Apuntes examen javascript

Índice

Tema 7, Eventos **3**
1. Eventos de ratón 3
2. Eventos de teclado 4
3. Eventos de formulario 4
4. Eventos de la ventana o documento 5
5. Eventos táctiles 6
Ejemplos con tablas: 7
1. Eventos de Mouse 7
2. Eventos de Teclado 8
3. Eventos de Formulario 8
4. Eventos de Documento y Ventana 8
5. Eventos de Drag & Drop 9
6. Eventos de Multimedia 9
7. Eventos de Otros Tipos 10
Diferencia entre this y event.target en eventos de JavaScript 10
1. this en los manejadores de eventos 10
Ejemplo con this: 11
2. event.target en los manejadores de eventos 11
Ejemplo con event.target: 11
3. ¿Se pueden usar indistintamente? 12
4. Diferencia clave en eventos delegados 12
Evento Delegado 12
¿Qué es un evento delegado? 12
Cómo funciona la delegación de eventos 12
Ventajas de la delegación de eventos 13
Ejemplo práctico sin delegación 13
Ejemplo práctico con delegación 13
Cuándo usar eventos delegados 14
Resumen 14
Tema 6, Modelo de Objetos del Cliente **15**
Atributos Data (data-*) 15
¿Qué son los Data Attributes? 15
Ventajas: 15
Acceso desde JavaScript 15
Propiedades y Métodos del DOM 16
Propiedades principales de los elementos 16
Métodos principales del DOM 17
Manipulación de clases con classList 18
Selección de Elementos en el DOM 18

Diferencias entre HTMLCollection y NodeList: 19
Ejemplo completo: 19
Tema 5, POO **22**
Tabla de Conceptos de POO en JavaScript 22
Expresiones Regulares (RegEx) 24
1. Componentes básicos de una expresión regular 25
2. Métodos principales en JavaScript 26
test 26
match 26
replace 26
split 26
Validaciones comunes 27
1. Validar un correo electrónico 27
2. Validar un número de teléfono (formato internacional) 27
3. Validar una URL 27
Búsquedas y coincidencias 27
4. Buscar todas las palabras en una cadena 27
5. Buscar números en una cadena 28
6. Extraer etiquetas HTML 28
Reemplazos 28
7. Reemplazar todas las vocales 28
8. Eliminar espacios extra 29
9. Censurar palabras inapropiadas 29
Validaciones avanzadas 29
10. Validar contraseñas fuertes 29
11. Validar fechas (formato dd/mm/yyyy o dd-mm-yyyy) 30
12. Validar códigos postales (España) 30
Métodos para practicar con RegEx en JavaScript 30
Combinación de búsqueda y validación 30
Iterar sobre coincidencias 31
Interacción con JSON 31
1. Conversión de JSON a un objeto JavaScript 31
2. Conversión de un objeto JavaScript a JSON 31
3. Acceso y manipulación de datos JSON 32
4. JSON y APIs 32
Resumen 32
Tema 4, Funciones **33**
Tipos de Funciones en JavaScript 33
Métodos y Propiedades Relacionados con Funciones 34
Ejemplo Práctico con Diferentes Tipos de Funciones 34
Tema 3, Estructura de datos **36**
Tabla de Conceptos de Estructuras de Datos 36
Operaciones Comunes 37
Métodos de Objetos (Diccionarios) 39

Métodos de Sets 40
Métodos de Maps 40
Tema 2, Control de flujo **42**
Estructuras de Control Condicionales 42
Estructuras de Control de Bucles 43
Control de Flujo en Bucles 43
Estructuras de Control con Excepciones 44
Estructuras Asíncronas 44
Ejemplo Completo con Varias Estructuras de Control 44
Métodos Iterativos de Arrays en JavaScript 46
Ejemplo Completo con Métodos Iterativos 48

Tema 7, Eventos

1. Eventos de ratón

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Ratón</title>
</head>
<body>
    <button id="mouseButton">Haz algo con el ratón</button>

    <script>
        const button = document.getElementById('mouseButton');

        button.addEventListener('click', () => alert('¡Hiciste
clic!'));
        button.addEventListener('dblclick', () => alert('¡Doble
clic!'));
        button.addEventListener('mouseover', () =>
button.style.backgroundColor = 'lightblue');
        button.addEventListener('mouseout', () =>
button.style.backgroundColor = '');
```

```
        button.addEventListener('mousedown', () =>
button.textContent = '¡Presionaste el botón!');
        button.addEventListener('mouseup', () => button.textContent
= '¡Soltaste el botón!');
    </script>
</body>
</html>
```

---

2. Eventos de teclado

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Teclado</title>
</head>
<body>
    <input type="text" id="keyboardInput" placeholder="Escribe
algo">

    <script>
        const input = document.getElementById('keyboardInput');

        input.addEventListener('keydown', () => console.log('Tecla
presionada'));
        input.addEventListener('keyup', () => console.log('Tecla
soltada'));
    </script>
</body>
</html>
```

---

3. Eventos de formulario

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Formulario</title>
</head>
<body>
    <form id="myForm">
        <input type="text" id="formInput" placeholder="Escribe
aquí">
        <select id="formSelect">
            <option value="opcion1">Opción 1</option>
            <option value="opcion2">Opción 2</option>
        </select>
        <button type="submit">Enviar</button>
    </form>

    <script>
        const form = document.getElementById('myForm');
        const input = document.getElementById('formInput');
        const select = document.getElementById('formSelect');

        form.addEventListener('submit', (event) => {
            event.preventDefault(); // Evita que se recargue la
página
            alert('Formulario enviado');
        });

        select.addEventListener('change', () =>
alert(`Seleccionaste: ${select.value}`));
        input.addEventListener('input', () => console.log(`Texto:
${input.value}`));
        input.addEventListener('focus', () =>
input.style.backgroundColor = 'lightyellow');
        input.addEventListener('blur', () =>
input.style.backgroundColor = '');
    </script>
</body>
</html>
```

---

4. Eventos de la ventana o documento

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de la Ventana</title>
</head>
<body>
    <h1>Redimensiona o desplaza la ventana</h1>

    <script>
        window.addEventListener('load', () => console.log('Página
cargada'));
        window.addEventListener('resize', () => console.log('Tamaño
de ventana cambiado'));
        window.addEventListener('scroll', () => console.log('Has
desplazado la página'));
    </script>
</body>
</html>
```

---

5. Eventos táctiles

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos Táctiles</title>
</head>
<body>
    <div id="touchArea" style="width: 100%; height: 200px;
background-color: lightgray;">
        Toca aquí
    </div>

