MANUAL DE JAVASCRIPT PARA F.P.

José Carlos do Carmo Raposo
I.E.S. La Marisma (Huelva)

1. Introducción

1.1. Historia y Evolución

JavaScript es un lenguaje de programación que fue **creado en 1995 por Brendan Eich** mientras trabajaba en Netscape Communications Corporation. Inicialmente llamado **"Mocha"** y luego **"LiveScript"**, finalmente se lanzó como **"JavaScript"** para aprovechar la popularidad del lenguaje Java. A pesar de su nombre, JavaScript no está relacionado directamente con Java; fue diseñado para ser un lenguaje de scripting ligero y fácil de usar que permitiera a los desarrolladores agregar interactividad a las páginas web, fundamentalmente para la validación de formularios.

Desde su creación, JavaScript ha evolucionado significativamente. La estandarización del lenguaje está a cargo de ECMA International bajo el nombre oficial de ECMAScript. Las versiones más notables incluyen:

- **ECMAScript 3 (1999):** Introdujo características básicas que son la base del lenguaje moderno.

- **ECMAScript 5 (2009):** Agregó muchas características importantes como el modo estricto y los métodos de array.

- **ECMAScript 6 (2015):** También conocido como ES6 o ES2015, introdujo una gran cantidad de mejoras y nuevas funcionalidades, como las clases, las funciones flecha y las promesas.

1.2. Programación de cliente versus programación del servidor

En el desarrollo web, la programación se divide en dos grandes categorías: “lado del cliente” y “lado del servidor”. Ambas son esenciales para crear aplicaciones web dinámicas y funcionales, pero tienen roles y características distintas.

a) **Programación del lado del cliente (Frontend):** La programación del lado del cliente se refiere a todo lo que ocurre en el navegador del usuario. Se encarga de gestionar la interacción y la experiencia del usuario con el sitio web. Los lenguajes más comunes usados en este tipo de programación son HTML, CSS y JavaScript.

   Características clave:

   1. Interfaz de usuario (UI): Todo lo que el usuario ve y con lo que interactúa, como botones, formularios, y diseño visual, se construye en el lado del cliente.

   2. Rapidez en la interacción: La mayoría de las interacciones se manejan directamente en el navegador, lo que permite respuestas rápidas sin necesidad de realizar solicitudes al servidor.

   3. Seguridad limitada: El código del lado del cliente es accesible para el usuario, lo que significa que no debe incluir datos sensibles o lógica crítica.

   4. Ejemplos: La validación de formularios antes de enviarlos, la carga dinámica de contenido (usando frameworks como React o Angular), o la animación y manipulación del DOM.

   Ventajas:

   - Menos carga en el servidor.
   - Experiencia de usuario más rápida y fluida.
   - Mejora la interactividad y la personalización.

   Desventajas:

   - Seguridad limitada.
   - Depende del rendimiento del dispositivo y navegador del usuario.
   - No es adecuado para manejar lógica compleja o procesos que requieran acceso a bases de datos.

b) **Programación del lado del servidor (Backend)**: La programación del lado del servidor se encarga de gestionar toda la lógica detrás de escena, procesando las solicitudes de los usuarios, interactuando con bases de datos y enviando respuestas al cliente. Lenguajes comunes para esta programación incluyen PHP, Python, Ruby, Node.js, y Java.

   Características clave:

1. Gestión de datos: Es responsable de almacenar, recuperar y procesar la información en bases de datos.

2. Seguridad y lógica crítica: La lógica compleja y los datos confidenciales se manejan aquí, lo que permite un mayor control sobre la seguridad.

3. Generación dinámica de contenido: El contenido se puede generar y personalizar en función de las solicitudes del cliente antes de enviarse al navegador.

4. Ejemplos: Autenticación de usuarios, procesamiento de formularios, generación de páginas web dinámicas, gestión de sesiones y cookies.

Ventajas:
- Seguridad más robusta.
- Maneja grandes volúmenes de datos y operaciones complejas.
- Centralización de la lógica empresarial.

Desventajas:
- Requiere más recursos y mantenimiento.
- El procesamiento es más lento en comparación con el lado del cliente, ya que se depende de las solicitudes y respuestas entre cliente y servidor.

Diferencias clave

1. Ubicación de la ejecución: La programación del lado del cliente se ejecuta en el navegador del usuario, mientras que la del lado del servidor se ejecuta en el servidor web.

2. Seguridad: El lado del servidor ofrece mayor seguridad al no exponer la lógica o datos críticos al usuario final.

3. Interacción: El lado del cliente es responsable de la experiencia de usuario, mientras que el servidor maneja la lógica empresarial y las operaciones críticas.

4. Dependencia de Internet: La programación del lado del cliente puede funcionar parcialmente sin conexión, mientras que el servidor depende completamente de la conexión a internet para procesar las solicitudes.

1.3. Importancia en el Desarrollo Web

El lenguaje del lado del cliente se refiere a los lenguajes de programación y tecnologías que se ejecutan directamente en el navegador web del usuario, es decir, en el cliente. Estos lenguajes son fundamentales para crear interfaces de usuario interactivas y dinámicas en las aplicaciones web. A diferencia del lenguaje del lado del servidor, que se ejecuta en el servidor y maneja la lógica de negocio y el acceso a bases de datos, el lenguaje del lado del cliente se encarga de presentar la información de manera efectiva al usuario y de gestionar la interacción en tiempo real.

Los lenguajes más comunes del lado del cliente incluyen:

