Tutorial: Manejo del DOM en JavaScript

El **Document Object Model (DOM)** es una interfaz de programación que permite a JavaScript interactuar con HTML y CSS de una página web. A través del DOM, JavaScript puede manipular el contenido, estructura y estilo de una página.

1. ¿Qué es el DOM?

El DOM representa la estructura de un documento HTML o XML en forma de árbol. Cada etiqueta HTML es un nodo en este árbol, y JavaScript puede acceder a ellos y modificarlos en tiempo real.

Por ejemplo, si tenemos el siguiente HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Ejemplo</title>
</head>
<body>
    <h1>Hola Mundo</h1>
    <p>Este es un párrafo.</p>
</body>
</html>
```

El DOM representaría este documento como un árbol de nodos donde `<html>`, `<head>`, `<body>`, `<h1>`, etc., son nodos que JavaScript puede manipular.

2. Selección de elementos del DOM

Para interactuar con los elementos HTML, primero necesitamos seleccionarlos. JavaScript ofrece múltiples maneras de seleccionar elementos.

2.1 `document.getElementById()`

Esta función permite seleccionar un elemento por su atributo `id`.

```
<p id="mi-parrafo">Este es un párrafo.</p>

<script>
    const parrafo = document.getElementById("mi-parrafo");
    console.log(parrafo); // Muestra el párrafo seleccionado en la consola.
</script>
```

2.2 `document.getElementsByClassName()`

Selecciona todos los elementos que compartan la misma clase.

```
<p class="texto">Párrafo 1</p>
<p class="texto">Párrafo 2</p>

<script>
    const parrafos = document.getElementsByClassName("texto");
    console.log(parrafos); // Devuelve un HTMLCollection de los párrafos con clase
"texto".
```

```
</script>
```

2.3 `document.querySelector()` y `document.querySelectorAll()`

`querySelector()` selecciona el primer elemento que coincida con un selector CSS, mientras que `querySelectorAll()` devuelve todos los elementos que coincidan.

```
<p class="texto">Párrafo 1</p>
<p class="texto">Párrafo 2</p>

<script>
    const primerParrafo = document.querySelector(".texto");
    console.log(primerParrafo); // Selecciona el primer párrafo con clase "texto".

    const todosLosParrafos = document.querySelectorAll(".texto");
    console.log(todosLosParrafos); // Selecciona todos los párrafos con clase
"texto".
</script>
```

3. Manipulación de contenido

Una vez que seleccionamos un elemento, podemos cambiar su contenido utilizando `innerHTML`, `innerText` o `textContent`.

3.1 `innerHTML`

Permite cambiar o acceder al contenido HTML de un elemento.

```
<p id="parrafo">Texto inicial.</p>

<script>
    const parrafo = document.getElementById("parrafo");
    parrafo.innerHTML = "<strong>Nuevo texto con HTML</strong>";
</script>
```

3.2 `innerText` y `textContent`

Ambas se usan para cambiar el texto de un elemento, pero `innerText` respeta los estilos CSS mientras que `textContent` no lo hace.

```
<p id="parrafo">Texto inicial.</p>

<script>
    const parrafo = document.getElementById("parrafo");
    parrafo.innerText = "Texto cambiado con innerText.";

    const otroParrafo = document.getElementById("parrafo");
    otroParrafo.textContent = "Texto cambiado con textContent.";
</script>
```

4. Manipulación de atributos

Podemos obtener, establecer y eliminar atributos de un elemento.

4.1 `getAttribute()` y `setAttribute()`

```
<img id="mi-imagen" src="imagen.jpg" alt="Descripción de la imagen">
```

```
<script>
    const imagen = document.getElementById("mi-imagen");

    // Obtener el atributo "src"
    console.log(imagen.getAttribute("src")); // "imagen.jpg"

    // Cambiar el atributo "src"
    imagen.setAttribute("src", "nueva-imagen.jpg");
</script>
```

4.2 `removeAttribute()`

Elimina un atributo de un elemento.

```
<script>
    imagen.removeAttribute("alt");
</script>
```

5. Manipulación de clases y estilos

5.1 `classList`

`classList` permite añadir, quitar o alternar clases de un elemento.

```
<p id="parrafo" class="texto">Párrafo</p>

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

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

    // Quitar una clase
    parrafo.classList.remove("texto");

    // Alternar una clase (la añade si no está, la quita si está)
    parrafo.classList.toggle("resaltado");
</script>
```

5.2 Modificación de estilos

Podemos modificar los estilos CSS directamente mediante la propiedad `style`.

```
<p id="parrafo">Párrafo</p>

<script>
    const parrafo = document.getElementById("parrafo");
    parrafo.style.color = "blue";
    parrafo.style.fontSize = "20px";
</script>
```

6. Creación y eliminación de nodos

6.1 `createElement()` y `appendChild()`

Podemos crear un nuevo elemento y añadirlo al DOM.

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

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

    // Crear un nuevo elemento
    const nuevoParrafo = document.createElement("p");
    nuevoParrafo.textContent = "Este es un párrafo añadido dinámicamente.";

    // Añadir el párrafo al contenedor
    contenedor.appendChild(nuevoParrafo);
</script>
```

6.2 removeChild()

Eliminar un nodo hijo.

```
<script>
    contenedor.removeChild(nuevoParrafo); // Elimina el párrafo recién añadido.
</script>
```

---

7. Eventos del DOM

Los **eventos** permiten que los usuarios interactúen con una página web. Los más comunes son clics, movimientos del mouse, teclas presionadas, entre otros.