    <script>
        const touchArea = document.getElementById('touchArea');
```

```
        touchArea.addEventListener('touchstart', () =>
console.log('Inició el toque'));
        touchArea.addEventListener('touchmove', () =>
console.log('Arrastrando...'));
        touchArea.addEventListener('touchend', () =>
console.log('Toque finalizado'));
    </script>
</body>
</html>
```

Ejemplos con tablas:

¡Claro! Aquí tienes una tabla completa que detalla los tipos de eventos en JavaScript organizados por categoría, junto con su descripción y ejemplos.

---

1. Eventos de Mouse

Evento	Descripción	Ejemplo
`click`	Ocurre cuando el usuario hace clic en un elemento.	`element.addEventListener('click', () => {...});`
`dblclick`	Ocurre cuando el usuario hace doble clic en un elemento.	`element.addEventListener('dblclick', () => {...});`
`mousemove`	Se activa cuando el puntero se mueve dentro del área del elemento.	`element.addEventListener('mousemove', () => {...});`
`mouseover`	Ocurre cuando el puntero entra en el área de un elemento.	`element.addEventListener('mouseover', () => {...});`
`mouseout`	Ocurre cuando el puntero sale del área de un elemento.	`element.addEventListener('mouseout', () => {...});`
`mousedown`	Ocurre cuando el botón del ratón se presiona sobre un elemento.	`element.addEventListener('mousedown', () => {...});`
`mouseup`	Ocurre cuando se suelta el botón del ratón sobre un elemento.	`element.addEventListener('mouseup', () => {...});`

`contextmenu`	Ocurre al abrir el menú contextual (botón derecho del ratón).	`element.addEventListener('contextmenu', () => {...});`

---

2. Eventos de Teclado

Evento	Descripción	Ejemplo
`keydown`	Ocurre cuando una tecla se presiona.	`document.addEventListener('keydown', () => {...});`
`keypress`	**[Deprecado]** Igual que `keydown`, pero no incluye teclas no imprimibles.	`document.addEventListener('keypress', () => {...});`
`keyup`	Ocurre cuando se suelta una tecla.	`document.addEventListener('keyup', () => {...});`

---

3. Eventos de Formulario

Evento	Descripción	Ejemplo
`submit`	Ocurre al enviar un formulario.	`form.addEventListener('submit', (e) => {...});`
`change`	Ocurre cuando un elemento `<input>`, `<select>` o `<textarea>` cambia de valor.	`input.addEventListener('change', () => {...});`
`input`	Ocurre cada vez que se modifica el valor de un elemento de entrada.	`input.addEventListener('input', () => {...});`
`focus`	Ocurre cuando un elemento recibe el foco.	`element.addEventListener('focus', () => {...});`
`blur`	Ocurre cuando un elemento pierde el foco.	`element.addEventListener('blur', () => {...});`
`reset`	Ocurre cuando se reinicia un formulario.	`form.addEventListener('reset', () => {...});`

---

4. Eventos de Documento y Ventana

Evento	Descripción	Ejemplo
`DOMContentLoaded`	Ocurre cuando el DOM se ha cargado completamente.	`document.addEventListener('DOMContentLoaded', () => {});`
`load`	Ocurre cuando una página o recurso específico (imágenes, scripts) se carga.	`window.addEventListener('load', () => {...});`
`resize`	Ocurre cuando se redimensiona la ventana.	`window.addEventListener('resize', () => {...});`
`scroll`	Ocurre cuando el usuario desplaza la barra de scroll.	`window.addEventListener('scroll', () => {...});`
`unload`	Ocurre cuando la página se está descargando.	`window.addEventListener('unload', () => {...});`

---

5. Eventos de Drag & Drop

Evento	Descripción	Ejemplo
`drag`	Ocurre cuando un elemento se arrastra.	`element.addEventListener('drag', () => {...});`
`dragstart`	Se dispara al iniciar el arrastre de un elemento.	`element.addEventListener('dragstart', () => {...});`
`dragend`	Se dispara al terminar el arrastre de un elemento.	`element.addEventListener('dragend', () => {...});`
`dragover`	Ocurre cuando un elemento arrastrado pasa sobre un área válida.	`area.addEventListener('dragover', (e) => { e.preventDefault(); });`
`drop`	Se dispara al soltar un elemento en un área válida.	`area.addEventListener('drop', (e) => {...});`

---

6. Eventos de Multimedia

Evento	Descripción	Ejemplo
`play`	Ocurre cuando se inicia la reproducción de un elemento multimedia.	`video.addEventListener('play', () => {...});`
`pause`	Ocurre cuando la reproducción es pausada.	`audio.addEventListener('pause', () => {...});`
`ended`	Ocurre cuando la reproducción termina.	`video.addEventListener('ended', () => {...});`
`timeupdate`	Se activa cuando el tiempo actual de reproducción cambia.	`video.addEventListener('timeupdate', () => {...});`

---

7. Eventos de Otros Tipos

Evento	Descripción	Ejemplo
`error`	Se activa cuando ocurre un error en la carga de recursos.	`img.addEventListener('error', () => {...});`
`copy`	Ocurre cuando se copia contenido.	`document.addEventListener('copy', () => {...});`
`paste`	Ocurre cuando se pega contenido.	`document.addEventListener('paste', () => {...});`

Diferencia entre `this` y `event.target` en eventos de JavaScript

Cuando trabajamos con manejadores de eventos en JavaScript, a menudo necesitamos distinguir entre dos conceptos clave: **`this`** y **`event.target`**. Aunque ambos se utilizan en el contexto de eventos, representan diferentes cosas.

---

1. `this` en los manejadores de eventos

Qué representa:
Hace referencia al **elemento al que está vinculado el manejador de eventos**. En otras palabras, `this` apunta al **elemento que recibió el evento**.

Cómo funciona:

- El valor de `this` se determina por el contexto de ejecución de la función.
- En un manejador de eventos registrado con `addEventListener`, `this` generalmente apunta al elemento donde se vinculó el manejador (es decir, el elemento que disparó el evento).
- Si usas **funciones flecha** como manejadores de eventos, el valor de `this` no se enlazará al elemento objetivo, sino que conservará el valor del **contexto externo**.

Ejemplo con `this`:

```
<button id="myButton">Click Me</button>

<script>
const button = document.getElementById('myButton');

// Agregamos un evento al botón
button.addEventListener('click', function () {
    console.log(this); // "this" se refiere al botón
});
</script>
```

En este caso, `this` apunta al botón que recibió el clic.

2. `event.target` en los manejadores de eventos

Qué representa:
`event.target` hace referencia al **elemento específico que disparó el evento**. Si un elemento hijo dentro de un contenedor dispara el evento, `event.target` se refiere al hijo, no al contenedor que tiene el manejador.

Cómo funciona:

- **`event.target`** siempre apunta al **elemento donde ocurrió físicamente el evento**, independientemente de dónde se haya registrado el manejador.
- Se usa especialmente cuando estamos manejando eventos en un contenedor o cuando trabajamos con **eventos delegados**.

Ejemplo con `event.target`:

```
<div id="container">
    <button>Button 1</button>
    <button>Button 2</button>
</div>

<script>
const container = document.getElementById('container');
```

```
container.addEventListener('click', function (event) {
    console.log(event.target); // "event.target" se refiere al botón
que fue clicado
    console.log(this);  // "this" se refiere al contenedor <div>
});
</script>
```

Si haces clic en uno de los botones dentro del `div`, `event.target` se referirá al botón específico, mientras que `this` seguirá apuntando al `div` contenedor.

3. ¿Se pueden usar indistintamente?

No siempre. Aunque en algunos casos ambos pueden coincidir (cuando haces clic directamente en el elemento que tiene el manejador), hay situaciones donde no son intercambiables:

- **Usa `this`** cuando necesites trabajar con el **elemento que tiene el manejador de eventos**.
- **Usa `event.target`** cuando necesites identificar **el elemento exacto que originó el evento**, especialmente en eventos delegados.

4. Diferencia clave en eventos delegados

En eventos delegados, el manejador de eventos se registra en un **elemento contenedor** y usamos **`event.target`** para identificar el **elemento hijo** que originó el evento.

Evento Delegado

¿Qué es un evento delegado?

La **delegación de eventos** es una técnica en JavaScript que permite asignar un único manejador de eventos a un **elemento padre** (contenedor) para gestionar los eventos de sus **elementos hijos**. Incluso si los elementos hijos se agregan dinámicamente al DOM, el manejador en el contenedor seguirá funcionando.

Cómo funciona la delegación de eventos

Cuando un evento ocurre en un elemento, **este evento burbujea** (bubble) hacia arriba en el árbol DOM, propagándose desde el elemento que disparó el evento hasta sus ancestros. En los eventos delegados, aprovechamos esta propagación colocando un único manejador en un **elemento ancestro** y utilizando **`event.target`** para identificar el **elemento hijo** que disparó el evento.

Ventajas de la delegación de eventos

1. **Menor consumo de recursos**:
   No necesitamos agregar un manejador a cada elemento hijo, solo uno en el contenedor.
2. **Soporte para elementos dinámicos**:
   Si los elementos hijos son creados o eliminados dinámicamente, el manejador en el contenedor sigue funcionando sin necesidad de reconfigurarlos.
3. **Mantenimiento más sencillo**:
   Con menos manejadores, el código es más fácil de mantener y depurar.

Ejemplo práctico sin delegación

En este ejemplo, necesitamos agregar un manejador de eventos a cada `li` de una lista. Esto puede ser ineficiente si la lista tiene muchos elementos o si se agregan dinámicamente.