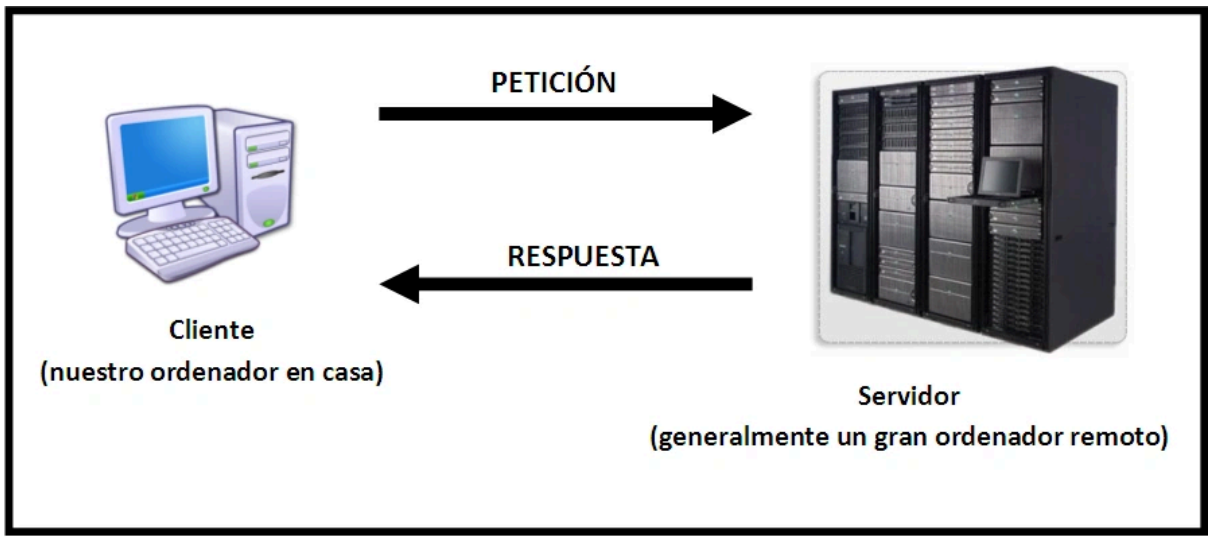
- **HTML (HyperText Markup Language):** Es el pseudo lenguaje estándar, ya que carece de estructuras de control, para crear y estructurar el contenido de una página web. HTML define la estructura de una página web mediante etiquetas que especifican elementos como encabezados, párrafos, imágenes, y enlaces.

- **CSS (Cascading Style Sheets):** CSS se utiliza para definir el estilo y la apariencia de los documentos HTML. Permite a los desarrolladores controlar el diseño visual, como colores, fuentes, espaciado y la disposición general de los elementos en la página.

- **JavaScript:** Es un lenguaje de programación que permite a los desarrolladores crear páginas web interactivas. JavaScript puede manipular elementos HTML y CSS, responder a eventos del usuario (como clics de botón), validar formularios, y mucho más. Con la ayuda de bibliotecas y frameworks como React, Angular, y Vue.js, JavaScript se ha vuelto aún más poderoso y esencial en el desarrollo web moderno.

- **AJAX (Asynchronous JavaScript and XML):** Aunque no es un lenguaje en sí, AJAX es una técnica que permite a las aplicaciones web enviar y recibir datos de un servidor de forma asíncrona, sin tener que recargar la página completa. Esto mejora la experiencia del usuario al permitir interacciones más rápidas y fluidas.

El lenguaje del lado del cliente es crucial para crear experiencias web modernas y atractivas. Permite que las páginas web sean no solo visualmente agradables, sino también interactivas y responsivas, adaptándose a diferentes dispositivos y resoluciones. Además, con el auge de las aplicaciones web progresivas (PWAs) y las interfaces de usuario complejas, el dominio de estas tecnologías se ha vuelto indispensable para cualquier desarrollador web.



JavaScript es uno de los tres lenguajes fundamentales del desarrollo web, junto con HTML y CSS. Cada uno tiene un rol específico:

Las características más importantes de este lenguaje son:

a) **Lenguaje Interpretado:** JavaScript es un lenguaje interpretado, lo que significa que no necesita ser compilado antes de ejecutarse. Los navegadores web tienen motores JavaScript integrados que interpretan y ejecutan el código directamente.

b) **Tipado Dinámico:** Las variables en JavaScript no tienen un tipo fijo y pueden almacenar valores de diferentes tipos a lo largo de la ejecución del programa. Esto permite una gran flexibilidad, pero también puede llevar a errores si no se tiene cuidado.

c) **Asincronía y Callbacks:** JavaScript soporta la programación asincrónica, permitiendo la ejecución de tareas sin bloquear el hilo principal. Las funciones de callback, las promesas y async/await son mecanismos utilizados para manejar operaciones asíncronas.

d) **Interacción con el DOM:** JavaScript puede interactuar y manipular el DOM (Document Object Model), permitiendo la modificación dinámica del contenido y la estructura de las páginas web.

e) **Ecosistema completo:** JavaScript tiene un vasto ecosistema de bibliotecas y frameworks (como React, Angular, y Vue.js) que aceleran el desarrollo de aplicaciones. También cuenta con una gran comunidad de desarrolladores y abundantes recursos de aprendizaje.

1.4. Inserción de Javascript en HTML

En HTML, puedes insertar JavaScript de tres formas principales: inline, en un bloque <script> en el documento, y referenciando un archivo externo. Cada método tiene sus propias aplicaciones y ventajas dependiendo del contexto en el que lo utilices.

a) **JavaScript Inline:** El JavaScript inline se coloca directamente dentro de un atributo de un elemento HTML. Este método se usa comúnmente para manejar eventos como `click`, `mouseover`, entre otros.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>JavaScript Inline</title>
</head>
<body>
    <button onclick="alert('¡Botón clickeado!')">Haz clic en mí</button>
</body>
</html>
```

En este ejemplo, el código JavaScript que muestra una alerta se encuentra directamente dentro del atributo `onclick` del botón.

b) **JavaScript en un bloque <script> Dentro del Documento HTML:** Este método coloca el código JavaScript directamente dentro de un bloque <script> en el documento HTML. Puedes colocar este bloque en la sección <head> o al final de la sección <body>. Colocar el bloque al final de <body> es una práctica común para asegurar que el DOM esté completamente cargado antes de ejecutar el script.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>JavaScript en Bloque</title>
    <script>
        function saludar() {
            alert('Hola desde el <head>!');
        }
    </script>
</head>
<body>
    <button onclick="saludar()">Haz clic en mí</button>
</body>
</html>
```

c) **JavaScript en un Archivo Externo:** En este método, el código JavaScript se coloca en un archivo separado con una extensión `.js`. Luego, este archivo se enlaza a la página HTML mediante un elemento <script> con el atributo `src`.

```
// script.js
function mostrarMensaje() {
    alert('Hola desde un archivo externo!');
}
```