7.1 addEventListener()

Para manejar eventos, usamos `addEventListener()`, que permite asignar una función que se ejecutará cuando ocurra un evento específico.

```
<button id="mi-boton">Haz clic aquí</button>

<script>
    const boton = document.getElementById("mi-boton");

    boton.addEventListener("click", function() {
        alert("¡Botón clickeado!");
    });
</script>
```

7.2 Eventos comunes

Algunos de los eventos más usados:

- `click`: Cuando el usuario hace clic en un elemento.
- `mouseover`: Cuando el mouse pasa sobre un elemento.
- `keydown`: Cuando el usuario presiona una tecla.

```
<input type="text" id="mi-input">

<script>
    const input = document.getElementById("mi-input");

    // Escucha el evento cuando el usuario presiona una tecla
    input.addEventListener("keydown", function(event) {
        console.log("Tecla presionada: " + event.key);
    });
```

```
</script>
```

7.3 Remover eventos

Puedes remover un evento usando `removeEventListener()`.

```
<script>
    function mostrarAlerta() {
        alert("¡Botón clickeado!");
    }

    boton.addEventListener("click", mostrarAlerta);

    // Para remover el evento
    boton.removeEventListener("click", mostrarAlerta);
</script>
```

8. Delegación de eventos

La **delegación de eventos** es una técnica eficiente para manejar eventos en elementos que se crean dinámicamente o en un gran número de nodos.

En lugar de asignar un evento a cada elemento, lo asignamos a un contenedor común y aprovechamos la propagación de eventos.

8.1 Ejemplo básico

```
<ul id="mi-lista">
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>

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

    // Delegamos el evento en el contenedor "ul"
    lista.addEventListener("click", function(event) {
        if (event.target.tagName === "LI") {
            console.log("Clic en el elemento: " + event.target.textContent);
        }
    });
</script>
```

En este ejemplo, el evento `click` está asignado al `ul`, pero solo los `li` responden, gracias a la condición `event.target.tagName`.

8.2 Ventajas de la delegación

- Menor uso de memoria.
- Fácil manejo de elementos creados dinámicamente.
- Menos complejidad al gestionar muchos nodos.

9. Manipulación eficiente de múltiples nodos

Cuando necesitas realizar cambios en múltiples nodos, es importante hacerlo de manera eficiente para evitar ralentizaciones.

9.1 `querySelectorAll()` para múltiples elementos

```
<p class="parrafo">Párrafo 1</p>
<p class="parrafo">Párrafo 2</p>

<script>
    const parrafos = document.querySelectorAll(".parrafo");

    parrafos.forEach(parrafo => {
        parrafo.style.color = "blue";
    });
</script>
```

9.2 Evita manipulaciones directas innecesarias

Cada vez que modificas el DOM, el navegador vuelve a dibujar la página, lo cual puede afectar el rendimiento. Para evitarlo, realiza las modificaciones fuera del flujo principal cuando sea posible.

9.3 Uso de `DocumentFragment`

Un `DocumentFragment` es un contenedor ligero que actúa como un fragmento de DOM. Permite hacer múltiples modificaciones sin afectar el DOM real hasta que esté listo.

```
<ul id="lista"></ul>

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

    for (let i = 0; i < 10; i++) {
        const li = document.createElement("li");
        li.textContent = "Elemento " + (i + 1);
        fragmento.appendChild(li);
    }

    lista.appendChild(fragmento); // Inserta todo de una vez en el DOM
</script>
```

10. Navegación entre nodos del DOM

El DOM permite navegar entre elementos relacionados como nodos padres, hijos y hermanos.

10.1 Nodos hijos y padres

- `parentNode`: Accede al nodo padre.
- `children`: Devuelve una colección de nodos hijos.

```
<div id="contenedor">
    <p>Hijo 1</p>
    <p>Hijo 2</p>
</div>

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

    console.log(contenedor.parentNode); // Devuelve el nodo padre del "div".
    console.log(contenedor.children); // Devuelve los hijos del "div".
</script>
```

10.2 Nodos hermanos

- `nextElementSibling`: Accede al siguiente hermano.
- `previousElementSibling`: Accede al hermano anterior.

```
<p id="parrafo1">Párrafo 1</p>
<p id="parrafo2">Párrafo 2</p>

<script>
    const parrafo1 = document.getElementById("parrafo1");
    const parrafo2 = parrafo1.nextElementSibling;

    console.log(parrafo2); // Devuelve el siguiente párrafo.
</script>
```

11. Fragmentos del DOM y rendimiento

Como vimos anteriormente, manipular el DOM directamente en iteraciones o grandes cambios puede afectar el rendimiento. Usar un `DocumentFragment` o hacer operaciones fuera del DOM principal mejora la eficiencia.

11.1 Reemplazar `innerHTML` vs manipulación directa

Aunque `innerHTML` es simple y efectivo, en algunos casos, modificar nodos directamente es más eficiente.

```
const lista = document.getElementById("lista");

// Manipulación directa con Fragment
const fragmento = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
    const li = document.createElement("li");
    li.textContent = "Elemento " + (i + 1);
    fragmento.appendChild(li);
}
lista.appendChild(fragmento);  // Añadimos todo de una vez

// Uso de innerHTML (menos eficiente para muchos cambios)
lista.innerHTML = "";
for (let i = 0; i < 1000; i++) {
    lista.innerHTML += `<li>Elemento ${i + 1}</li>`;
}
```