```
<ul>
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>

<script>
const items = document.querySelectorAll('li');
items.forEach(item => {
    item.addEventListener('click', () => {
        console.log('Elemento clicado:', item.textContent);
    });
});
</script>
```

Ejemplo práctico con delegación

Aquí, usamos **delegación de eventos**: colocamos un único manejador de eventos en el `ul` (el contenedor de los `li`). Usamos **`event.target`** para identificar qué `li` fue clicado.

```
<ul id="list">
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>
```

```
<script>
const list = document.getElementById('list');
list.addEventListener('click', function (event) {
    if (event.target.tagName === 'LI') {
        console.log('Elemento clicado:', event.target.textContent);
    }
});
</script>
```

En este caso:

- Solo hay un manejador en el `ul`.
- Si agregamos nuevos elementos `li` dinámicamente, el manejador en el `ul` seguirá funcionando.

Cuándo usar eventos delegados

1. **Grandes cantidades de elementos**:
   Es útil cuando tenemos muchos elementos similares y queremos reducir la cantidad de manejadores.
2. **Elementos dinámicos**:
   Ideal para listas, tablas o galerías donde los elementos pueden ser añadidos o eliminados después de cargar la página.
3. **Mejorar el rendimiento**:
   Permite optimizar la memoria y el rendimiento al reducir la cantidad de manejadores y aprovechar la burbujeo de eventos.

Resumen

- **`this`** hace referencia al **elemento al que está vinculado el manejador de eventos**.
- **`event.target`** hace referencia al **elemento que realmente disparó el evento**.
- La **delegación de eventos** permite manejar eventos en un contenedor para todos los elementos dentro de él, utilizando `event.target` para identificar el origen del evento.

Espero que esto te ayude a entender mejor la diferencia entre **`this`** y **`event.target`**, así como los beneficios de usar **eventos delegados** en tus aplicaciones JavaScript.

Tema 6, Modelo de Objetos del Cliente

Atributos Data (`data-*`)

¿Qué son los Data Attributes?

Los **Data Attributes** son atributos personalizados en HTML5 que permiten almacenar datos adicionales en elementos HTML. Se identifican con el prefijo `data-` y son útiles para manejar información dinámica o específica.

html
Copiar código
```
<div data-user-id="12345" data-role="admin">Usuario</div>
```

- **`data-user-id`** y **`data-role`** son atributos personalizados.
- Estos almacenan información (como el ID y el rol del usuario) que puede usarse en JavaScript para manipular o acceder a los datos.

Ventajas:

- **Flexibilidad:** Permite definir atributos personalizados para adaptarse a las necesidades de la aplicación.
- **Compatibilidad:** No interfiere con los atributos estándar de HTML.
- **Acceso fácil desde JavaScript:** Se pueden manipular directamente con el objeto `dataset` o métodos como `getAttribute`.

Acceso desde JavaScript

1. **Usando `getAttribute`**:

```
const element = document.querySelector('div');
console.log(element.getAttribute('data-user-id')); // "12345"
```

2. **Usando `dataset`**:

```
const element = document.querySelector('div');
console.log(element.dataset.userId); // "12345"
element.dataset.role = 'editor'; // Cambia el valor de data-role
```

Propiedades y Métodos del DOM

Propiedades principales de los elementos

Propiedad	Descripción	Ejemplo
`id`	Obtiene o establece el atributo `id` del elemento.	`element.id = 'nuevoId';`
`className`	Obtiene o establece las clases como texto.	`element.className = 'miClase';`
`classList`	Facilita la manipulación de clases como un objeto.	`element.classList.add('miClase');`
`innerHTML`	Obtiene o establece el contenido HTML interno.	`element.innerHTML = '<p>Hola</p>';`
`outerHTML`	Igual que `innerHTML`, pero incluye el propio elemento.	`element.outerHTML = '<div>Nuevo</div>';`
`textContent`	Obtiene o establece el texto, excluyendo etiquetas HTML.	`element.textContent = 'Hola';`
`style`	Manipula estilos CSS en línea.	`element.style.color = 'red';`
`children`	Devuelve una colección de hijos elementos.	`const hijos = element.children;`
`parentElement`	Devuelve el elemento padre.	`const padre = element.parentElement;`
`nextElementSibling`	Devuelve el siguiente hermano elemento.	`const siguiente = element.nextElementSibling;`
`previousElementSibling`	Devuelve el hermano anterior elemento.	`const anterior = element.previousElementSibling;`
`dataset`	Accede a los `data-*` como objeto.	`element.dataset.userId = '123';`

Métodos principales del DOM

Método	Descripción	Ejemplo
`getAttribute(name)`	Obtiene el valor de un atributo.	`element.getAttribute('src');`
`setAttribute(name, value)`	Establece o actualiza un atributo.	`element.setAttribute('alt', 'imagen');`
`removeAttribute(name)`	Elimina un atributo.	`element.removeAttribute('id');`
`hasAttribute(name)`	Verifica si un atributo existe (`true` o `false`).	`element.hasAttribute('href');`
`appendChild(child)`	Agrega un nodo hijo.	`element.appendChild(nuevoElemento);`
`removeChild(child)`	Elimina un nodo hijo.	`element.removeChild(hijo);`
`replaceChild(newChild, oldChild)`	Reemplaza un nodo hijo con otro.	`element.replaceChild(nuevo, viejo);`
`cloneNode(deep)`	Crea una copia del nodo.	`const copia = element.cloneNode(true);`
`insertAdjacentHTML(pos, html)`	Inserta HTML en una posición relativa.	`element.insertAdjacentHTML('beforeend', '<p>Hola</p>');`
`focus()`	Lleva el foco al elemento.	`element.focus();`
`blur()`	Quita el foco del elemento.	`element.blur();`

Manipulación de clases con `classList`

Método	Descripción	Ejemplo
`add(className)`	Agrega una clase al elemento.	`element.classList.add('miClase');`
`remove(className)`	Elimina una clase del elemento.	`element.classList.remove('miClase');`
`toggle(className)`	Alterna entre agregar y eliminar una clase.	`element.classList.toggle('miClase');`
`contains(className)`	Verifica si el elemento tiene una clase (`true/false`).	`element.classList.contains('miClase');`
`replace(old, new)`	Reemplaza una clase por otra.	`element.classList.replace('vieja', 'nueva');`

Selección de Elementos en el DOM

Método	Descripción	Devuelve	Ejemplo
`getElementById(id)`	Selecciona un elemento por su `id`.	Elemento	`document.getElementById('miId');`
`getElementsByClassName(class)`	Selecciona elementos por clase.	HTMLCollection	`document.getElementsByClassName('miClase');`
`getElementsByTagName(tag)`	Selecciona elementos por etiqueta HTML.	HTMLCollection	`document.getElementsByTagName('div');`
`querySelector(selector)`	Selecciona el primer elemento que coincida con un selector CSS.	Elemento	`document.querySelector('.miClase');`

`querySelectorAll(selector)`	Selecciona todos los elementos que coincidan con un selector CSS.	NodeList	`document.querySelectorAll('div');`

Diferencias entre HTMLCollection y NodeList:

- **HTMLCollection:** Se actualiza dinámicamente si cambia el DOM.
- **NodeList:** Es estática y permite métodos como `.forEach()`.

Ejemplo completo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Manipulación DOM Completa</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      line-height: 1.5;
    }
    .dynamic-container {
      border: 1px solid #ddd;
      padding: 10px;
      margin-top: 10px;
    }
    .highlight {
      background-color: #ffeb3b;
      color: #000;
      font-weight: bold;
    }
    button {
      padding: 10px 15px;
      margin: 5px;
      border: none;
```

```
      background-color: #007bff;
      color: white;
      cursor: pointer;
      border-radius: 5px;
    }
    button:hover {
      background-color: #0056b3;
    }
  </style>
</head>
<body>
  <h1>Ejemplo Completo de Manipulación del DOM</h1>
  <p>Presiona los botones para interactuar dinámicamente con el
contenido.</p>

  <button id="add-element">Agregar elemento dinámico</button>
  <button id="toggle-highlight">Alternar clase "highlight"</button>
  <button id="update-data">Actualizar atributos data-*</button>
  <button id="show-data">Mostrar todos los atributos data-*</button>

  <div id="container" class="dynamic-container"
data-name="contenedor" data-count="0">
    <p>Este es un contenedor dinámico. Su contenido cambiará al
interactuar con los botones.</p>
  </div>

  <script>
    // Seleccionar el contenedor principal
    const container = document.getElementById('container');

    // Función para agregar un elemento dinámico
    document.getElementById('add-element').addEventListener('click',
() => {
      const newElement = document.createElement('div');
      const count = parseInt(container.dataset.count) + 1;
      newElement.textContent = `Elemento dinámico #${count}`;
      newElement.setAttribute('data-item', count);
      newElement.className = 'dynamic-item';
      container.appendChild(newElement);

      // Actualizar el contador en data-count
```

```
      container.dataset.count = count;
    });

    // Función para alternar la clase "highlight" en el contenedor

document.getElementById('toggle-highlight').addEventListener('click'
, () => {
      container.classList.toggle('highlight');
    });

    // Función para actualizar atributos data-*
    document.getElementById('update-data').addEventListener('click',
() => {
      const currentName = container.dataset.name;
      const newName = currentName === 'contenedor' ?
'nuevo-contenedor' : 'contenedor';
      container.dataset.name = newName;

      alert(`Atributo "data-name" actualizado a: ${newName}`);
    });

    // Función para mostrar todos los atributos data-*
    document.getElementById('show-data').addEventListener('click',
() => {
      const dataAttributes = Object.entries(container.dataset);
      const messages = dataAttributes.map(([key, value]) =>
`data-${key}: ${value}`);
      alert('Atributos data-* del contenedor:\n' +
messages.join('\n'));
    });

    // Evento dinámico en elementos hijos (delegación de eventos)
    container.addEventListener('click', (event) => {
      if (event.target.classList.contains('dynamic-item')) {
        const itemData = event.target.getAttribute('data-item');
        alert(`Hiciste clic en el elemento dinámico con
data-item="${itemData}"`);
      }
    });
  </script>
</body>
```