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>JavaScript Externo</title>
    <script src="script.js" defer></script>
</head>
<body>
    <button onclick="mostrarMensaje()">Haz clic en mí</button>
</body>
</html>
```

1.5. Restricciones del lenguaje

JavaScript, como cualquier lenguaje de programación, tiene varias restricciones y limitaciones que es importante tener en cuenta al desarrollar aplicaciones. A continuación, se detallan algunas de las restricciones y limitaciones más significativas del lenguaje:

a) **Ambiente de Ejecución:** JavaScript se ejecuta principalmente en el navegador web, lo que significa que está limitado por las capacidades y políticas de seguridad del navegador. Algunas operaciones que podrían ser permitidas en un entorno de servidor no están permitidas en el navegador por razones de seguridad.

   JavaScript ejecutado en un navegador tiene acceso limitado a los archivos y recursos del sistema local del usuario. Esto está restringido para proteger la privacidad y la seguridad del usuario.

b) **Seguridad:**

   - **Política de Origen Cruzado (CORS):** JavaScript en el navegador está restringido por la política de mismo origen, que impide que un script de un dominio haga solicitudes a un dominio diferente, a menos que el servidor de destino permita explícitamente estas solicitudes.

   - **Sandboxing:** El código JavaScript se ejecuta en un entorno aislado (sandbox) para proteger al sistema operativo y otras aplicaciones de código potencialmente malicioso.

c) **Tipado Dinámico:** JavaScript es un lenguaje de tipado dinámico, lo que significa que los tipos de datos se verifican en tiempo de ejecución, no en tiempo de compilación. Esto puede llevar a errores que son difíciles de detectar antes de la ejecución aunque por otra parte permite más libertad al programador no teniendo que especificar explícitamente el tipo de la variable.

```
let numero = 5;
let texto = "Hola";
console.log(numero + texto); // Output: "5Hola"
```

d) **Rendimiento:**

   - **Ejecutado en el Hilo Principal:** En la mayoría de los navegadores, JavaScript se ejecuta en el hilo principal, lo que significa que puede bloquear la interfaz de usuario si se realizan operaciones intensivas en el CPU. Aunque los Web Workers permiten la ejecución en hilos en segundo plano, su uso está limitado.

   - **Optimización de Código:** El rendimiento del código JavaScript puede verse afectado por cómo está escrito. El código mal optimizado puede llevar a tiempos de carga más largos y una experiencia de usuario menos fluida.

e) **Compatibilidad del Navegador:** No todos los navegadores implementan las características más recientes del lenguaje de manera uniforme. Esto puede causar problemas de compatibilidad si se utilizan características modernas que no están soportadas por versiones anteriores de navegadores.

JavaScript es un lenguaje poderoso y flexible, pero tiene limitaciones inherentes que los desarrolladores deben tener en cuenta. Comprender estas restricciones ayuda a escribir código más eficiente, seguro y compatible. Además, la evolución constante del lenguaje y del entorno de ejecución (como los navegadores y Node.js) continúa abordando muchas de estas limitaciones, mejorando así la experiencia de desarrollo.

1.6. Otros ámbitos del lenguaje

JavaScript es un lenguaje de programación versátil que se utiliza en una variedad de campos más allá del desarrollo web tradicional. A continuación, se describen algunos de los principales campos y aplicaciones en los que JavaScript desempeña un papel importante:

a) **Desarrollo de Aplicaciones Móviles:** JavaScript se usa en el desarrollo de aplicaciones móviles a través de frameworks y librerías como: React Native, Ionic ó Cordova/PhoneGap.

b) **Desarrollo de Aplicaciones de Escritorio: Electron** es un framework que permite desarrollar aplicaciones de escritorio utilizando tecnologías web como JavaScript, HTML y CSS. Algunas aplicaciones populares construidas con Electron incluyen:
    - Visual Studio Code
    - Slack
    - Discord

c) **Desarrollo de Juegos:**

- **HTML5 y WebGL:** JavaScript se utiliza para el desarrollo de juegos en el navegador mediante:
    - Canvas API: Permite la renderización de gráficos 2D.
    - WebGL: Permite la renderización de gráficos 3D en el navegador.
- **Librerías de Juegos:** Como Phaser y Three.js, que facilitan la creación de juegos interactivos y gráficos avanzados.

d) **Desarrollo de Servidores y Backend**: Node.js: Un entorno de ejecución de JavaScript en el lado del servidor que permite construir aplicaciones web, API y servicios backend. Node.js es conocido por su rendimiento y escalabilidad, y se usa en aplicaciones como:

- **Express:** Un marco de aplicaciones web para Node.js.
- **NestJS:** Un marco para construir aplicaciones backend eficientes y escalables con Node.js.

e) **Internet de las Cosas (IoT)**: JavaScript se usa en la programación de dispositivos y sistemas IoT, normalmente mediante librerías especializadas

   - **Johnny-Five:** Una librería de JavaScript para programar hardware basado en Arduino.
   - **Node-RED:** Una herramienta para conectar dispositivos y servicios en un entorno de desarrollo visual utilizando JavaScript.

f) **Computación en la Nube:** JavaScript se utiliza en el desarrollo de aplicaciones que se ejecutan en la nube y en el diseño de interfaces de usuario para servicios en la nube.

g) **Edición de Video y Multimedia:** JavaScript se usa en la edición y manipulación de video y audio en el navegador con bibliotecas como Video.js que permite la reproducción de video HTML5 ó Three.js para la creación de gráficos 3D que se puede usar en aplicaciones multimedia interactivas.

En resumen, JavaScript es un lenguaje extremadamente flexible y ampliamente utilizado, que se extiende mucho más allá del desarrollo web tradicional. Su capacidad para funcionar tanto en el navegador como en el servidor, junto con su integración con múltiples tecnologías y plataformas, lo convierte en una herramienta valiosa en numerosos campos de desarrollo y aplicaciones.

2. Entorno de Desarrollo

Configurar un entorno de desarrollo adecuado es fundamental para trabajar eficientemente con JavaScript. En esta sección, se describen las herramientas y configuraciones básicas necesarias para empezar a escribir y ejecutar código JavaScript.

2.1. Configuración de un Entorno de Desarrollo

a) Editor de Código

Un buen editor de código es esencial para escribir, editar y depurar código JavaScript. Algunos de los editores más populares incluyen:

- **Visual Studio Code (VSCode):** Un editor gratuito, de código abierto, desarrollado por Microsoft. Ofrece soporte integrado para JavaScript y muchas extensiones que mejoran la productividad.

- **Sublime Text:** Un editor de texto ligero y altamente personalizable. Aunque no es gratuito, ofrece una versión de prueba sin límite de tiempo.

- **Atom:** Un editor de código fuente gratuito y de código abierto desarrollado por GitHub. Es altamente extensible y cuenta con una gran cantidad de paquetes disponibles.

b) Instalación de Node.js

Node.js es un entorno de ejecución para JavaScript que permite ejecutar código JavaScript fuera del navegador. Es útil no solo para desarrollar aplicaciones del lado del servidor, sino también para utilizar herramientas y paquetes de desarrollo. Para instalar Node.js:

1. Descargar e instalar: Visita nodejs.org y descarga el instalador adecuado para tu sistema operativo (Windows, macOS o Linux). Sigue las instrucciones de instalación.

2. Verificar la instalación: Abre una terminal o consola de comandos y ejecuta los siguientes comandos para verificar que Node.js y npm (Node Package Manager) se instalaron correctamente:

```
node -v
npm -v
```

Deberías ver los números de versión de Node.js y npm.

c) Extensiones y Complementos

Para mejorar la experiencia de desarrollo en tu editor de código, puedes instalar varias extensiones. Aquí hay algunas recomendadas para Visual Studio Code:

- **ESLint:** Ayuda a identificar y corregir problemas en tu código JavaScript.

- **Prettier:** Un formateador de código que ayuda a mantener un estilo consistente.

- **Live Server:** Lanza un servidor local con recarga automática para ver los cambios en tiempo real.

2.2. Uso de la Consola del Navegador

La consola del navegador es una herramienta invaluable para depurar y probar código JavaScript. Aquí se muestra cómo utilizarla en los navegadores más populares.

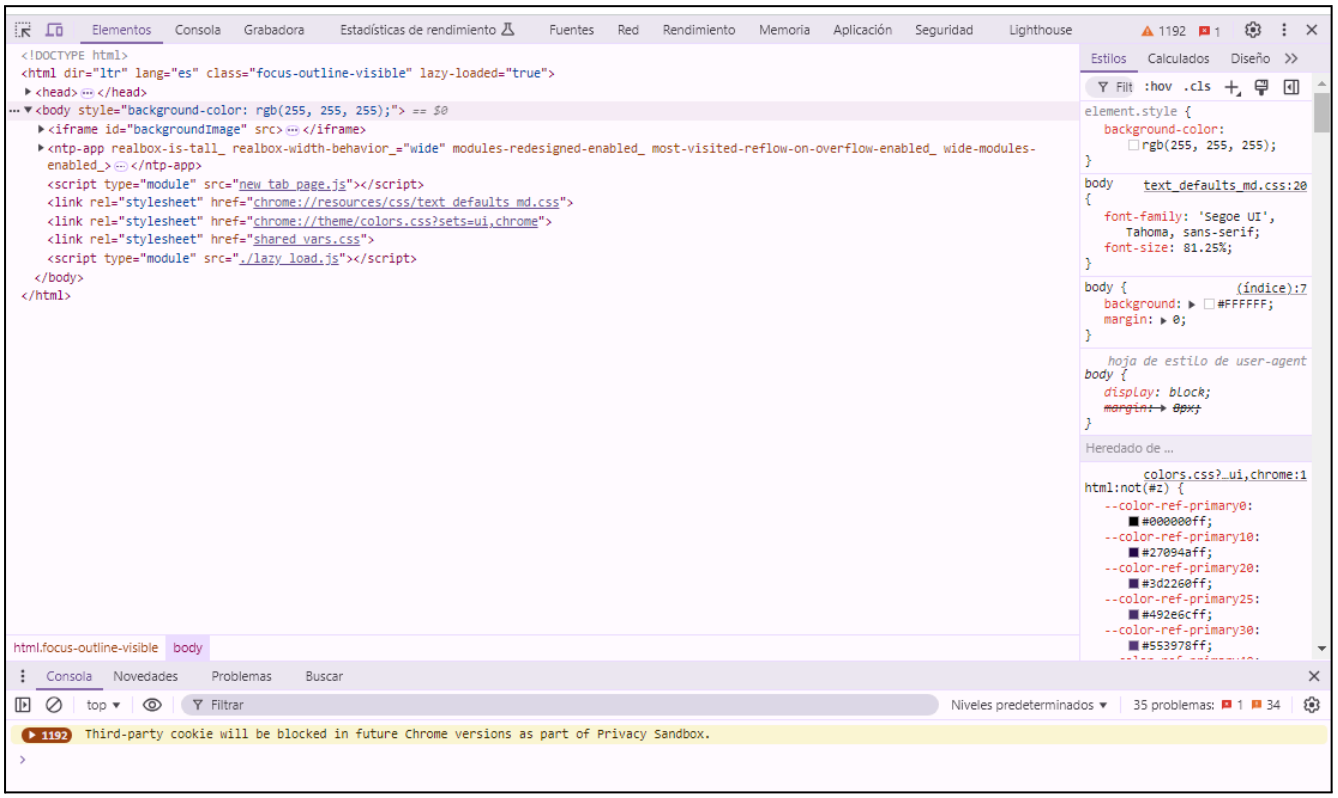
Google Chrome

1. Abrir la consola: Haz clic derecho en cualquier parte de la página web y selecciona "Inspeccionar". Luego, selecciona la pestaña "Consola". También se puede acceder mediante F12.

2. Ejecutar código: Puedes escribir y ejecutar código JavaScript directamente en la consola. Por ejemplo, intenta escribir:

```
console.log("¡Hola, mundo!");
```

Presiona Enter y verás el mensaje "¡Hola, mundo!" en la consola.



Mozilla Firefox

1. Abrir la consola: Haz clic derecho en la página web y selecciona "Inspeccionar". Luego, selecciona la pestaña "Consola".

2. Ejecutar código: Funciona de manera similar a Chrome. Escribe tu código y presiona Enter para ejecutarlo.

Microsoft Edge

1. Abrir la consola: Haz clic derecho en la página web y selecciona "Inspeccionar". Luego, selecciona la pestaña "Consola".

2. Ejecutar código: Escribe y ejecuta código de la misma manera que en los otros navegadores.

Debugging en la Consola del Navegador

La consola del navegador no solo permite ejecutar código, sino también depurarlo. Aquí hay algunas técnicas básicas de depuración:

- **console.log()**: La forma más simple de depurar. Imprime mensajes o valores de variables en la consola.

- **Breakpoints**: Puedes establecer puntos de interrupción en tu código desde la pestaña "Sources" en Chrome o "Debugger" en Firefox. Esto detiene la ejecución del código en una línea específica para inspeccionar el estado del programa.

- **Herramientas de depuración**: La consola también ofrece herramientas avanzadas como el "Step Over" y "Step Into", que te permiten ejecutar el código línea por línea para entender mejor su flujo.

2.3. Cómo depurar código e inspeccionar elementos

La herramienta consola está integrada en los navegadores web (como Chrome, Firefox, Safari, Edge, etc.) y permite a los desarrolladores interactuar con una página web en tiempo real. Su propósito principal es depurar código JavaScript, inspeccionar elementos del DOM, y obtener información útil sobre el estado de una página web o aplicación.

a) **Depuración de Código JavaScript:** La consola permite escribir y ejecutar código JavaScript en tiempo real directamente en el navegador. Esto es útil para probar fragmentos de código sin necesidad de editar los archivos originales. Además, puedes ver errores y advertencias relacionados con el código JavaScript que se ejecuta en la página. Cuando hay un error en el código, el navegador lo muestra en la consola con detalles sobre el tipo de error y la línea en la que ocurrió.

