¿Qué es el DOM?

Origen y Función del DOM

El DOM (Document Object Model) fue desarrollado como un estándar para que los navegadores pudieran representar documentos estructurados (como HTML) y ofrecer una forma de interactuar con ellos mediante scripts. Fue introducido por el **World Wide Web Consortium (W3C)** para garantizar la interoperabilidad entre navegadores y herramientas de desarrollo.

En esencia, el DOM convierte un documento HTML en una estructura en forma de árbol, donde cada nodo representa un elemento, atributo o contenido del documento. Este modelo permite a los desarrolladores:

- Acceder a los elementos de la página.
- Modificar su estructura, contenido o estilos.
- Reaccionar a eventos (como clics o desplazamientos).

Cómo Funciona

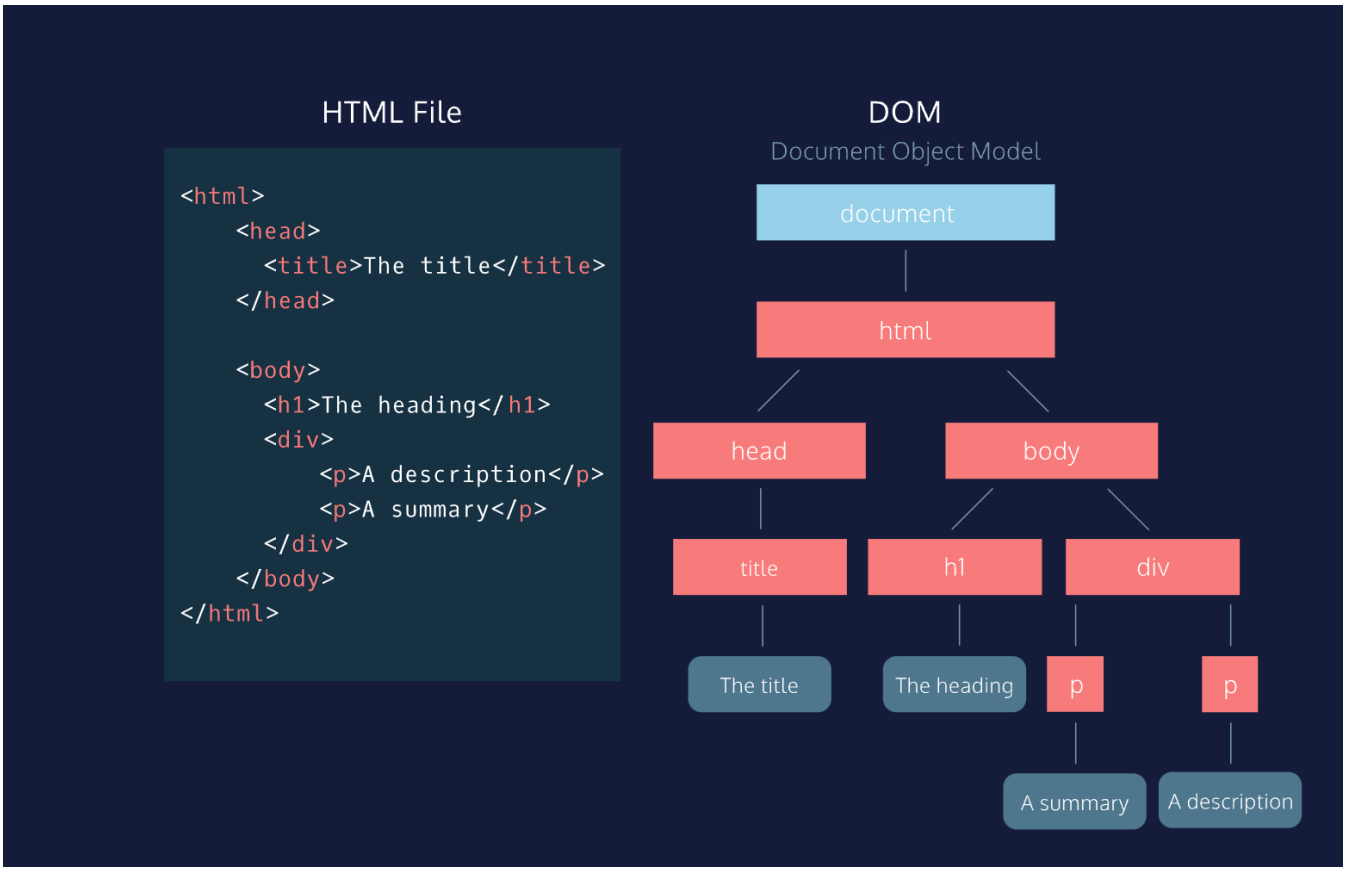
Imagina que el DOM es un mapa jerárquico de tu documento HTML. Por ejemplo, dado este fragmento:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo</title>
  </head>
  <body>
    <h1>Hola, mundo</h1>
    <p>Este es un párrafo.</p>
  </body>
</html>
```

El DOM lo representa como un árbol:

```
- html
  - head
    - title: "Ejemplo"
  - body
    - h1: "Hola, mundo"
    - p: "Este es un párrafo."
```

Cada nodo tiene propiedades y métodos que permiten interactuar con él desde JavaScript.



Manipulación del DOM con JavaScript

Acceder a Elementos del DOM

Para interactuar con un elemento de una página web, primero necesitas "encontrarlo" dentro del árbol del DOM. JavaScript ofrece diferentes métodos para lograrlo, dependiendo de tus necesidades. Vamos a ver algunos de los más comunes:

Usando `getElementById`

El método `getElementById` es uno de los más utilizados debido a su simplicidad y eficiencia. Permite acceder a un elemento directamente a través del atributo `id` que tiene asignado en el HTML.

Por ejemplo:

```
<div id="miDiv">Contenido del div</div>
<script>
  const div = document.getElementById("miDiv");
  console.log(div.textContent); // "Contenido del div"
</script>
```

En este caso, `document.getElementById("miDiv")` busca dentro del documento HTML el elemento cuyo atributo `id` es igual a `miDiv`. Si el elemento existe, devuelve una referencia a él; si no, devuelve `null`.

Ventajas:

- Es rápido porque accede directamente al elemento.
- Ideal para elementos únicos en la página.

> **Nota**
>
> Asegúrate de que el atributo `id` sea único dentro del documento para evitar problemas de selección.

Usando `querySelector`

El método `querySelector` selecciona el primer elemento que coincide con un selector CSS que proporciones. Esto lo hace extremadamente versátil, ya que puedes usar cualquier selector válido de CSS (como clases, atributos o combinaciones de ellos).

Por ejemplo:

```
<p class="texto">Hola</p>
<script>
  const parrafo = document.querySelector(".texto");
  console.log(parrafo.textContent); // "Hola"
</script>
```

En este caso, el selector `.texto` busca el primer elemento con la clase `texto`. Puedes usar otros selectores, como `#id`, `div > p`, `[atributo="valor"]`, entre otros.

Ventajas:

- Permite usar cualquier selector CSS.
- Flexible para seleccionar elementos anidados o específicos.

> **Nota**
>
> Si hay varios elementos que coinciden con el selector, solo devuelve el primero. Para seleccionar múltiples elementos, utiliza `querySelectorAll`.

Usando `querySelectorAll`

El método `querySelectorAll` permite seleccionar todos los elementos que coinciden con un selector CSS. Devuelve una **NodeList**, que es similar a un array, pero no tiene todos los métodos de un array tradicional. Puedes recorrer esta lista usando un bucle.

Por ejemplo:

```
<ul>
  <li class="item">Item 1</li>
  <li class="item">Item 2</li>
  <li class="item">Item 3</li>
</ul>
<script>
  const items = document.querySelectorAll(".item");
  items.forEach((item, index) => {
    console.log(`Elemento ${index + 1}: ${item.textContent}`);
  });
</script>
```

En este caso, `querySelectorAll(".item")` selecciona todos los elementos con la clase `item`. Luego, el método `forEach` se utiliza para recorrer cada elemento y mostrar su contenido en la consola.

Ventajas:

- Útil cuando necesitas trabajar con múltiples elementos al mismo tiempo.
- Compatible con cualquier selector CSS.

> **Atención**
>
> Aunque la **NodeList** no es un array completo, puedes convertirla a un array usando `Array.from()` si necesitas usar métodos avanzados de arrays.

Crear y Agregar Elementos

La creación dinámica de elementos HTML en una página web es una de las tareas más comunes al manipular el DOM. Esto se realiza con el método `document.createElement`, que permite generar un nuevo elemento del tipo que especifiques, y luego agregarlo al DOM mediante métodos como `appendChild`. Vamos a ver cómo usarlos en unos sencillos pasos:

Paso 1: Crear un Nuevo Elemento

El primer paso es usar el método `document.createElement`. Este método toma como argumento el nombre de la etiqueta HTML que deseas crear y devuelve un elemento vacío.

Ejemplo: Crear un párrafo

```
const nuevoElemento = document.createElement("p");
```

En este caso, hemos creado un elemento `<p>` (párrafo) que todavía no contiene ningún contenido ni está visible en la página.

Paso 2: Configurar el Nuevo Elemento

Una vez creado el elemento, puedes configurarlo añadiéndole contenido, clases, estilos, o atributos.

Ejemplo: Agregar texto al elemento

```
nuevoElemento.textContent = "Este es un párrafo creado dinámicamente.";
```

La propiedad `textContent` define el texto que se mostrará dentro del elemento. Es una forma segura y recomendada de agregar texto, ya que no interpreta HTML.

También puedes personalizar el elemento con otros atributos, como clases o estilos:

```
nuevoElemento.className = "mi-clase";
nuevoElemento.style.color = "blue";
```

Paso 3: Agregar el Elemento al DOM

El nuevo elemento no será visible en la página hasta que lo agregues al DOM. Esto se hace utilizando métodos como `appendChild`, que añade el elemento como hijo del nodo especificado.

Ejemplo: Agregar un párrafo a un contenedor existente

```
<div id="contenedor"></div>
<script>
  const nuevoElemento = document.createElement("p");
  nuevoElemento.textContent = "Este es un párrafo creado dinámicamente.";

  const contenedor = document.getElementById("contenedor");
  contenedor.appendChild(nuevoElemento);
</script>
```

En este ejemplo:

1. Creamos un nuevo elemento `<p>`.
2. Agregamos texto al elemento.
3. Seleccionamos el elemento con el `id` `contenedor`.
4. Usamos `appendChild` para insertar el nuevo párrafo como hijo de este contenedor.

> ⚠ **Consideraciones importantes:**
>
> 1. **Crear el elemento no lo hace visible.** Hasta que lo añadas al DOM con métodos como `appendChild`, el elemento solo existirá en la memoria del navegador.
> 2. **Organización del código.** Si estás creando y manipulando muchos elementos, considera usar funciones reutilizables para evitar redundancia.

Con estos pasos, puedes generar dinámicamente contenido HTML y agregarlo al DOM, mejorando la interacción y experiencia de los usuarios en tus páginas web.

Modificar el Contenido:

En el DOM, el contenido de los elementos puede modificarse de forma dinámica utilizando las propiedades `textContent` como ya hemos visto e `innerHTML`. Ambas son herramientas esenciales para actualizar el contenido de una página web en respuesta a acciones del usuario o cambios en los datos.

Propiedad `textContent`

Como ya hemos visto durante la creación de un elemento, la propiedad `textContent` se utiliza para modificar o leer el texto dentro de un elemento. Esto incluye todo el texto contenido en el nodo, ignorando etiquetas HTML o scripts que puedan estar presentes.

Ejemplo: Cambiar el texto de un encabezado

