

Unidad 7: DOM y Eventos en JavaScript

# **Introducción al DOM**

## **¿Qué es el DOM?**

El **Modelo de Objetos del Documento (DOM)** es una interfaz de programación para documentos HTML y XML. Representa la estructura de un documento como un árbol de nodos, donde cada nodo corresponde a una parte del documento: elementos, atributos, texto o comentarios. El DOM permite a los lenguajes de programación, como JavaScript, interactuar y manipular la estructura, el estilo y el contenido del documento de manera dinámica.

### **Estructura del DOM como un árbol de nodos**

Cuando un documento HTML se carga en el navegador, se transforma en el DOM, un modelo que organiza el documento en una estructura jerárquica similar a un árbol. En este árbol, cada elemento HTML, atributo o fragmento de texto se convierte en un nodo del árbol.

* **Nodo raíz**: El nodo más alto en la jerarquía del DOM es el nodo Document, que representa todo el documento.
* **Nodos elementos**: Cada etiqueta HTML se convierte en un nodo de tipo "Elemento". Estos nodos pueden contener otros nodos, formando así la estructura jerárquica del árbol.
* **Nodos de texto**: Los textos dentro de las etiquetas HTML se convierten en nodos de tipo "Texto".
* **Nodos de atributo**: Los atributos de las etiquetas HTML, como class o id, se representan como nodos de tipo "Atributo".
* **Nodos de comentario**: Los comentarios en el código HTML se convierten en nodos de tipo "Comentario".

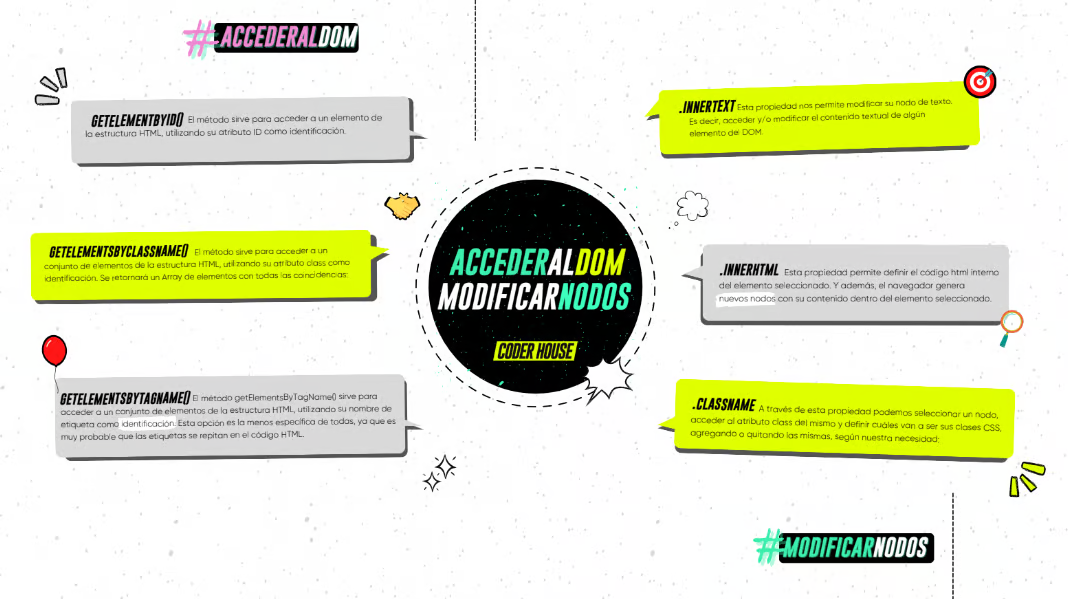
### **Importancia del DOM en la manipulación dinámica de HTML y CSS**

El DOM es crucial para la manipulación dinámica de páginas web porque permite que JavaScript acceda y modifique el contenido, la estructura y el estilo de un documento mientras se está ejecutando en el navegador. Esto es esencial para crear páginas web interactivas y dinámicas.

**Algunas capacidades del DOM incluyen:**

* **Modificar contenido**: Con el DOM, se puede cambiar el texto, los atributos o las propiedades de cualquier elemento HTML. Por ejemplo, se puede actualizar un encabezado de una página sin necesidad de recargar toda la página.
* **Agregar o eliminar elementos**: Se pueden crear nuevos elementos HTML y agregarlos al documento, o eliminar elementos existentes, permitiendo que la página web responda a las acciones del usuario en tiempo real.
* **Estilizar elementos**: Mediante el DOM, también es posible modificar el CSS de la página, cambiando estilos como colores, tamaños o posicionamientos de elementos dinámicamente.
* **Manejo de eventos**: El DOM permite escuchar y reaccionar a eventos del usuario, como clics, teclas presionadas o movimientos del mouse, lo que es fundamental para la interactividad en las páginas web.