```
let x = 10;
let y = 20;
console.log(x + y);  // 30
```

b) Imprimir Mensajes (console.log, console.error, console.warn):

- console.log(): Se usa para imprimir mensajes o valores de variables. Es muy útil para depurar y ver lo que está ocurriendo en el código.

- console.error(): Muestra mensajes de error en la consola.

- console.warn(): Muestra advertencias en la consola.

c) Inspección del DOM (Document Object Model): La consola te permite acceder e inspeccionar el DOM, que es la estructura de la página HTML. Puedes seleccionar elementos HTML, modificarlos, y ver los cambios reflejados en la página en tiempo real. Este comando cambiaría el texto de un elemento `<h1>` en la página por "Nuevo título".

```
document.querySelector('h1').textContent = 'Nuevo título';
```

d) Ver Errores y Advertencias de Red: La consola del navegador también muestra errores relacionados con la red, como fallos al cargar recursos (imágenes, scripts, archivos CSS) o problemas con solicitudes HTTP (por ejemplo, errores `404`, `500`, etc.). Esto es útil para identificar problemas con la carga de archivos o con el funcionamiento de APIs en una aplicación web.

e) Comprobar el Estado de Objetos y Variables: Puedes inspeccionar el contenido de objetos, arrays y variables en JavaScript. Esto permite ver su estructura, propiedades, y valores. En el siguiente ejemplo la consola mostrará el objeto persona de manera interactiva, lo que permite expandirlo y ver sus propiedades.

```
const persona = { nombre: 'Ana', edad: 25 };
console.log(persona);
```

f) Medir Rendimiento (console.time y console.timeEnd): La consola tiene métodos que permiten medir el tiempo que tarda en ejecutarse un bloque de código, lo cual es útil para identificar cuellos de botella de rendimiento.

```
console.time('Tiempo de ejecución');
// Código que deseas medir
console.timeEnd('Tiempo de ejecución');
```

Esto mostrará el tiempo exacto en milisegundos que tardó en ejecutarse el código entre console.time y console.timeEnd.

g) **Exploración y Manejo de APIs:** La consola es ideal para interactuar con APIs o manejar datos en formato JSON. Puedes hacer peticiones de red y ver la respuesta directamente en la consola.

```
fetch('https://api.example.com/datos')
     .then(response => response.json())
     .then(data => console.log(data));
```

Este código hace una solicitud a una API y luego imprime los datos en la consola.

h) **Depurar con Breakpoints:** Desde la pestaña "Sources" (Fuentes), puedes colocar breakpoints en el código JavaScript para detener la ejecución y examinar el estado actual del programa, las variables, y el flujo de ejecución. Esto te permite inspeccionar paso a paso cómo se ejecuta el código.

2.4. Ejercicios

En este apartado realizaremos una serie de ejercicios que nos ayudarán a practicar la inspección de elementos utilizando la consola del navegador. Estos ejercicios te permitirán interactuar con el DOM, manipular elementos y observar los cambios en tiempo real.

Ejercicio 1: Cambiar el contenido de un elemento

Abre la consola del navegador en una página web. Selecciona un elemento (por ejemplo, un título o párrafo) usando document.querySelector() o inspeccionando el DOM. Usa JavaScript en la consola para cambiar el contenido de ese elemento.

Ejercicio 2: Modificar estilos CSS de un elemento

Selecciona un elemento HTML en la página usando `document.querySelector()` o la herramienta de inspección. Modifica el estilo CSS del elemento, por ejemplo, cambia el color del texto o el fondo. Verifica cómo cambian los estilos en tiempo real.

Ejercicio 3: Agregar y remover clases de un elemento

Selecciona un elemento que tenga clases CSS aplicadas. Usa la consola para agregar una nueva clase usando `classList.add()` y verifica los cambios en el estilo del elemento. Luego, elimina una clase usando `classList.remove()` y observa los efectos.

3. Sintaxis Básica

Dominar la sintaxis básica de JavaScript es el primer paso para convertirse en un desarrollador competente. En esta sección, cubriremos los fundamentos de los comentarios, las variables y los tipos de datos.

Javascript es un lenguaje en el que cada sentencia que escribamos finaliza con punto y coma, aunque estos son opcionales y sólo obligatorios si queremos escribir más de una sentencia en la misma línea.

3.1. Comentarios

Los comentarios son líneas de texto dentro del código que no se ejecutan. Se utilizan para explicar y documentar el código. JavaScript admite dos tipos de comentarios:

a) **Comentarios de una sola línea:** Se utilizan para añadir notas breves. Se inicia con **//**.

```
// Esto es un comentario de una sola línea
console.log("Hola, mundo!"); // Esto imprimirá "Hola, mundo!" en la
consola
```

b) **Comentarios de múltiples líneas:** Se utilizan para añadir notas más extensas. Se encierran entre **/*** y ***/**.

```
/*
Este es un comentario de múltiples líneas.
Se puede utilizar para documentar bloques de código más extensos.
*/
console.log("Hola, mundo!");
```

3.2. Variables

Las variables son contenedores para almacenar datos. En JavaScript, se pueden declarar usando **var, let** o **const.**

a) **Declaración de variables con var**: se utilizaba comúnmente antes de ES6. Tiene un alcance global o de función.

```
var nombre = "Juan";
console.log(nombre); // Salida: Juan
```

b) **Declaración de variables con let**: se introdujo en ES6 y tiene un alcance de bloque, lo que lo hace más seguro y predecible que var.