```
<h1 id="titulo">Título Original</h1>
<script>
  const titulo = document.getElementById("titulo");
  titulo.textContent = "Título Modificado";
</script>
```

Ventajas de `textContent`:

- Más seguro que `innerHTML`, ya que no interpreta ni ejecuta código HTML o JavaScript.
- Ideal para trabajar exclusivamente con texto plano.

Propiedad `innerHTML`

La propiedad `innerHTML` permite modificar o leer el contenido HTML de un elemento, lo que incluye tanto texto como etiquetas HTML. Es útil cuando necesitas agregar o cambiar contenido estructurado.

Ejemplo: Insertar HTML en un contenedor

```
<div id="contenedor">Texto inicial</div>
<script>
  const contenedor = document.getElementById("contenedor");
  contenedor.innerHTML = "<p>Este es un <strong>nuevo</strong> contenido.</p>";
</script>
```

Consideraciones de `innerHTML`:

- Permite insertar contenido estructurado con etiquetas HTML.
- Útil para generar fragmentos de HTML dinámicamente.
- La propiedad `innerHTML` reemplaza todo el contenido del contenedor con el HTML proporcionado.

Peligro:

Usar `innerHTML` puede ser un riesgo de seguridad si el contenido que se inserta proviene de fuentes externas, ya que podría dar lugar a ataques de **Cross-Site Scripting (XSS)**. Para evitarlo, valida siempre los datos antes de usarlos.

Comparación entre `textContent` e `innerHTML`

Propiedad	Uso principal	Seguridad	Manipula etiquetas HTML
`textContent`	Texto plano	Seguro	No
`innerHTML`	Texto + contenido estructurado	Requiere validación cuidadosa	Sí

Ejemplo Comparativo

```
  <div id="contenedor"></div>
  <div id="contenedor2"></div>
<script>
  const contenedor = document.getElementById("contenedor");

  // Usando textContent
  contenedor.textContent = "<strong>Texto con etiquetas</strong>";
  console.log(contenedor.innerHTML);

  // Usando innerHTML
  contenedor2.innerHTML = "<strong>Texto con etiquetas</strong>";
  console.log(contenedor2.innerHTML);
</script>
```

En el ejemplo, `textContent` trata el texto literalmente, mientras que `innerHTML` interpreta y renderiza las etiquetas HTML.

Modificar Estilos CSS

En el DOM, puedes cambiar el estilo de los elementos de manera dinámica utilizando la propiedad `style` o manipulando las clases del elemento con `classList`. Esto es particularmente útil para mejorar la interactividad de las páginas web y personalizar el diseño en tiempo real.

Cambiar Estilos con `style`

La propiedad `style` permite acceder y modificar los estilos CSS en línea de un elemento. Los nombres de las propiedades CSS en esta interfaz siguen la convención de camelCase, es decir, los guiones se eliminan y las palabras comienzan con mayúscula después del primer término.

Ejemplo: Cambiar el color y el tamaño del texto

```
<p id="parrafo">Texto con estilo.</p>
<script>
  const parrafo = document.getElementById("parrafo");
  parrafo.style.color = "blue"; // Cambia el color del texto a azul
  parrafo.style.fontSize = "20px"; // Cambia el tamaño de la fuente a 20px
</script>
```

> **Info**
>
> Los estilos aplicados con `style` se añaden como propiedades en línea, lo que puede sobrescribir estilos definidos en hojas de estilo externas.

Modificar Estilos con Clases (`classList`)

Cuando necesitas aplicar múltiples estilos o estilos complejos, es preferible usar clases CSS predefinidas y manipularlas con el método `classList`.

Ejemplo: Agregar y remover una clase

```
<p id="parrafo" class="original">Texto con clase.</p>
<script>
  const parrafo = document.getElementById("parrafo");

  // Agrega una nueva clase
  parrafo.classList.add("nueva-clase");