## **Visualización del DOM**



## **Estructura del DOM**

El Modelo de Objetos del Documento (DOM) es una representación estructurada de un documento HTML o XML en forma de un árbol de nodos. Cada elemento del documento se convierte en un nodo en este árbol, formando una jerarquía que refleja la estructura del documento original. Esta jerarquía es fundamental para permitir que los lenguajes de programación, como JavaScript, accedan y manipulen los contenidos y la estructura del documento de manera dinámica.

### **Estructura del Árbol del DOM**

* **Nodo raíz**: El nodo superior del árbol es el nodo Document, que representa todo el documento. Este nodo contiene todos los demás nodos del documento, como los nodos html, head, y body.
* **Nodos de elementos**: Cada etiqueta HTML en el documento se convierte en un nodo de tipo "Elemento". Estos nodos pueden tener otros nodos como hijos, representando las etiquetas anidadas dentro de ellas. Por ejemplo, un nodo body puede contener nodos div, header, footer, etc.
* **Nodos de texto**: Los textos contenidos dentro de las etiquetas HTML se representan como nodos de tipo "Texto". Estos nodos son hijos de los nodos de elementos que los contienen.
* **Nodos de atributos**: Los atributos de las etiquetas HTML, como id, class, o src, se representan como nodos de tipo "Atributo" y están asociados a los nodos de elementos correspondientes.
* **Nodos de comentario**: Los comentarios en el documento HTML se convierten en nodos de tipo "Comentario".

### **Jerarquía del DOM**

La estructura del DOM es jerárquica, similar a un árbol genealógico. Comienza con un único nodo raíz, del cual derivan otros nodos en distintos niveles. Cada nodo puede tener múltiples nodos hijos, pero solo un nodo padre, lo que crea una relación clara de contención entre los elementos del documento.

Comprender cómo se estructura y organiza el DOM es esencial para manipular documentos HTML de manera efectiva. La jerarquía del DOM permite a los desarrolladores acceder a cualquier parte del documento, modificarlo, agregar nuevos elementos, eliminar los existentes y responder a las acciones del usuario, lo que es crucial para el desarrollo de aplicaciones web interactivas.

El DOM no solo refleja la estructura de un documento, sino que también proporciona la base sobre la cual se pueden aplicar estilos y scripts para mejorar la experiencia del usuario, facilitando la interacción dinámica con el contenido de la página.

# **Accediendo al DOM**

## **Métodos de Acceso**

### **1. getElementById()**

El método getElementById() se utiliza para acceder a un único elemento del DOM que tiene un atributo id específico. Es uno de los métodos más comunes y eficaces para obtener una referencia directa a un elemento HTML particular, dado que el atributo id es único dentro de un documento HTML.

**Sintaxis:**

| let elemento = document.getElementById('id-del-elemento'); |
| --- |

**Ejemplo:**

| // HTML de referencia  <div id="app">  <p id="parrafo1">Hola Mundo</p>  </div>  // JavaScript  let div = document.getElementById("app");  let parrafo = document.getElementById("parrafo1");  console.log(div.innerHTML); // Muestra el contenido HTML dentro del div  console.log(parrafo.innerHTML); // Muestra "Hola Mundo" |
| --- |

Este método es ideal cuando se sabe con certeza que el elemento tiene un id único y se necesita acceder a él de manera directa.

### **2. getElementsByClassName()**

El método getElementsByClassName() se usa para obtener todos los elementos que tienen una clase específica. Este método retorna una colección de nodos (HTMLCollection) que contiene todos los elementos que coinciden con la clase proporcionada.

**Sintaxis:**

| let elementos = document.getElementsByClassName('nombre-de-clase'); |
| --- |

**Ejemplo:**

| // HTML de referencia  <ul>  <li class="paises">AR</li><li class="paises">CL</li><li class="paises">UY</li>  </ul>  // JavaScript  let paises = document.getElementsByClassName("paises");  console.log(paises[0].innerHTML); // Muestra "AR"  console.log(paises[1].innerHTML); // Muestra "CL"  console.log(paises[2].innerHTML); // Muestra "UY" |
| --- |

Este método es útil cuando se necesita trabajar con múltiples elementos que comparten una misma clase, como cuando se aplican estilos o se realizan manipulaciones a un grupo de elementos.