```
let edad = 25;
console.log(edad); // Salida: 25

if (true) {
  let edad = 30;
  console.log(edad); // Salida: 30 (dentro del bloque)
}

console.log(edad); // Salida: 25 (fuera del bloque)
```

c) **Declaración de variables con const:** también se introdujo en ES6 y se utiliza para declarar constantes, es decir, valores que no cambiarán.

```
const pi = 3.1416;
console.log(pi); // Salida: 3.1416

// Intentar reasignar una constante resultará en un error
// pi = 3.14; // Error: Assignment to constant variable.
```

3.3. Tipos de Datos

JavaScript tiene varios tipos de datos básicos, los cuales se pueden combinar para llegar a producir tipos de datos mucho más complejos.

a) **Números:** Representan valores numéricos como enteros y decimales.

```
let entero = 42;
let decimal = 3.14;
console.log(entero); // Salida: 42
console.log(decimal); // Salida: 3.14
```

En JavaScript, puedes formatear números de varias maneras dependiendo de tus necesidades, como para mostrar decimales, agregar comas de separación, formatear como moneda, etc. Aquí tienes algunas de las formas más comunes de formatear números:

toFixed(): Limitar el número de decimales

```
let num = 1234.56789;
console.log(num.toFixed(2));  // "1234.57"
console.log(num.toFixed(0));  // "1235" (redondea al entero más cercano)
```

toPrecision(): Define el número de dígitos significativos

```
let num = 1234.56789;
console.log(num.toPrecision(5));  // "1234.6" (total de 5 dígitos)
console.log(num.toPrecision(3));  // "1.23e+3" (notación exponencial)
```

b) **Cadenas de Texto (Strings):** Representan texto y se pueden definir con **comillas simples, comillas dobles o backticks.** Si usamos backticks tendremos acceso a la interpolación de cadenas que nos resultará una herramienta muy útil para concatenar cadenas y valores.

```
let saludo = "Hola, mundo!";
let despedida = 'Adiós, mundo!';
let interpolacion = `El valor de pi es ${pi}`;
console.log(saludo); // Salida: Hola, mundo!
console.log(despedida); // Salida: Adiós, mundo!
console.log(interpolacion); // Salida: El valor de pi es 3.1416
```

c) **Booleanos:** Representan valores de verdadero o falso.

```
let esVerdadero = true;
let esFalso = false;
console.log(esVerdadero); // Salida: true
console.log(esFalso); // Salida: false
```

d) **Arrays:** Son listas ordenadas de valores.

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros); // Salida: [1, 2, 3, 4, 5]
console.log(numeros[0]); // Salida: 1 (accediendo al primer elemento)
```

e) **Objetos:** Son colecciones de pares clave-valor.

```
let persona = {
  nombre: "Ana",
  edad: 28,
  ciudad: "Madrid"
};
console.log(persona); // Salida: { nombre: 'Ana', edad: 28, ciudad:
'Madrid' }
console.log(persona.nombre); // Salida: Ana (accediendo a una propiedad)
```

Ejemplos Combinados

```
// Declarando variables
```

```
let nombre = "Carlos";
let edad = 30;
const ciudad = "Barcelona";

// Creando un objeto
let persona = {
  nombre: nombre,
  edad: edad,
  ciudad: ciudad
};

console.log(persona); // Salida: { nombre: 'Carlos', edad: 30, ciudad:
'Barcelona' }

// Modificando propiedades del objeto
persona.edad = 31;
console.log(`La edad actualizada de ${persona.nombre} es
${persona.edad}.`); // Salida: La edad actualizada de Carlos es 31.
```

4. Operadores

Los operadores en JavaScript permiten realizar diversas operaciones sobre variables y valores. Estos se pueden clasificar en varias categorías:

4.1. Operadores Aritméticos

Estos operadores se utilizan para realizar operaciones matemáticas sobre números.

a) **Suma (+):** Suma dos operandos.

```
let a = 5;
let b = 3;
let suma = a + b; // 8
```

Este operador está sobrecargado de modo que si se aplica a una cadena de texto su efecto será la concatenación.

```
let a = 'hola';
let b = 5;
let suma = a + b; // 'Hola5'
```

b) **Resta (-):** Resta el segundo operando del primero.

```
let diferencia = a - b; // 2
```

c) Multiplicación (*): Multiplica dos operandos.

```
let producto = a * b; // 15
```

d) División (/): Divide el primer operando por el segundo. Si el segundo es 0 generaría un error.

```
let product = a / b; // 1
```

e) Módulo (%): Devuelve el residuo de la división del primer operando por el segundo.

```
let remainder = a % b; // 2
```

f) Incremento (++): Incrementa el operando en uno.

```
a++; // a es ahora 6
```

g) Decremento (--): Decrementa el operando en uno.

```
b--; // b es ahora 2
```

4.2. Operadores de Asignación

Estos operadores se utilizan para asignar valores a las variables.

a) Asignación (=): Asigna el valor del operando derecho al operando izquierdo.

```
let c = 10;
```

b) Asignación de Suma (+=): Suma el valor del operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.

```
c += 5; // c es ahora 15
```

c) Asignación de Resta (-=): Resta el valor del operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.

```
c -= 3; // c es ahora 12
```

d) Asignación de Multiplicación (`*=`): Multiplica el valor del operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.

```
c *= 2; // c es ahora 24
```

e) Asignación de División (/=): Divide el valor del operando izquierdo por el operando derecho y asigna el resultado al operando izquierdo.

```
c /= 4; // c es ahora 6
```

f) Asignación de Módulo (%=: Calcula el residuo de la división del operando izquierdo por el operando derecho y asigna el resultado al operando izquierdo.

```
c %= 5; // c es ahora 1
```

4.3. Operadores de Comparación

Estos operadores comparan dos valores y devuelven un valor booleano (`true` o `false`).

a) Igualdad (==): Devuelve `true` si los operandos son iguales.

```
let isEqual = (a == b); // true
```

b) Desigualdad (!=): Devuelve `true` si los operandos no son iguales.

```
let isNotEqual = (a != b); // true
```

c) Igualdad estricta (===): Devuelve `true` si los operandos son iguales y del mismo tipo.

```
let isStrictEqual = (a === b); // false
```

d) Desigualdad estricta (!==): Devuelve `true` si los operandos no son iguales o no son del mismo tipo.

```
let isStrictNotEqual = (a !== b); // true
```

e) Mayor que (>): Devuelve `true` si el operando izquierdo es mayor que el operando derecho.

```
let isGreater = (a > b); // true
```

f) **Mayor o igual que (>=):** Devuelve `true` si el operando izquierdo es mayor o igual que el operando derecho.

```
let isGreaterOrEqual = (a >= b); // true
```

g) **Menor que (<):** Devuelve `true` si el operando izquierdo es menor que el operando derecho.

```
let isLess = (a < b); // false
```

h) **Menor o igual que (<=):** Devuelve `true` si el operando izquierdo es menor o igual que el operando derecho.

```
let isLessOrEqual = (a <= b); // false
```