</html>

Tema 5, POO

Tabla de Conceptos de POO en JavaScript

Concepto	Descripción	Ejemplo
Clase	Es una plantilla para crear objetos.	`class Persona { constructor(nombre) { this.nombre = nombre; } }`
Objeto	Es una instancia de una clase, creada con el operador `new`.	`const persona1 = new Persona('Juan');`
Propiedad	Es una variable que pertenece a una clase u objeto.	`this.nombre`
Método	Es una función que pertenece a una clase u objeto.	`hablar() { console.log('Hola'); }`
Encapsulación	Es la ocultación de los detalles internos del objeto y la exposición de solo lo necesario a través de métodos públicos.	`#edad = 25; getEdad() { return this.#edad; }`
Herencia	Permite que una clase (subclase) herede propiedades y métodos de otra (superclase).	`class Empleado extends Persona {}`
Polimorfismo	Es la capacidad de un método en una clase derivada para sobrescribir el comportamiento de un método de la clase base.	`saludar() { console.log('¡Hola, empleado!'); }` (sobreescribe `saludar` de la clase base)
Abstracción	Es la capacidad de definir clases abstractas que no pueden instanciarse, utilizadas como base para otras clases.	`class Animal { constructor() { if (new.target === Animal) { throw new Error('No se puede instanciar'); } } }`

Static	Define propiedades o métodos que pertenecen a la clase en lugar de a las instancias.	`static crearPersona() { return new Persona('Predeterminada'); }`
Getters y Setters	Métodos especiales para obtener o establecer valores de propiedades privadas o protegidas.	`get nombre() { return this._nombre; } set nombre(valor) { this._nombre = valor; }`
Prototype	Es un mecanismo por el cual los objetos pueden compartir métodos y propiedades.	`Persona.prototype.saludar = function() { console.log('Hola desde el prototipo'); };`

Sistema básico de gestión de personas y empleados.

```
// Clase base Persona
class Persona {
    #edad; // Encapsulación: propiedad privada

    constructor(nombre, edad) {
        this.nombre = nombre;
        this.#edad = edad;
    }

    // Getter y Setter para edad
    get edad() {
        return this.#edad;
    }

    set edad(nuevaEdad) {
        if (nuevaEdad > 0) {
            this.#edad = nuevaEdad;
        } else {
            console.error('La edad debe ser positiva.');
        }
    }

    // Método de instancia
    saludar() {
        console.log(`Hola, soy ${this.nombre} y tengo ${this.#edad}
años.`);
    }
```

```
    // Método estático
    static crearAnonimo() {
        return new Persona('Anónimo', 30);
    }
}

// Subclase Empleado que hereda de Persona
class Empleado extends Persona {
    constructor(nombre, edad, puesto) {
        super(nombre, edad); // Llama al constructor de la clase base
        this.puesto = puesto;
    }

    // Sobreescritura de método
    saludar() {
        console.log(`Hola, soy ${this.nombre}, trabajo como
${this.puesto}.`);
    }
}

// Uso de las clases
const persona1 = new Persona('Luis', 25);
persona1.saludar(); // Hola, soy Luis y tengo 25 años.
persona1.edad = 26; // Cambia la edad usando el setter
console.log(`Nueva edad: ${persona1.edad}`); // Nueva edad: 26

const empleado1 = new Empleado('Ana', 30, 'Ingeniera');
empleado1.saludar(); // Hola, soy Ana, trabajo como Ingeniera.

const anonimo = Persona.crearAnonimo(); // Método estático
anonimo.saludar(); // Hola, soy Anónimo y tengo 30 años.
```

Expresiones Regulares (RegEx)

Una **expresión regular** es un patrón utilizado para buscar coincidencias dentro de cadenas de texto. Es muy útil para validar, buscar o reemplazar contenido en textos.

---

1. Componentes básicos de una expresión regular

Las expresiones regulares tienen caracteres especiales que ayudan a definir patrones. Aquí tienes algunos elementos básicos:

Elemento	Descripción	Ejemplo	Resultado
`.`	Coincide con **cualquier carácter** excepto nueva línea	`/a.c/`	Coincide con "abc", "axc", pero no con "ac".
`^`	Coincide con el **inicio** de una cadena	`/^Hola/`	Coincide con cadenas que empiezan con "Hola".
`$`	Coincide con el **final** de una cadena	`/adiós$/`	Coincide con cadenas que terminan con "adiós".
`*`	Coincide con **cero o más repeticiones**	`/ab*c/`	Coincide con "ac", "abc", "abbc", etc.
`+`	Coincide con **una o más repeticiones**	`/ab+c/`	Coincide con "abc", "abbc", pero no con "ac".
`?`	Hace que el carácter anterior sea **opcional**	`/colou?r/`	Coincide con "color" y "colour".
`[]`	Define un **conjunto de caracteres** permitidos	`/[aeiou]/`	Coincide con cualquier vocal.
`\d`	Coincide con un **dígito** (equivale a `[0-9]`)	`/\d+/`	Coincide con "123", "456", etc.
`\w`	Coincide con un **carácter alfanumérico** (A-Z, 0-9, _)	`/\w+/`	Coincide con "hello_123".
`\s`	Coincide con un **espacio en blanco**	`/\s+/`	Coincide con espacios, tabulaciones, etc.

