elemento, este debe ser primero seleccionado. JavaScript ofrece métodos que permiten localizar nodos específicos en el árbol del DOM.

### 2.1. `document.getElementById('idElemento')`

- **Propósito:** Seleccionar un único elemento por su atributo id.
- **Retorno:** El objeto `Element` si se encuentra, `null` en caso contrario.
- **Consideración:** Los `id` deben ser **únicos** en el documento. Este es el método más rápido para selección directa por ID.

```
6  <body>
7  <div id="miContenedor">Contenido</div>
8  <button id="btnAccion">Clic</button>
9  <script>
10  const contenedor = document.getElementById('miContenedor');
11  const boton = document.getElementById('btnAccion');
12
13  console.log(contenedor); // Muestra el elemento <div>
14  console.log(boton.textContent); // Muestra "Clic
15  </script>
16 </body>
```

## 2.2. document.querySelector('selectorCSS')

- **Propósito:** Seleccionar el **primer elemento** que coincida con un selector CSS.
- **Retorno:** El primer objeto **Element** coincidente, **null** si no se encuentra.
- **Consideración:** Permite usar cualquier selector CSS válido (clases, IDs, etiquetas, atributos, pseudoclases, combinadores).

```
1  <!DOCTYPE html>
2  <html lang="es">
3  <head>
4  |   <title>Pagina Web</title>
5  </head>
6  <body>
7  <div class="caja">Caja 1</div>
8  <div class="caja">Caja 2</div>
9  <p id="info">Info</p>
10 <script>
11   const primeraCaja = document.querySelector('.caja');
12   console.log(primeraCaja.textContent);
13
14   const infoParrafo = document.querySelector('#info');
15   console.log(infoParrafo.tagName);
16 </script>
17 </body>
18 </html>
```

## 2.3. document.querySelectorAll('selectorCSS')

- **Propósito:** Seleccionar **todos los elementos** que coincidan con un selector CSS.
- **Retorno:** Una **NodeList** estática con todos los elementos coincidentes. Una NodeList vacía si no hay coincidencias.
- **Consideración:** La NodeList es similar a un array y se puede iterar con `forEach()`. Es "estática", lo que significa que no se actualiza automáticamente si los elementos cambian después de la selección.

```
<body>
<ul>
|   <li class="item">Uno</li>
|   <li class="item">Dos</li>
</ul>
<script>
const todosLosItems = document.querySelectorAll('.item');
console.log(todosLosItems.length); // Muestra 2

todosLosItems.forEach(item => {
|   console.log(item.textContent); // Itera y muestra "Uno", "Dos"
});
</script>
</body>
```

## 2.4. document.getElementsByClassName('nombreClase')

- **Propósito:** Seleccionar todos los elementos que tienen una clase CSS específica.
- **Retorno:** Un **HTMLCollection** "en vivo" (live).
- **Consideración:** El **HTMLCollection** es "en vivo", lo que significa que se actualiza automáticamente si se añaden o eliminan elementos en el documento que coinciden con la clase. No tiene `forEach()` directamente; se debe usar `Array.from()` o un bucle `for` tradicional para iterar.

```
6   <body>
7   <p class="nota">Importante</p>
8   <span class="nota">Advertencia</span>
9   <script>
10  const notas = document.getElementsByClassName('nota');
11  console.log(notas.length); // Muestra 2
12  // Convertir a Array para usar forEach
13  Array.from(notas).forEach(el => {
14    |   el.style.backgroundColor = 'yellow';
15    | });
16  </script>
17  </body>
18  </html>
```

## 2.5. document.getElementsByTagName('nombreEtiqueta')

- **Propósito:** Seleccionar todos los elementos con una etiqueta HTML específica.
- **Retorno:** Un **HTMLCollection** "en vivo".
- **Consideración:** Similar a **getElementsByClassName** en su comportamiento de "en vivo" y la necesidad de convertir a **Array** para **forEach()**.

```
6   <body>
7   <div></div>
8   <span></span>
9   <div></div>
10  <script>
11  const divs = document.getElementsByTagName('div');
12  console.log(divs.length); // Muestra 2 (inicialmente)
13
14  const nuevoDiv = document.createElement('div');
15  document.body.appendChild(nuevoDiv);
16
17  console.log(divs.length); // Muestra 3 (se actualiza automáticamente)
18  </script>
19  </body>
```

### 3. Modificación de Contenido

Una vez que un elemento ha sido seleccionado, su contenido puede ser modificado.