### **3. getElementsByTagName()**

El método getElementsByTagName() permite acceder a todos los elementos del DOM que tienen un nombre de etiqueta específico. Al igual que getElementsByClassName(), este método retorna una colección de nodos que coinciden con la etiqueta especificada.

**Sintaxis:**

| let elementos = document.getElementsByTagName('nombre-de-etiqueta'); |
| --- |

**Ejemplo:**

| // HTML de referencia  <div>  <div>CONTENEDOR 2</div><div>CONTENEDOR 3</div>  </div>  // JavaScript  let contenedores = document.getElementsByTagName("div");  console.log(contenedores[0].innerHTML); // Muestra "CONTENEDOR 2"  console.log(contenedores[1].innerHTML); // Muestra "CONTENEDOR 3" |
| --- |

Este método es conveniente cuando se desea acceder a todos los elementos de un tipo particular, como todos los <div>, <p>, o <span> en un documento.

### **Conclusión**

Los métodos getElementById(), getElementsByClassName(), y getElementsByTagName() son fundamentales para la manipulación del DOM en JavaScript. Cada uno tiene su uso específico: getElementById() es excelente para acceder a un solo elemento, mientras que getElementsByClassName() y getElementsByTagName() permiten trabajar con múltiples elementos que comparten una clase o etiqueta común, respectivamente. Estos métodos son esenciales para interactuar con el HTML desde JavaScript, permitiendo que las páginas web sean dinámicas e interactivas.

## **Acceso Moderno al DOM**

El método querySelector() es una herramienta poderosa en JavaScript que permite seleccionar el primer elemento del DOM que coincida con uno o más selectores CSS especificados. Este método es especialmente útil cuando se necesita acceder rápidamente a un solo elemento que cumpla con ciertos criterios, utilizando la misma sintaxis que se usa en CSS.

**Sintaxis:**

| let elemento = document.querySelector('selector-css'); |
| --- |

* **selector-css**: Puede ser cualquier selector válido en CSS, como un id, una clase, un atributo, o una combinación de estos.

**Ejemplo:**

| let encabezado = document.querySelector('h1');  let parrafo = document.querySelector('.texto-destacado');  let elementoEnlace = document.querySelector('a[href="#seccion"]'); |
| --- |

En este ejemplo:

* querySelector('h1') selecciona el primer elemento <h1> en el DOM.
* querySelector('.texto-destacado') selecciona el primer elemento con la clase texto-destacado.
* querySelector('a[href="#seccion"]') selecciona el primer enlace con el atributo href que apunta a #seccion.

### **querySelectorAll()**

El método querySelectorAll() es similar a querySelector(), pero en lugar de devolver solo el primer elemento coincidente, devuelve una **lista estática** de todos los elementos que coinciden con los selectores CSS especificados. La lista devuelta es de tipo NodeList, que puede ser iterada con métodos como forEach.

**Sintaxis:**

| let elementos = document.querySelectorAll('selector-css'); |
| --- |

**Ejemplo:**

| let parrafos = document.querySelectorAll('p');  let itemsMenu = document.querySelectorAll('ul.menu li'); |
| --- |

En este ejemplo:

* querySelectorAll('p') selecciona todos los elementos <p> en el DOM y los devuelve en una NodeList.
* querySelectorAll('ul.menu li') selecciona todos los elementos <li> que son hijos directos de un elemento <ul> con la clase menu.

### **Ventajas de querySelector() y querySelectorAll()**

1. **Flexibilidad**: Permiten el uso de selectores CSS complejos para acceder a elementos del DOM, lo que brinda gran precisión en la manipulación de los elementos.
2. **Consistencia**: Ambos métodos utilizan la misma sintaxis que CSS, lo que facilita su uso, especialmente para aquellos que ya están familiarizados con las hojas de estilo.
3. **Universalidad**: A diferencia de métodos más antiguos como getElementById() o getElementsByClassName(), estos métodos pueden combinar múltiples criterios de selección en una sola llamada.

# **Modificación de Nodos**

## **Modificar el DOM**

Manipular el DOM es una de las tareas fundamentales en JavaScript para crear experiencias web dinámicas. A continuación, se describen algunos de los métodos más comunes para modificar, añadir y eliminar nodos en el DOM.