`	`	Operador **"o" lógico**	`/gato
( )	Agrupa patrones	/(ab)+/	Coincide con "abab".

---

2. Métodos principales en JavaScript

JavaScript tiene métodos que facilitan el trabajo con expresiones regulares.

test

Comprueba si el patrón existe en una cadena (devuelve true o false):

javascript
Copiar código
```
const regex = /hola/i; // La "i" significa insensible a mayúsculas
console.log(regex.test("Hola mundo")); // true
```

match

Devuelve las coincidencias encontradas en un array (o null si no hay coincidencias):

```
const texto = "El número es 12345";
console.log(texto.match(/\d+/)); // ["12345"]
```

replace

Reemplaza el texto que coincide con el patrón:

```
const texto = "Hola mundo";
console.log(texto.replace(/mundo/, "amigos")); // "Hola amigos"
```

split

Divide un string en partes usando una RegEx como separador:

```
const texto = "uno,dos,tres";
console.log(texto.split(/,/)); // ["uno", "dos", "tres"]
```

Validaciones comunes

1. Validar un correo electrónico

Comprueba si una cadena tiene un formato de correo válido.

```
const emailRegex =
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

console.log(emailRegex.test("usuario@dominio.com")); // true
console.log(emailRegex.test("usuario@dominio"));     // false
```

---

2. Validar un número de teléfono (formato internacional)

Formato: +34 600 123 456 o 600123456.

```
const phoneRegex = /^(\+34\s?)?(\d{9}|\d{3}\s\d{3}\s\d{3})$/;

console.log(phoneRegex.test("+34 600 123 456")); // true
console.log(phoneRegex.test("600123456"));      // true
console.log(phoneRegex.test("12345"));          // false
```

---

3. Validar una URL

Comprueba si una cadena tiene el formato de una URL.

```
const urlRegex =
/^(https?:\/\/)?([\w.-]+)\.([a-z]{2,6})(\/[^\s]*)?$/;

console.log(urlRegex.test("https://www.example.com")); // true
console.log(urlRegex.test("http://example.org/page")); // true
console.log(urlRegex.test("www.example"));            // false
```

Búsquedas y coincidencias

4. Buscar todas las palabras en una cadena

Devuelve un array con todas las palabras (separadas por espacios).

```
const text = "Hola, mundo! Aprende JavaScript.";
const wordRegex = /\b\w+\b/g;

console.log(text.match(wordRegex)); // ["Hola", "mundo", "Aprende",
"JavaScript"]
```

---

5. Buscar números en una cadena

Encuentra todas las secuencias de dígitos.

```
const text = "Hay 15 manzanas y 20 naranjas.";
const numberRegex = /\d+/g;

console.log(text.match(numberRegex)); // ["15", "20"]
```

---

6. Extraer etiquetas HTML

Obtén todas las etiquetas <...> de un texto.

```
const html = "<div><p>Hola</p><span>Mundo</span></div>";
const tagRegex = /<\/?[\w\s="'-]+>/g;

console.log(html.match(tagRegex)); // ["<div>", "<p>", "</p>",
"<span>", "</span>", "</div>"]
```

Reemplazos

7. Reemplazar todas las vocales

Convierte todas las vocales en mayúsculas.

```
const text = "Hola, mundo!";
const vowelRegex = /[aeiouáéíóúü]/gi;

console.log(text.replace(vowelRegex, match => match.toUpperCase()));
```

```
// "HOlA, mUndO!"
```

---

8. Eliminar espacios extra

Reduce múltiples espacios consecutivos a uno solo.

```
const text = "Este   texto   tiene   muchos   espacios.";
const spaceRegex = /\s+/g;

console.log(text.replace(spaceRegex, " ")); // "Este texto tiene
muchos espacios."
```

---

9. Censurar palabras inapropiadas

Reemplaza palabras no deseadas por asteriscos.

```
const text = "Esto es un mal ejemplo de lenguaje grosero.";
const badWordRegex = /\b(mal|grosero)\b/gi;

console.log(text.replace(badWordRegex, "****"));
// "Esto es un **** ejemplo de lenguaje ****."
```

Validaciones avanzadas

10. Validar contraseñas fuertes

Requiere al menos 8 caracteres, una mayúscula, una minúscula, un número y un carácter especial.

```
const passwordRegex =
/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}
$/;

console.log(passwordRegex.test("Hola123!"));    // true
console.log(passwordRegex.test("hola123"));     // false
console.log(passwordRegex.test("12345678"));    // false
```

---

11. Validar fechas (formato dd/mm/yyyy o dd-mm-yyyy)

Asegúrate de que la fecha tenga un formato correcto.

```
const dateRegex =
/^(0?[1-9]|[12][0-9]|3[01])[\/-](0?[1-9]|1[0-2])[\/-]\d{4}$/;

console.log(dateRegex.test("25/12/2024")); // true
console.log(dateRegex.test("31-01-1999")); // true
console.log(dateRegex.test("99/99/9999")); // false
```

---

12. Validar códigos postales (España)

Valida un código postal español de 5 dígitos, comenzando con 01-52.

```
const postalCodeRegex = /^(0[1-9]|[1-4]\d|5[0-2])\d{3}$/;

console.log(postalCodeRegex.test("28080")); // true
console.log(postalCodeRegex.test("52999")); // false
```

Métodos para practicar con RegEx en JavaScript

Combinación de búsqueda y validación

Podemos buscar coincidencias y, al mismo tiempo, validarlas.

```
const text = "Mi correo es usuario@example.com y mi web es
https://example.com.";
const emailRegex =
/[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}/g;
const urlRegex = /https?:\/\/[^\s]+/g;

console.log(text.match(emailRegex)); // ["usuario@example.com"]
console.log(text.match(urlRegex));  // ["https://example.com"]
```

Iterar sobre coincidencias

Podemos usar exec en un bucle para capturar múltiples coincidencias.

```
const text = "El precio es $5, luego $10 y después $15.";
const priceRegex = /\$\d+/g;

let match;
while ((match = priceRegex.exec(text)) !== null) {
  console.log(match[0]); // "$5", "$10", "$15"
}
```

Interacción con JSON

El JSON (**JavaScript Object Notation**) es un formato ligero para intercambiar datos. Es fácil de leer y escribir, y está basado en la sintaxis de objetos de JavaScript.

---

1. Conversión de JSON a un objeto JavaScript

Usamos JSON.parse para convertir una cadena JSON en un objeto.

```
const jsonString = '{"nombre": "Luis", "edad": 30}';
const objeto = JSON.parse(jsonString);

console.log(objeto.nombre); // "Luis"
console.log(objeto.edad);   // 30
```

---

2. Conversión de un objeto JavaScript a JSON

Usamos JSON.stringify para convertir un objeto JavaScript a una cadena JSON.

```
const objeto = { nombre: "Ana", edad: 25 };
const jsonString = JSON.stringify(objeto);
```

```
console.log(jsonString); // '{"nombre":"Ana","edad":25}'
```

---

3. Acceso y manipulación de datos JSON

Una vez que el JSON se convierte en un objeto JavaScript, podemos acceder y manipular sus propiedades como cualquier objeto normal.

```
const jsonString = '{"ciudad": "Madrid", "poblacion": 3265000}';
const datos = JSON.parse(jsonString);

// Acceder a datos
console.log(datos.ciudad); // "Madrid"

// Modificar datos
datos.poblacion += 50000;

// Convertir de vuelta a JSON
const nuevoJSON = JSON.stringify(datos);
console.log(nuevoJSON); // '{"ciudad":"Madrid","poblacion":3310000}'
```

---

4. JSON y APIs

Cuando trabajamos con APIs, el formato JSON se utiliza para enviar y recibir datos.

```
fetch("https://api.example.com/data")
  .then(response => response.json()) // Convertir JSON a objeto
  .then(data => console.log(data))   // Trabajar con los datos
  .catch(error => console.error("Error:", error));
```

Resumen

- Las **expresiones regulares** son patrones potentes para validar, buscar o manipular cadenas.