4.4. Operadores Lógicos

Estos operadores se utilizan para combinar múltiples condiciones lógicas.

a) **AND lógico (&&):** Devuelve `true` si ambos operandos son verdaderos.

```
let isTrue = (a > b && b > 1); // true
```

b) **OR lógico (||):** Devuelve `true` si al menos uno de los operandos es verdadero.

```
let isTrueOr = (a > b || b > 3); // true
```

c) **NOT lógico (!):** Invierte el valor lógico de su operando.

```
let isNotTrue = !(a > b); // false
```

4.5. Operadores de Bit a Bit

Estos operadores trabajan a nivel de bits y realizan operaciones bit a bit.

a) **AND bit a bit (&)**: Realiza una operación AND bit a bit.

```
  let bitAnd = a & b; // 1
```

b) **OR bit a bit (|):** Realiza una operación OR bit a bit.

```
let bitOr = a | b; // 7
```

c) **XOR bit a bit (^):** Realiza una operación XOR bit a bit.

```
let bitXor = a ^ b; // 6
```

d) **NOT bit a bit (~):** Invierte los bits de su operando.

```
let bitNot = ~a; // -6
```

e) **Desplazamiento a la izquierda (<<):** Desplaza los bits de su operando a la izquierda.

```
let leftShift = a << 1; // 10
```

f) **Desplazamiento a la derecha con signo (>>):** Desplaza los bits de su operando a la derecha, manteniendo el signo.

```
let rightShift = a >> 1; // 2
```

g) **Desplazamiento a la derecha sin signo (>>>):** Desplaza los bits de su operando a la derecha, rellenando con ceros.

```
let unsignedRightShift = a >>> 1; // 2
```

4.6. Operador Ternario

El operador ternario **(? :)** es una forma abreviada de una declaración **if-else**.

```
let resultado = (a > b) ? 'a es mayor' : 'b es mayor';
```

4.7. Operador de Coma

El operador de coma (`,`) permite evaluar múltiples expresiones y devuelve el valor de la última.

```
let x = (a = 1, b = 2, a + b); // 3
```

4.8. Operador de Tipo (`typeof`)

El operador typeof devuelve una cadena que indica el tipo de un operando.

```
let tipo = typeof a; // 'number'
```

4.9. Operador de Eliminación (delete)

El operador `delete` elimina una propiedad de un objeto.

```
let obj = { nombre: 'John', edad: 30 };
delete obj.edad; // obj es ahora { nombre: 'John' }
```

4.10. Operador de Instancia (instanceof)

El operador `instanceof` verifica si un objeto es una instancia de una clase o constructor específico.

```
let fecha = new Date();
let esFecha = fecha instanceof Date; // true
```

4.11. Operador de Propiedad (in)

El operador in verifica si una propiedad existe en un objeto. Es muy útil para buscar información dentro del objeto que bien pudiera ser un array sin necesidad de recorrerlo explícitamente.

```
let tieneNombre = 'nombre' in obj; // true
```

Este operador proporciona una amplia gama de funcionalidades para manipular y trabajar con datos en JavaScript, permitiendo escribir código más eficiente y efectivo.

5. Estructuras de Control.

Las estructuras de control permiten que un programa tome decisiones y ejecute diferentes bloques de código en función de ciertas condiciones. En JavaScript, las principales estructuras de control incluyen condicionales y bucles. A continuación, se describen en detalle con ejemplos prácticos.

5.2. Condicionales

Las declaraciones condicionales se utilizan para realizar diferentes acciones en función de diferentes condiciones lo que permiten desviar el flujo del programa en un sentido o el contrario dependiendo de una condición lógica.

a) **if...else**: La declaración `if...else` se utiliza para ejecutar un bloque de código si una condición es verdadera, y otro bloque si es falsa.

```
let edad = 18;
if (edad >= 18) {
    console.log("Eres mayor de edad.");
} else {
    console.log("Eres menor de edad.");
}

// Salida: Eres mayor de edad.
```

b) **else if**: La declaración `else if` permite comprobar múltiples condiciones. Si la primera condición es falsa, se verifica la siguiente.

```
let nota = 8.5;

if (nota >= 9) {
    console.log("Tu calificación es Sobresaliente.");
} else if (nota >= 8) {
    console.log("Tu calificación es Notable Alto.");
} else if (nota >= 7) {
    console.log("Tu calificación es Notable Bajo.");
} else if (nota >= 6) {
    console.log("Tu calificación es Bien.");
} else {
    console.log("Tu calificación es Suspenso.");
}

// Salida: Tu calificación es B.
```

c) **switch:** La declaración `switch` se utiliza para realizar múltiples comparaciones de valores. Es una alternativa más ordenada cuando se tienen muchas condiciones que evaluar.

```
let dia = 3;
let nombreDia;

switch (dia) {
    case 1:
        nombreDia = "Lunes";
        break;
    case 2:
        nombreDia = "Martes";
        break;
    case 3:
        nombreDia = "Miércoles";
        break;
    case 4:
        nombreDia = "Jueves";
        break;
    case 5:
        nombreDia = "Viernes";
        break;
    case 6:
        nombreDia = "Sábado";
        break;
    case 7:
        nombreDia = "Domingo";
        break;
    default:
        nombreDia = "Día inválido";
}

console.log(nombreDia); // Salida: Miércoles
```

5.3. Bucles

Los bucles permiten ejecutar un bloque de código varias veces. Actúan como un contador en el que en cada paso podemos ejecutar un bloque de acciones.

a) **for:** El bucle for se utiliza para ejecutar un bloque de código un número específico de veces.

```
for (let i = 0; i < 5; i++) {
    console.log(`Iteración número: ${i}`);
```

```
}

// Salida:
// Iteración número: 0
// Iteración número: 1
// Iteración número: 2
// Iteración número: 3
// Iteración número: 4
```