#### 3.1. elemento.textContent

- **Propósito:** Obtener o establecer el **contenido de texto plano** de un elemento y sus descendientes. Ignora el marcado HTML.
- **Uso:** Seguro y rápido para manipular texto. Si se asigna HTML, este se interpreta como texto literal.

```
6   <body>
7   <p id="saludo">Hola, <strong>mundo</strong>.</p>
8
9   <script>
10  const saludo = document.getElementById('saludo');
11  console.log(saludo.textContent);
12
13  saludo.textContent = '¡Ten un buen día!';
14  </script>
15  </body>
```

#### 3.2. elemento.innerHTML

- **Propósito:** Obtener o establecer el **contenido HTML completo** de un elemento, incluyendo etiquetas y texto.
- **Uso:** Potente para insertar estructuras HTML dinámicamente.
- **Advertencia de Seguridad:** Susceptible a ataques de **Cross-Site Scripting (XSS)** si se utiliza con contenido no sanitizado proveniente de fuentes externas o de usuario. Siempre sanitice las entradas antes de insertarlas con **innerHTML**.

```
6   <body>
7   <div id="zonaContenido">texto que se va a remplazar</div>
8   <script>
9   const zona = document.getElementById('zonaContenido');
10  zona.innerHTML = '<h2>Noticia</h2><p>El DOM permite crear <strong>contenido dinámico</strong>.</p>';
11  </script>
12  </body>
```

## 4. Modificación de Atributos y Estilos

El DOM permite el control programático de los atributos HTML y la apariencia visual de los elementos mediante CSS.

### 4.1. Manipulación de Atributos

Los atributos definen propiedades adicionales de los elementos HTML (ej., **src** para imágenes, **href** para enlaces, **class** para estilos).

- **elemento.setAttribute('nombreAtributo', 'valor'):**
  - **Función:** Establece el valor de un atributo o lo crea si no existe.
  - **Ejemplo:**

```
✓ <body>
  
✓ <script>
  const imagen = document.getElementById('miImagen');
  imagen.setAttribute('src', 'nueva_imagen.png');
  imagen.setAttribute('alt', 'Imagen descriptiva');
</script>
</body>
```

**elemento.getAttribute('nombreAtributo'):**

- **Función:** Obtiene el valor actual de un atributo.
- **Retorno:** El valor del atributo (**string**) o **null** si no existe.
- **Ejemplo:**

```
<body>
<a id="miEnlace" href="https://google.com">Enlace</a>
<script>
const enlace = document.getElementById('miEnlace');
const url = enlace.getAttribute('href');
</script>
</body>
```

**element.removeAttribute('nombreAtributo'):**

- **Función:** Elimina un atributo del elemento.
- **Ejemplo**

```
6   <body>
7   <input id="miInput" type="text" disabled>
8   <script>
9   const input = document.getElementById('miInput');
10  input.removeAttribute('disabled');
11  </script>
12  </body>
```

## 4.2. Manipulación de Estilos

Existen dos enfoques principales para cambiar estilos: directamente en línea o gestionando clases CSS.



### 4.2.1. elemento.style.propiedadCSS

- **Propósito:** Acceder y modificar directamente los estilos CSS en línea de un elemento.
- **Convención:** Las propiedades CSS con guiones (background-color, font-size) se escriben en **CamelCase** en JavaScript (backgroundColor, fontSize).
- **Uso:** Adecuado para cambios de estilo dinámicos, pequeños y puntuales. Para estilos complejos, es más mantenible usar clases CSS.

```
<body>
<div id="cajaColor" style="background-color:  lightgray;">
</div>
<script>
const caja = document.getElementById('cajaColor');
caja.style.backgroundColor = 'blue';
caja.style.width = '200px';
caja.style.height = '100px';
caja.style.borderRadius = '10px'
</script>
</body>
```