- JSON es un formato clave para el intercambio de datos, y `JSON.parse` y `JSON.stringify` nos permiten convertir entre cadenas JSON y objetos JavaScript.

Tema 4, Funciones

Tipos de Funciones en JavaScript

Tipo	Descripción	Ejemplo
Declarativas	Funciones definidas con la palabra clave `function`. Pueden ser llamadas antes de su declaración debido al hoisting.	`function saludar() { console.log("Hola"); }`
Expresivas	Funciones asignadas a una variable. No tienen hoisting; deben definirse antes de ser llamadas.	`const sumar = function(a, b) { return a + b; };`
Flecha (Arrow)	Funciones concisas con una sintaxis moderna, introducidas en ES6. No tienen su propio `this`.	`const multiplicar = (a, b) => a * b;`
Anónimas	Funciones sin nombre, generalmente usadas como callbacks o en expresiones.	`(function() { console.log("Soy anónima"); })();`
Autoejecutables (IIFE)	Funciones que se ejecutan automáticamente tras definirse.	`(function() { console.log("Hola mundo"); })();`
Generadoras	Funciones que pueden pausar su ejecución y reanudarla con `yield`. Se declaran con `function*`.	`function* contador() { let i = 0; while (true) yield i++; }`
Asíncronas	Permiten trabajar con operaciones asincrónicas usando `async` y `await`.	`async function obtenerDatos() { const res = await fetch(url); }`

Constructores	Usadas con `new` para crear objetos personalizados.	`function Persona(nombre) { this.nombre = nombre; }`

Métodos y Propiedades Relacionados con Funciones

Método/Propiedad	Descripción	Ejemplo
`call`	Llama una función con un `this` y argumentos específicos.	`saludar.call(obj, arg1, arg2);`
`apply`	Similar a `call`, pero los argumentos se pasan en un array o `arguments`.	`saludar.apply(obj, [arg1, arg2]);`
`bind`	Crea una nueva función con `this` vinculado a un objeto específico.	`const nuevaFuncion = saludar.bind(obj);`
`arguments`	Objeto especial disponible dentro de las funciones que contiene los argumentos pasados a la función.	`function suma() { return arguments[0] + arguments[1]; }`
`length`	Devuelve el número de argumentos esperados por la función.	`function sumar(a, b) {}; console.log(sumar.length);`
`name`	Devuelve el nombre de la función.	`console.log(saludar.name);`
`toString`	Devuelve una cadena con la definición de la función.	`console.log(saludar.toString());`
`constructor`	Devuelve la función constructora de la función (generalmente `Function`).	`console.log(sumar.constructor);`

Ejemplo Práctico con Diferentes Tipos de Funciones

```
// Declarativa
function saludar(nombre) {
    console.log(`Hola, ${nombre}`);
}
```

```
saludar("Carlos");

// Expresiva
const sumar = function(a, b) {
    return a + b;
};
console.log("Suma:", sumar(5, 3));

// Flecha
const multiplicar = (a, b) => a * b;
console.log("Multiplicación:", multiplicar(4, 6));

// Generadora
function* contador() {
    let i = 0;
    while (true) yield i++;
}
const gen = contador();
console.log("Generador:", gen.next().value, gen.next().value);

// Asíncrona
const fetchSimulado = () => new Promise(resolve => setTimeout(() =>
resolve("Datos recibidos"), 2000));

async function obtenerDatos() {
    const datos = await fetchSimulado();
    console.log(datos);
}
obtenerDatos();

// Autoejecutable (IIFE)
(function() {
    console.log("Hola desde una IIFE");
})();

// Constructor
function Persona(nombre) {
    this.nombre = nombre;
}
const juan = new Persona("Juan");
console.log("Persona:", juan.nombre);
```

```
// Uso de call, apply y bind
const persona = {
    nombre: "Luis",
    saludar: function(lugar) {
        console.log(`Hola, soy ${this.nombre} y estoy en ${lugar}`);
    }
};
persona.saludar("Madrid");

const otraPersona = { nombre: "Ana" };
persona.saludar.call(otraPersona, "Barcelona");
persona.saludar.apply(otraPersona, ["Valencia"]);
const saludarDesdeSevilla = persona.saludar.bind(otraPersona,
"Sevilla");
saludarDesdeSevilla();
```

Tema 3, Estructura de datos

Tabla de Conceptos de Estructuras de Datos

Estructura	Descripción	Ejemplo
Array	Colección ordenada de elementos, accesibles por índices numéricos.	`const numeros = [1, 2, 3]; console.log(numeros[0]); // 1`
Objeto (Diccionario)	Colección de pares clave-valor, donde las claves son únicas y los valores pueden ser de cualquier tipo.	`const persona = { nombre: 'Ana', edad: 25 }; console.log(persona.nombre); // Ana`
Set (Conjunto)	Colección de valores únicos, no ordenados.	`const conjunto = new Set([1, 2, 2, 3]); console.log(conjunto); // Set { 1, 2, 3 }`
Map (Mapa)	Colección de pares clave-valor que preserva el orden de inserción y permite claves de cualquier tipo.	`const mapa = new Map([['clave1', 'valor1'], ['clave2', 'valor2']]); console.log(mapa.get('clave1')); // valor1`

WeakMap	Similar a un Map, pero las claves deben ser objetos y las referencias son débiles (no impiden GC).	`const objeto = {}; const weakMap = new WeakMap([[objeto, 'valor']]); console.log(weakMap.get(objeto)); // valor`
WeakSet	Similar a un Set, pero solo admite objetos como valores y utiliza referencias débiles.	`const objeto = {}; const weakSet = new WeakSet([objeto]); console.log(weakSet.has(objeto)); // true`

Operaciones Comunes

Operación	Array	Objeto (Diccionario)	Set	Map
Agregar elementos	`arr.push(valor)`	`objeto['clave'] = valor`	`set.add(valor)`	`map.set(clave, valor)`
Acceder a elementos	`arr[indice]`	`objeto['clave']`	No aplica (iterar)	`map.get(clave)`
Eliminar elementos	`arr.splice(indice, 1)`	`delete objeto['clave']`	`set.delete(valor)`	`map.delete(clave)`
Verificar existencia	`arr.includes(valor)`	`'clave' in objeto`	`set.has(valor)`	`map.has(clave)`
Iterar elementos	`arr.forEach()` o `for...of`	`for...in` o `Object.entries()`	`set.forEach()` o `for...of`	`map.forEach()` o `for...of`

Método	Descripción	Ejemplo
`push(valor)`	Agrega uno o más elementos al final del array.	`arr.push(4);`
`pop()`	Elimina y devuelve el último elemento del array.	`arr.pop();`
`shift()`	Elimina y devuelve el primer elemento del array.	`arr.shift();`