### **Modificar nodos existentes**

* **innerHTML**: Este método permite leer o modificar el contenido HTML interno de un elemento. Se utiliza para actualizar todo el contenido de un nodo de una sola vez.

**Ejemplo:**

| let contenedor = document.getElementById('contenedor');  contenedor.innerHTML = '<h2>Nuevo título</h2><p>Nuevo párrafo</p>'; |
| --- |

* **innerText / textContent**: Permiten modificar el texto dentro de un nodo. innerText considera el estilo y el renderizado actual, mientras que textContent devuelve el contenido textual sin tener en cuenta el estilo.

**Ejemplo:**

| let titulo = document.getElementById('titulo');  titulo.innerText = 'Texto actualizado'; |
| --- |

* **setAttribute()**: Se usa para modificar o establecer un nuevo atributo en un elemento.

**Ejemplo:**

| let enlace = document.querySelector('a');  enlace.setAttribute('href', '<<https://www.example.com>>'); |
| --- |

### **Añadir nodos**

* **createElement()**: Este método permite crear un nuevo elemento en el DOM.

**Ejemplo:**

| let nuevoDiv = document.createElement('div');  nuevoDiv.innerHTML = '<p>Este es un nuevo div</p>';  document.body.appendChild(nuevoDiv); |
| --- |

* **appendChild()**: Añade un nodo como el último hijo de un nodo padre.

**Ejemplo:**

| let lista = document.getElementById('lista');  let nuevoItem = document.createElement('li');  nuevoItem.textContent = 'Nuevo ítem';  lista.appendChild(nuevoItem); |
| --- |

* **insertBefore()**: Inserta un nodo antes de un nodo hijo específico.

**Ejemplo:**

| let primerItem = lista.firstChild;  lista.insertBefore(nuevoItem, primerItem); |
| --- |

### **Eliminar nodos**

* **removeChild()**: Este método elimina un nodo hijo de un elemento.

**Ejemplo:**

| let itemAEliminar = document.getElementById('item');  lista.removeChild(itemAEliminar); |
| --- |

* **remove()**: Elimina directamente el nodo seleccionado.

**Ejemplo:**

| let elemento = document.getElementById('elemento');  elemento.remove(); |
| --- |

**Conclusión**

Dominar la manipulación del DOM es esencial para desarrollar aplicaciones web dinámicas y responsivas. Con métodos como innerHTML, createElement(), appendChild(), y remove(), se puede modificar la estructura y el contenido del DOM, lo que permite a los desarrolladores crear y modificar contenido en tiempo real, mejorar la interactividad y proporcionar una mejor experiencia de usuario.

# **Plantillas Literales e Interactividad**

## **Uso de Plantillas Literales**

Las plantillas literales, introducidas en ES6, son una característica poderosa en JavaScript que permite la creación de cadenas de texto más dinámicas y flexibles. Utilizando plantillas literales, es posible inyectar variables, expresiones complejas e incluso estructuras HTML directamente en el DOM, simplificando así la manipulación del contenido dinámico en una página web.

### **¿Qué son las plantillas literales?**

Las plantillas literales se definen utilizando comillas invertidas (```) en lugar de las comillas simples o dobles tradicionales. Esto permite incluir variables y expresiones dentro de la cadena utilizando la sintaxis ${expresion}. Además, las plantillas literales permiten la creación de cadenas de múltiples líneas sin necesidad de concatenación o caracteres especiales.

### **Inyección de Variables y Expresiones en el DOM**

Uno de los usos más comunes de las plantillas literales es la generación dinámica de contenido HTML que se puede inyectar en el DOM. A través del método innerHTML, es posible utilizar plantillas literales para construir y actualizar la estructura HTML de un elemento directamente desde JavaScript.

**Ejemplo básico:**

| const nombre = "Juan";  const edad = 30;  const contenedor = document.getElementById('info');  contenedor.innerHTML = `  <h2>Información del Usuario</h2>  <p>Nombre: ${nombre}</p>  <p>Edad: ${edad}</p>  `; |
| --- |

En este ejemplo, se utiliza una plantilla literal para crear un bloque de HTML que incluye variables JavaScript (nombre y edad). Este bloque de HTML se inyecta en un contenedor del DOM con el id info. La ventaja de esta técnica es que permite combinar fácilmente texto estático con valores dinámicos.