### 4.2.2. elemento.classList

- **Propósito:** Gestionar las clases CSS de un elemento. Es la forma preferida para aplicar y quitar grupos de estilos definidos en hojas de estilo externas, lo que promueve un código más limpio y modular.
- **elemento.classList.add('clase1', 'clase2', ...):**
  - **Función:** Añade una o más clases.
  - **Ejemplo:**

```
5      <style>
6      .btn-activo {
7          background-color:  green;
8          color:  yellow;
9      }
10     </style>
11     </head>
12     <body>
13     <button id="btn" class="btn-base">Botón</button>
14     <script>
15     const btn = document.getElementById('btn');
16     btn.classList.add('btn-activo');|
17     </script>
```



- `elemento.classList.remove('clase1', 'clase2', ...):`
  - **Función:** Elimina una o más clases.
  - **Ejemplo:**

```
<head>
|   <title>Pagina Web</title>
</head>
<body>
<div id="alerta" class="mensaje error">Error!</div>
<script>
const alerta = document.getElementById('alerta');
alerta.classList.remove('error');
</script>
</body>
```

- `elemento.classList.toggle('clase'):`
  - **Función:** Si la clase existe, la elimina; si no existe, la añade. Ideal para alternar estados.
  - **Ejemplo:**

```
<head>
|   <title>Pagina Web</title>
|   <style>
|       .oculto {
|           display: none;
|       }
|       .visible
|       {
|           display: block;
|       }
|   </style>
</head>
<body>
<div id="menu" class="oculto">Menú</div>
<script>
const menu = document.getElementById('menu');
menu.classList.toggle('visible');
</script>
```



```

18 <script>
19     const nuevoDiv = document.createElement('div');
20     const nuevoParrafo = document.createElement('p');
21     nuevoParrafo.textContent = 'Este párrafo fue creado dinámicamente.';
22     nuevoDiv.classList.add('tarjeta-dinamica');
23     nuevoDiv.appendChild(nuevoParrafo);
24     document.body.appendChild(nuevoDiv);
25 </script>

```

- **elemento.classList.contains('clase'):**
  - **Función:** Verifica si el elemento tiene una clase específica.
  - **Retorno:** true o false.
  - **Ejemplo:**

```

6 <body>
7 <input id="campo" class="validado" value="abc">
8 <script>
9     const campo = document.getElementById('campo');
10    if (campo.classList.contains('validado')) {
11        console.log('El campo ha sido validado.');

```

## 5. Creación y Eliminación de Elementos

La capacidad de añadir o remover elementos dinámicamente es crucial para la construcción de interfaces de usuario interactivas y reactivas.

### 5.1. Creación de Elementos: `document.createElement()`

- **Propósito:** Crear un nuevo nodo de elemento HTML en la memoria del navegador.
- **Importante:** El elemento recién creado aún no es parte del documento visible. Debe ser insertado en el árbol del DOM mediante métodos de inserción.

```

18 ∨ <script>
19     const nuevoDiv = document.createElement('div');
20     const nuevoParrafo = document.createElement('p');
21     nuevoParrafo.textContent = 'Este párrafo fue creado dinámicamente.';
22     nuevoDiv.classList.add('tarjeta-dinamica');
23 </script>

```

## 5.2. Inserción de Elementos: `padre.appendChild(hijo)`

- **Propósito:** Añadir un nodo (**hijo**) como el **último hijo** de un nodo padre especificado.
- **Uso:** El método más común para integrar elementos creados con `createElement()` al documento visible. Si el nodo **hijo** ya estaba en el DOM, es movido a la nueva posición (no se duplica).

```
18  <script>
19      const nuevoDiv = document.createElement('div');
20      const nuevoParrafo = document.createElement('p');
21      nuevoParrafo.textContent = 'Este párrafo fue creado dinámicamente.';
22      nuevoDiv.classList.add('tarjeta-dinamica');
23      nuevoDiv.appendChild(nuevoParrafo);
24      document.body.appendChild(nuevoDiv);
25  </script>
```