`unshift(valor)`	Agrega uno o más elementos al inicio del array.	`arr.unshift(0);`
`splice(start, count)`	Elimina, reemplaza o inserta elementos en el array.	`arr.splice(1, 1, 'nuevo');`
`slice(start, end)`	Devuelve una copia de una porción del array, sin modificar el original.	`arr.slice(1, 3);`
`concat(array)`	Combina dos o más arrays en uno nuevo.	`arr.concat([4, 5]);`
`forEach(callback)`	Ejecuta una función para cada elemento del array.	`arr.forEach(val => console.log(val));`
`map(callback)`	Crea un nuevo array con los resultados de la función aplicada a cada elemento.	`arr.map(val => val * 2);`
`filter(callback)`	Crea un nuevo array con los elementos que cumplen la condición del callback.	`arr.filter(val => val > 2);`
`find(callback)`	Devuelve el primer elemento que cumple la condición del callback.	`arr.find(val => val > 2);`
`findIndex(callback)`	Devuelve el índice del primer elemento que cumple la condición del callback.	`arr.findIndex(val => val > 2);`
`reduce(callback, acc)`	Reduce el array a un solo valor mediante una función acumulativa.	`arr.reduce((sum, val) => sum + val, 0);`
`every(callback)`	Devuelve `true` si todos los elementos cumplen la condición del callback.	`arr.every(val => val > 0);`
`some(callback)`	Devuelve `true` si al menos un elemento cumple la condición del callback.	`arr.some(val => val > 2);`
`includes(valor)`	Verifica si el array contiene un valor específico.	`arr.includes(3);`
`indexOf(valor)`	Devuelve el índice de la primera aparición de un valor, o `-1` si no está.	`arr.indexOf(2);`

`lastIndexOf(valor)`	Devuelve el índice de la última aparición de un valor, o `-1` si no está.	`arr.lastIndexOf(2);`
`join(separador)`	Combina los elementos en una cadena, separándolos por el separador especificado.	`arr.join(', ');`
`reverse()`	Invierte el orden de los elementos en el array.	`arr.reverse();`
`sort(callback)`	Ordena los elementos del array.	`arr.sort((a, b) => a - b);`
`flat(profundidad)`	Aplana arrays anidados hasta la profundidad especificada.	`arr.flat(2);`
`flatMap(callback)`	Aplica una función a cada elemento y aplana el resultado en un nivel.	`arr.flatMap(val => [val, val * 2]);`
`keys()`	Devuelve un iterador con los índices del array.	`for (let i of arr.keys()) { console.log(i); }`
`values()`	Devuelve un iterador con los valores del array.	`for (let v of arr.values()) { console.log(v); }`
`entries()`	Devuelve un iterador con pares `[índice, valor]` del array.	`for (let [i, v] of arr.entries()) { ... }`

Métodos de Objetos (Diccionarios)

Método/Propiedad	Descripción	Ejemplo
`Object.keys(obj)`	Devuelve un array con las claves del objeto.	`Object.keys(obj); // ['a', 'b']`
`Object.values(obj)`	Devuelve un array con los valores del objeto.	`Object.values(obj); // [1, 2]`
`Object.entries(obj)`	Devuelve un array de pares `[clave, valor]`.	`Object.entries(obj); // [['a', 1], ...]`

`Object.assign(obj1, obj2)`	Copia las propiedades de un objeto a otro.	`Object.assign({}, obj);`
`Object.freeze(obj)`	Congela un objeto, impidiendo modificaciones.	`Object.freeze(obj);`
`Object.seal(obj)`	Sella un objeto, permitiendo solo cambios en propiedades existentes.	`Object.seal(obj);`
`hasOwnProperty(prop)`	Verifica si el objeto tiene una propiedad específica.	`obj.hasOwnProperty('a');`

Métodos de Sets

Método	Descripción	Ejemplo
`add(valor)`	Agrega un valor al conjunto.	`set.add(1);`
`delete(valor)`	Elimina un valor del conjunto.	`set.delete(1);`
`has(valor)`	Verifica si un valor está en el conjunto.	`set.has(1);`
`clear()`	Elimina todos los valores del conjunto.	`set.clear();`
`forEach(callback)`	Itera sobre los valores del conjunto, ejecutando una función para cada valor.	`set.forEach(val => console.log(val));`
`values()`	Devuelve un iterador con los valores del conjunto.	`for (let v of set.values()) { ... }`

Métodos de Maps

Método	Descripción	Ejemplo
`set(clave, valor)`	Agrega un par clave-valor al mapa.	`map.set('a', 1);`
`get(clave)`	Obtiene el valor asociado a una clave.	`map.get('a');`
`delete(clave)`	Elimina un par clave-valor del mapa.	`map.delete('a');`

`has(clave)`	Verifica si una clave está en el mapa.	`map.has('a');`
`clear()`	Elimina todos los pares clave-valor del mapa.	`map.clear();`
`forEach(callback)`	Itera sobre los pares clave-valor, ejecutando una función para cada uno.	`map.forEach((v, k) => console.log(k));`
`keys()`	Devuelve un iterador con todas las claves del mapa.	`map.keys();`
`values()`	Devuelve un iterador con todos los valores del mapa.	`map.values();`
`entries()`	Devuelve un iterador con pares `[clave, valor]`.	`map.entries();`

```
// 1. Usando Arrays para almacenar productos
const productos = [
    { id: 1, nombre: 'Manzana', categoria: 'Frutas', precio: 0.5 },
    { id: 2, nombre: 'Pan', categoria: 'Panadería', precio: 1 },
    { id: 3, nombre: 'Leche', categoria: 'Lácteos', precio: 1.5 }
];

// 2. Usando un Objeto como Diccionario para la categoría
const categorias = {
    Frutas: 'Alimentos frescos',
    Panadería: 'Productos horneados',
    Lácteos: 'Productos derivados de leche'
};

// 3. Usando un Set para verificar duplicados
const idsUnicos = new Set();
productos.forEach(producto => {
    if (idsUnicos.has(producto.id)) {
        console.log(`Duplicado encontrado: ${producto.nombre}`);
    } else {
        idsUnicos.add(producto.id);
    }
});

// 4. Usando un Map para un inventario con cantidades
const inventario = new Map();
productos.forEach(producto => {
    inventario.set(producto.nombre, { cantidad: 10, precio: producto.precio });
});
```

```
// Operaciones sobre el inventario
// Agregar un producto
inventario.set('Queso', { cantidad: 5, precio: 2.5 });

// Verificar existencia
if (inventario.has('Manzana')) {
    console.log('Manzana está en el inventario.');
}

// Actualizar cantidades
if (inventario.has('Pan')) {
    const pan = inventario.get('Pan');
    pan.cantidad += 5;
    inventario.set('Pan', pan);
}

// Calcular el valor total del inventario
let valorTotal = 0;
inventario.forEach(({ cantidad, precio }) => {
    valorTotal += cantidad * precio;
});
console.log(`Valor total del inventario: $${valorTotal.toFixed(2)}`);

// 5. Usando WeakSet para registrar objetos temporales
const objetosTemporales = new WeakSet();
const productoTemporal = { id: 99, nombre: 'Sandía', categoria: 'Frutas' };
objetosTemporales.add(productoTemporal);
console.log(objetosTemporales.has(productoTemporal)); // true

// Limpieza del objeto temporal
// Una vez que el objeto no tiene referencias, WeakSet lo elimina automáticamente.
```

Tema 2, Control de flujo

Estructuras de Control Condicionales

Estructura	Descripción	Sintaxis/Ejemplo
`if`	Ejecuta un bloque de código si la condición es verdadera.	`js if (x > 0) { console.log("Positivo"); }`
`if...else`	Ejecuta un bloque de código si la condición es verdadera; otro bloque si es falsa.	`js if (x > 0) { console.log("Positivo"); } else { console.log("No positivo"); }`

`else if`	Agrega condiciones adicionales en una cadena de decisiones.	`js if (x > 0) { console.log("Positivo"); } else if (x < 0) { console.log("Negativo"); } else { console.log("Cero"); }`
`switch`	Evalúa una expresión y ejecuta el bloque correspondiente a su valor.	`js switch (color) { case "rojo": console.log("Stop"); break; default: console.log("Sigue"); }`
Operador Ternario (`? :`)	Simplifica una condición con dos resultados.	`js const mensaje = x > 0 ? "Positivo" : "No positivo"; console.log(mensaje);`