### **Incluir Expresiones Complejas**

Además de inyectar simples variables, las plantillas literales permiten incluir expresiones más complejas directamente en la cadena. Esto es útil cuando se necesita realizar cálculos o lógica antes de mostrar el contenido en el DOM.

**Ejemplo con expresiones:**

| const productos = [  { nombre: "Arroz", precio: 125 },  { nombre: "Fideos", precio: 70 },  { nombre: "Pan", precio: 50 }  ];  const contenedor = document.getElementById('productos');  contenedor.innerHTML = `  <h2>Lista de Productos</h2>  <ul>  ${productos.map(producto => `<li>${producto.nombre} - $${producto.precio}</li>`).join('')}  </ul>  `; |
| --- |

En este caso, se utiliza una plantilla literal para construir una lista HTML (<ul>) que muestra cada producto en un <li>. La expresión ${productos.map(...).join('')} recorre el array productos y genera un string HTML para cada elemento, que luego se inyecta en el DOM.

**Conclusión**

Las plantillas literales en JavaScript son una herramienta fundamental para trabajar con contenido dinámico en el DOM. Facilitan la combinación de HTML y JavaScript, permitiendo la inyección de variables y expresiones directamente en el código HTML de manera limpia y eficiente. Al dominar las plantillas literales, los desarrolladores pueden crear interfaces web más dinámicas y mantener un código más legible y fácil de mantener.

# **Enlaces y Rutas**

## **Enlaces en HTML**

En HTML, los enlaces son fundamentales para la navegación web, permitiendo a los usuarios moverse entre páginas y secciones. A continuación, se describe cómo crear y utilizar enlaces relativos, absolutos e internos, junto con ejemplos prácticos y su relación con la estructura del sitio web.

### **1. Enlaces Relativos**

Los enlaces relativos apuntan a recursos ubicados en la misma estructura o servidor que la página actual. Son útiles cuando se enlaza a otros documentos o recursos dentro del mismo sitio web. Su principal ventaja es la flexibilidad al mover el sitio a diferentes dominios o servidores, ya que los enlaces siguen funcionando sin necesidad de modificaciones.

**Ejemplo:**

| <a href="pagina2.html">Ir a la página 2</a> |
| --- |

En este ejemplo, pagina2.html está en el mismo directorio que la página actual. El enlace llevará al usuario a pagina2.html cuando se haga clic.

* **Subdirectorios:**

| <a href="carpeta/pagina3.html">Ir a la página 3</a> |
| --- |

Este enlace apunta a pagina3.html dentro de la carpeta carpeta.

* **Directorios padres:**

| <a href="../pagina4.html">Ir a la página 4</a |
| --- |

Este enlace navega un nivel hacia arriba en la estructura de directorios antes de buscar pagina4.html.

### **2. Enlaces Absolutos**

Los enlaces absolutos especifican la ruta completa a un recurso, incluyendo el protocolo (como http:// o https://), el dominio, y la ruta completa del archivo. Estos enlaces son esenciales para apuntar a recursos externos o cuando se quiere garantizar que el enlace funcione independientemente de la ubicación del documento actual.

**Ejemplo:**

| <a href="<https://www.ejemplo.com/pagina2.html>">Visitar Ejemplo</a> |
| --- |

Aquí, el enlace lleva al usuario a pagina2.html en el dominio www.ejemplo.com, sin importar desde dónde se acceda al enlace.

### **3. Enlaces Internos**

Los enlaces internos (o anclas) permiten navegar dentro de la misma página o a secciones específicas de otra página. Se utilizan comúnmente para facilitar la navegación en documentos largos.

* **Anclas en la misma página:**

| <a href="#seccion1">Ir a la Sección 1</a>  ...  <h2 id="seccion1">Sección 1</h2> |
| --- |

En este ejemplo, el enlace lleva al usuario directamente a la sección etiquetada con id="seccion1" dentro de la misma página.

* **Enlaces internos a otras páginas:**

| <a href="pagina2.html#seccion2">Ir a la Sección 2 en Página 2</a> |
| --- |

Este enlace lleva al usuario a seccion2 dentro de pagina2.html.

**Conclusión**

Comprender cómo y cuándo utilizar enlaces relativos, absolutos e internos es clave para estructurar y navegar de manera eficiente en un sitio web. Los enlaces relativos son ideales para recursos dentro del mismo sitio, los absolutos son necesarios para recursos externos, y los internos facilitan la navegación dentro de documentos largos o entre secciones específicas. La correcta implementación de estos enlaces mejora la usabilidad y accesibilidad de cualquier sitio web.