## 5.3. Eliminación de Elementos: `padre.removeChild(hijo)`

- **Propósito:** Eliminar un nodo **hijo** específico del DOM.
- **Consideración:** Se debe llamar al método desde el **elemento padre** del nodo que se desea eliminar. Es fundamental tener una referencia tanto al padre como al hijo.

```
7      <body>
8      <ul id="miLista">
9          <li id="item1">Item a borrar</li>
10         <li id="item2">Otro item</li>
11     </ul>
12     <script>
13         const miLista = document.getElementById('miLista');
14         const itemABorrar = document.getElementById('item1');
15
16         if (itemABorrar && miLista.contains(itemABorrar)) {
17             miLista.removeChild(itemABorrar);
18         }
19     </script>
20 </body>
```

## 6. Eventos del DOM

Los eventos son la base de la interactividad en el desarrollo web. Son notificaciones que el navegador envía cuando algo significativo sucede (ej., un clic, una pulsación de tecla, la carga de la página). JavaScript permite "escuchar" estos eventos y ejecutar funciones en respuesta.

### 6.1. `elemento.addEventListener('tipoDeEvento', funcionManejadora)`

- **Propósito:** Registrar una función ("manejador de evento" o "callback") que se ejecutará cuando un tipo específico de evento ocurra en un elemento.
- **Argumentos:**
  - **'tipoDeEvento':** Una cadena que especifica el tipo de evento a escuchar (ej., `'click'`, `'mouseover'`, `'keydown'`).
  - **funcionManejadora:** La función (callback) que se ejecuta cuando el evento se dispara. Esta función recibe automáticamente un objeto `Event` como su primer argumento, que contiene información contextual sobre el evento.
- **Ventajas:** Permite añadir múltiples manejadores para el mismo evento en un mismo elemento sin sobrescribirse.

```
7   <body>
8   <button id="miBoton" class="estilo-boton">Haz clic</button>
9   <p id="feedback">Mensaje aquí</p>
10  <script>
11    const boton = document.getElementById('miBoton');
12    const feedback = document.getElementById('feedback');
13    boton.addEventListener('click', function(event) {
14      feedback.textContent = '¡El botón ha sido clicado!';
15      console.log('Tipo de evento:', event.type);
16      console.log('Elemento objetivo:', event.target);
17    });
18    boton.addEventListener('mouseover', () => {
19      boton.style.backgroundColor = '#4CAF50';
20    });
21    boton.addEventListener('mouseout', () => {
22      boton.style.backgroundColor = '';
23    });
24  </script>
```

## 6.2. Tipos de Eventos Comunes

Categoría	Tipo de Evento	Descripción
Ratón	click	Elemento clicado.
	dblclick	Elemento doble clicado.
	mouseover	Puntero del ratón entra en el área del elemento.
	mouseout	Puntero del ratón sale del área del elemento.
	mousemove	Ratón se mueve sobre el elemento.
Teclado	keydown	Tecla es presionada (se repite mientras se mantiene).
	keyup	Tecla es liberada.
Formulario	submit	Formulario es enviado.
	change	El valor de un campo (<input>, <select>, <textarea>) cambia.
	input	El valor de un campo cambia (en tiempo real mientras se escribe).
	focus	Un elemento obtiene el foco.
	blur	Un elemento pierde el foco.
Carga/Ventana	load	Todo el recurso (página, imágenes, scripts) ha cargado.
	DOMContentLoaded	El documento HTML ha sido completamente cargado y parseado.
	resize	La ventana del navegador cambia de tamaño.
	scroll	El usuario se desplaza por el documento.

## 7. Conceptos Avanzados y Curiosidades 💡

La manipulación del DOM puede ser optimizada y extendida con técnicas más avanzadas.

### 7.1. Delegación de Eventos

Consiste en adjuntar un único manejador de eventos a un elemento padre en lugar de a cada uno de sus elementos hijos. Cuando un evento se propaga (burbujea) desde un hijo hasta el padre, el manejador del padre puede identificar qué hijo fue el origen del evento y reaccionar en consecuencia. Esto es muy eficiente para listas grandes o elementos creados dinámicamente.