  // Remueve la clase original
  parrafo.classList.remove("original");
```

```
  // Verifica si tiene una clase específica
  console.log(parrafo.classList.contains("nueva-clase")); // true
</script>
```

Ventajas:

- Los estilos se mantienen centralizados en las hojas de estilo, lo que mejora la mantenibilidad.
- Es más limpio y eficiente para aplicar varios estilos simultáneamente.

Ejemplo Comparativo

```
<style>
  .azul {
    color: blue;
    font-size: 18px;
  }
</style>
<p id="parrafo">Texto con estilo.</p>
<script>
  const parrafo = document.getElementById("parrafo");

  // Usando la propiedad style
  parrafo.style.color = "blue";
  parrafo.style.fontSize = "18px";

  // Usando classList
  parrafo.classList.add("azul");
</script>
```

Ambas opciones logran un efecto visual similar, pero `classList` permite mantener los estilos en la hoja de estilo, promoviendo un diseño más limpio y escalable.

Para estilos dinámicos que dependen de acciones del usuario (por ejemplo, hacer que un botón cambie de color al hacer clic), considera combinar ambas técnicas. Usa `style` para cambios específicos y temporales, y `classList` para cambios más complejos y reutilizables.

Ejemplo: Combinación de `style` y `classList`

```
<button id="boton">Haz clic aquí</button>
<script>
  const boton = document.getElementById("boton");

  boton.addEventListener("click", () => {
    boton.style.backgroundColor = "yellow"; // Cambio inmediato y específico
    boton.classList.toggle("activo"); // Cambia entre clases dinámicamente
  });
</script>
```

Eliminar Elementos

En ocasiones, es necesario eliminar elementos del DOM para ajustar dinámicamente la estructura de la página. Esto puede lograrse de diferentes maneras en JavaScript, dependiendo de si quieres eliminar el elemento directamente o manejar su relación con su elemento padre.

Usar el Método `remove`

El método más directo para eliminar un elemento es usar `remove()`. Este método elimina el elemento seleccionado del DOM por completo.

Ejemplo: Eliminar un elemento con `remove`

```
<p id="parrafo">Este párrafo será eliminado.</p>
<script>
  // Seleccionamos el elemento por su ID
  const parrafo = document.getElementById("parrafo");

  // Eliminamos el elemento del DOM
  parrafo.remove();
</script>
```

Este método no requiere manipular el elemento padre, lo que lo hace sencillo y directo para eliminar elementos específicos.

Eliminar Usando el Nodo Padre

Otra manera de eliminar un elemento es acceder a su nodo padre y usar el método `removeChild()`. Aunque esto requiere un paso adicional, puede ser útil si necesitas manipular o guardar referencias del elemento eliminado.

Ejemplo: Usar el nodo padre para eliminar un hijo

```
<div id="contenedor">
  <p id="parrafo">Este párrafo será eliminado.</p>
</div>
<script>
  // Seleccionamos el elemento padre y el hijo
  const contenedor = document.getElementById("contenedor");
  const parrafo = document.getElementById("parrafo");

  // Eliminamos el hijo usando removeChild
  contenedor.removeChild(parrafo);
</script>
```

- Te permite eliminar elementos y mantener acceso a ellos como referencia.
- Útil si necesitas realizar operaciones adicionales con el elemento eliminado.

Diferencias entre `remove` y `removeChild`

Método	¿Cómo funciona?	Ventajas	Desventajas
`remove()`	Elimina el elemento directamente del DOM.	Más sencillo y directo.	No proporciona contexto del nodo padre.
`removeChild()`	Elimina un elemento desde su nodo padre.	Conserva la relación con el padre.	Requiere acceso al nodo padre.

Info

- **Usa** `remove` cuando solo necesites eliminar un elemento específico y no te importe la relación con su nodo padre.
- **Usa** `removeChild` cuando necesites un mayor control sobre el árbol del DOM o planees trabajar con el nodo padre después de eliminar el elemento o incluso con el elemento eliminado para su reinserción posterior.