Estructuras de Control de Bucles

Estructura	Descripción	Sintaxis/Ejemplo
`for`	Itera un número determinado de veces, usando un contador.	`js for (let i = 0; i < 5; i++) { console.log(i); }`
`for...of`	Itera sobre elementos de un objeto iterable (como arrays o strings).	`js for (const item of [1, 2, 3]) { console.log(item); }`
`for...in`	Itera sobre las propiedades enumerables de un objeto.	`js for (const key in {a: 1, b: 2}) { console.log(key); }`
`while`	Repite mientras la condición sea verdadera.	`js let i = 0; while (i < 5) { console.log(i); i++; }`
`do...while`	Similar a `while`, pero asegura que el bloque se ejecute al menos una vez.	`js let i = 0; do { console.log(i); i++; } while (i < 5);`

Control de Flujo en Bucles

Estructura	Descripción	Sintaxis/Ejemplo

`break`	Termina un bucle o `switch` antes de que complete todas las iteraciones.	`js for (let i = 0; i < 5; i++) { if (i === 3) break; console.log(i); }`
`continue`	Salta a la siguiente iteración del bucle.	`js for (let i = 0; i < 5; i++) { if (i === 3) continue; console.log(i); }`

Estructuras de Control con Excepciones

Estructura	Descripción	Sintaxis/Ejemplo
`try...catch`	Captura errores que ocurren en el bloque `try`.	`js try { let x = a; } catch (error) { console.log("Error: ", error.message); }`
`try...catch...finally`	Permite ejecutar un bloque de código al final, independientemente de si hubo errores.	`js try { let x = a; } catch (error) { console.log("Error"); } finally { console.log("Terminado"); }`
`throw`	Lanza una excepción personalizada.	`js throw new Error("Algo salió mal");`

Estructuras Asíncronas

Estructura	Descripción	Sintaxis/Ejemplo
`async...await`	Permite escribir código asíncrono que parece sincrónico.	`js async function fetchData() { const data = await fetch(url); return data.json(); }`
`Promise`	Representa una operación que se resolverá en el futuro.	`js new Promise((resolve, reject) => { if (success) resolve("Éxito"); else reject("Error"); });`

Ejemplo Completo con Varias Estructuras de Control

```
// Diccionario con datos de estudiantes
const estudiantes = [
    { nombre: "Ana", nota: 90 },
    { nombre: "Luis", nota: 75 },
    { nombre: "Juan", nota: 80 },
];

// Evaluación de estudiantes
for (const estudiante of estudiantes) {
    const resultado = estudiante.nota >= 80
        ? "Aprobado"
        : "Reprobado";

    console.log(`${estudiante.nombre} está ${resultado}.`);
}

// Agregar estudiante con entrada manual
const nuevoEstudiante = { nombre: "Carlos", nota: 85 };
estudiantes.push(nuevoEstudiante);

// Filtrar aprobados
const aprobados = estudiantes.filter(est => est.nota >= 80);
console.log("Estudiantes aprobados:", aprobados);

// Buscar un estudiante específico
const buscar = "Ana";
const encontrado = estudiantes.find(est => est.nombre === buscar);
if (encontrado) {
    console.log(`${buscar} fue encontrado con nota
${encontrado.nota}.`);
} else {
    console.log(`${buscar} no está en la lista.`);
}

// Manejo de errores
try {
    const promedio = estudiantes.reduce((acc, est) => acc +
est.nota, 0) / estudiantes.length;
    if (isNaN(promedio)) {
        throw new Error("Error al calcular promedio.");
    }
```

```
    console.log("Promedio de notas:", promedio);
} catch (error) {
    console.log("Error:", error.message);
} finally {
    console.log("Evaluación terminada.");
}
```

Métodos Iterativos de Arrays en JavaScript

Aquí tienes una lista completa de los métodos más comunes para recorrer o manipular arrays, incluyendo `forEach` y similares, con sus descripciones y ejemplos.

Método	Descripción	Sintaxis/Ejemplo
`forEach`	Ejecuta una función para cada elemento del array.	`js [1, 2, 3].forEach((num) => console.log(num)); // Imprime 1, 2, 3`
`map`	Crea un nuevo array con los resultados de aplicar una función a cada elemento.	`js const dobles = [1, 2, 3].map((num) => num * 2); console.log(dobles); // [2, 4, 6]`
`filter`	Crea un nuevo array con los elementos que cumplen una condición.	`js const pares = [1, 2, 3, 4].filter((num) => num % 2 === 0); console.log(pares); // [2, 4]`
`reduce`	Aplica una función acumulativa a los elementos del array para reducirlo a un solo valor.	`js const suma = [1, 2, 3].reduce((acc, num) => acc + num, 0); console.log(suma); // 6`
`reduceRight`	Similar a `reduce`, pero recorre el array desde el final hacia el principio.	`js const concat = ['a', 'b', 'c'].reduceRight((acc, char) => acc + char, ''); console.log(concat); // "cba"`
`some`	Devuelve `true` si al menos un elemento cumple una condición.	`js const hayNegativos = [1, -2, 3].some((num) => num < 0); console.log(hayNegativos); // true`

every	Devuelve `true` si todos los elementos cumplen una condición.	`js const todosPositivos = [1, 2, 3].every((num) => num > 0); console.log(todosPositivos); // true`
find	Devuelve el primer elemento que cumple una condición.	`js const primeroPar = [1, 2, 3].find((num) => num % 2 === 0); console.log(primeroPar); // 2`
findIndex	Devuelve el índice del primer elemento que cumple una condición.	`js const indicePar = [1, 2, 3].findIndex((num) => num % 2 === 0); console.log(indicePar); // 1`
flat	Aplana un array de arrays en uno de una sola dimensión.	`js const plano = [1, [2, 3], [4, [5]]].flat(2); console.log(plano); // [1, 2, 3, 4, 5]`
flatMap	Aplica un `map` y luego aplana el resultado en un único array.	`js const resultado = [1, 2, 3].flatMap((num) => [num, num * 2]); console.log(resultado); // [1, 2, 2, 4, 3, 6]`
includes	Verifica si un array contiene un valor específico.	`js const existe = [1, 2, 3].includes(2); console.log(existe); // true`
indexOf	Devuelve el índice de la primera aparición de un valor.	`js const indice = [1, 2, 3].indexOf(2); console.log(indice); // 1`
lastIndexOf	Devuelve el índice de la última aparición de un valor.	`js const ultimoIndice = [1, 2, 3, 2].lastIndexOf(2); console.log(ultimoIndice); // 3`
sort	Ordena los elementos del array (modifica el array original).	`js const numeros = [3, 1, 2]; numeros.sort((a, b) => a - b); console.log(numeros); // [1, 2, 3]`
reverse	Invierte el orden de los elementos del array (modifica el array original).	`js const invertido = [1, 2, 3].reverse(); console.log(invertido); // [3, 2, 1]`

Ejemplo Completo con Métodos Iterativos

```
const frutas = ['manzana', 'banana', 'cereza', 'banana'];

// forEach
frutas.forEach((fruta, index) => console.log(`${index}: ${fruta}`));

// map
const longitudes = frutas.map((fruta) => fruta.length);
console.log("Longitudes:", longitudes);

// filter
const sinBananas = frutas.filter((fruta) => fruta !== 'banana');
console.log("Frutas sin banana:", sinBananas);

// reduce
const totalLetras = frutas.reduce((acumulado, fruta) => acumulado +
fruta.length, 0);
console.log("Total de letras:", totalLetras);

// some
const tieneCereza = frutas.some((fruta) => fruta === 'cereza');
console.log("¿Hay cereza?", tieneCereza);

// every
const todasSonLargas = frutas.every((fruta) => fruta.length > 3);
console.log("¿Todas tienen más de 3 letras?", todasSonLargas);

// find
const primeraBanana = frutas.find((fruta) => fruta === 'banana');
console.log("Primera banana encontrada:", primeraBanana);

// sort
const frutasOrdenadas = [...frutas].sort();
console.log("Frutas ordenadas:", frutasOrdenadas);
```