#### Ventajas:

- Mejora el rendimiento (menos manejadores).
- Simplifica el código para elementos dinámicos.

### 7.2. Fragmentos de Documento (DocumentFragment)

`DocumentFragment` es un nodo DOM ligero que sirve como un contenedor temporal para otros nodos. Puedes construir una estructura de DOM compleja dentro de un `DocumentFragment` y luego insertarla en el DOM real con una única operación `appendChild()`.

#### Ventajas:

- Reduce el número de operaciones de manipulación directa del DOM, lo que minimiza el "reflow" y "repaint" del navegador.
- Mejora el rendimiento, especialmente al añadir muchos elementos.

### 7.3. Reflow (Layout) y Repaint (Paint)

Cuando el DOM se modifica o los estilos CSS cambian, el navegador realiza procesos de "reflow" (recalcular la geometría y posición de todos los elementos afectados) y "repaint" (redibujar los píxeles en la pantalla). Estas operaciones son costosas en términos de rendimiento.

#### Curiosidad:

- Cambiar propiedades como **width**, **height**, **left**, **top**, **display** o añadir/eliminar elementos suele causar un **reflow** y un **repaint**.
- Cambiar solo propiedades de color o visibilidad (**opacity**, **background-color**, **visibility**) a menudo solo causa un **repaint**, que es menos costoso.
- Técnicas como el uso de **DocumentFragment** y la delegación de eventos ayudan a minimizar estas operaciones y mejorar la fluidez de las interfaces de usuario.

## 8. Ejercicios Prácticos Básicos

Para consolidar la comprensión de la manipulación del DOM, se proponen los siguientes ejercicios:

### Ejercicio 1: Contador Interactivo

Cree una página HTML con:

1. Un encabezado `<h1>` con el texto inicial "Contador: 0".
2. Un botón con el texto "Incrementar".
3. Al hacer clic en el botón, el número en el `<h1>` debe aumentar en 1.

### Ejercicio 2: Lista de Tareas Dinámica

Cree una página HTML con:

1. Un campo de entrada de texto (`<input type="text">`).
2. Un botón con el texto "Añadir Tarea".
3. Una lista desordenada vacía (`<ul>`) con el ID "listaTareas".
4. Al escribir una tarea en el input y hacer clic en el botón, se debe añadir un nuevo `<li>` con el texto de la tarea a la "listaTareas". El campo de entrada debe limpiarse después de añadir la tarea.

### Ejercicio 3: Alternar Visibilidad

Cree una página HTML con:

1. Un `<div>` con algún contenido de texto y un estilo inicial (ej., un borde, un color de fondo).
2. Un botón con el texto "Alternar Contenido".
3. Al hacer clic en el botón, el `<div>` debe alternar su visibilidad (mostrar/ocultar) utilizando las propiedades de `classList` y CSS (ej., una clase `oculto { display: none; }`).

## 9. Reflexión y Análisis

- Considere cómo la manipulación del DOM permite ir más allá de las páginas web estáticas. ¿Qué tipo de interacciones o aplicaciones no serían posibles sin ella?
- Reflexione sobre la importancia de la eficiencia en la manipulación del DOM, especialmente en aplicaciones web complejas. ¿Por qué es crucial minimizar los "reflows" y "repaints"?
- Analice la diferencia entre `textContent` e `innerHTML` en términos de seguridad y funcionalidad. ¿Cuándo usaría uno u otro?

## 10. Referencias

Para una exploración más profunda y detallada de los conceptos presentados, se recomienda consultar la siguiente documentación oficial y recursos educativos:

- **MDN Web Docs (Mozilla Developer Network):**
  - [Introducción al DOM](#)
  - [Manipulación del DOM](#)
  - [Eventos del DOM](#)
- **W3Schools:**
  - [HTML DOM Introduction](#)
  - [HTML DOM Events](#)
- **Can I use... (Para compatibilidad de características del DOM entre navegadores):**
  - [Can I use... DOM](#)