En el siguiente código vemos un ejemplo un poco más avanzado que nos muestra la tabla de multiplicar del número 3:

```
  let resultado = "";
  let numero = 3;

    // Bucle para calcular la multiplicación del número con los valores
del 1 al 10
    for (let i = 1; i <= 10; i++) {
        // Añadir cada resultado a la cadena, seguido de un salto de
línea
        resultado += numero + " x " + i + " = " + (numero * i) + "\n";
    }

    // Devolver la cadena completa con los resultados
    console.log(resultado);
```

b) while: Un bucle **while** se utiliza para repetir un bloque de código mientras se cumpla una condición determinada. La condición se evalúa antes de cada iteración del bucle, y si la condición es verdadera, el bloque de código se ejecuta. Si la condición es falsa, el bucle se detiene y el control del programa pasa a la siguiente instrucción después del bucle.

```
let contador = 0;

while (contador < 5) {
    console.log(`Contador: ${contador}`); //Interpolación de cadenas
    contador++;
}

// Salida:
// Contador: 0
// Contador: 1
// Contador: 2
```

```
// Contador: 3
// Contador: 4
```

c) **do...while:** Un bucle **do while** es similar a un bucle while, pero con una diferencia clave: la condición se evalúa después de ejecutar el bloque de código, lo que garantiza que el bloque se ejecute al menos una vez, independientemente de si la condición es verdadera o falsa al inicio.

```
let numero = 0;

do {
    console.log(`Número: ${numero}`);
    numero++;
} while (numero < 5);

// Salida:
// Número: 0
// Número: 1
// Número: 2
// Número: 3
// Número: 4
```

d) **Ejemplos Avanzados**

En aplicaciones del mundo real, a menudo se combinan condicionales y bucles para resolver problemas más complejos.

Ejemplo: FizzBuzz

El problema FizzBuzz es un clásico en las entrevistas de programación. El objetivo es imprimir los números del 1 al 100, pero:

- Para múltiplos de 3, imprimir "Fizz" en lugar del número.
- Para múltiplos de 5, imprimir "Buzz" en lugar del número.
- Para múltiplos de ambos 3 y 5, imprimir "FizzBuzz".

```
for (let i = 1; i <= 100; i++) {
    if (i % 3 === 0 && i % 5 === 0) {
        console.log("FizzBuzz");
    } else if (i % 3 === 0) {
        console.log("Fizz");
    } else if (i % 5 === 0) {
        console.log("Buzz");
    } else {
        console.log(i);
```

```
    }
}

// Salida:
// 1
// 2
// Fizz
// 4
// Buzz
// ...
// FizzBuzz
```

Ejemplo: Encontrar Números Primos
Un número primo es un número mayor que 1 que solo es divisible por 1 y por sí mismo. El siguiente ejemplo encuentra todos los números primos entre 1 y 50.

```
for (let num = 2; num <= 50; num++) {
    let esPrimo = true;

    for (let divisor = 2; divisor < num; divisor++) {
        if (num % divisor === 0) {
            esPrimo = false;
            break;
        }
    }

    if (esPrimo) {
        console.log(num);
    }
}
```

// Salida:
// 2
// 3
// 5
// 7
// 11
// 13
// 17
// 19
// 23
// 29
// 31
// 37
// 41
// 43

```
// 47
```

e) **break:** El comando `break` termina el bucle inmediatamente.

```
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break;
    }
    console.log(i);
}
```

```
// Salida:
// 0
// 1
// 2
// 3
// 4
```

f) **continue:** El comando `continue` salta a la siguiente iteración del bucle.

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue;
    }
    console.log(i);
}
```

```
// Salida:
// 1
// 3
// 5
// 7
// 9
```

5.4. Ejercicios

Ejercicio 1: Números pares e impares

Escribe una script llamado contarParesImpares que tome un array de números y devuelva un objeto con dos propiedades: pares e impares. Estas propiedades deben contener la cantidad de números pares e impares en el array, respectivamente.

Ejercicio 2: Números primos

Escribe un script llamado esPrimo que tome un número y devuelva true si el número es primo y false en caso contrario. Un número primo es un número mayor que 1 que solo tiene dos divisores: 1 y sí mismo.

Ejercicio 3: FizzBuzz

Escribe una función llamada fizzBuzz que tome un número n y devuelva un array con los números del 1 al n pero para múltiplos de 3, debe añadir "Fizz" en lugar del número, y para los múltiplos de 5, debe añadir "Buzz". Para números que son múltiplos de ambos, debe añadir "FizzBuzz".

Ejercicio 4: Generar secuencia de números

Escribe una función llamada generarSecuencia que tome dos números, inicio y fin, y devuelva un array con los números en el rango desde `inicio` hasta `fin` (inclusive). Si `inicio` es mayor que `fin`, el array debe estar vacío.

Ejercicio 5: Encontrar el máximo común divisor (MCD)

Escribe un script llamado encontrarMCD que tome dos números y devuelva su máximo común divisor (MCD). El MCD de dos números es el número más grande que los divide a ambos sin dejar residuos.

Ejercicio 6: Tabla de multiplicar

Escribe un script llamado tabla que nos devuelva la tabla de multiplicar de un número solicitado al usuario. Podéis usar el método prompt aunque no es muy aconsejable.

6. Funciones

Las funciones son bloques de código diseñados para realizar una tarea específica. En JavaScript, las funciones permiten reutilizar código, modularizar aplicaciones y hacer el código más limpio y mantenible. Vamos a explorar las características y usos de las funciones en JavaScript.

6.1. Definir una función

En JavaScript, puedes definir funciones de varias maneras. La forma más común de definir una función es utilizando la declaración de función. Se define usando la palabra clave **function**, seguida del nombre de la función, los parámetros entre paréntesis y un bloque de código entre llaves **{}.**

```
function saludar(nombre) {
    console.log("Hola, " + nombre + "!");
}

// Llamar a la función
saludar("Juan"); // Salida: Hola, Juan!
```

Una función también puede ser definida como una expresión. En este caso, la función es asignada a una variable. Este tipo de función puede ser anónima (sin nombre) o nombrada.

```
const multiplicar = function(a, b) {
    return a * b;
};

// Llamar a la función
console.log(multiplicar(3, 4)); // Salida: 12
```

Función Flecha (Arrow Function)

Las funciones flecha, introducidas en ES6, proporcionan una sintaxis más concisa para definir funciones. También tienen un comportamiento diferente con respecto al objeto `this`.

```
const sumar = (a, b) => a + b;

// Llamar a la función
console.log(sumar(5, 6)); // Salida: 11
```

Función Anónima Immediately Invoked Function Expression (IIFE)

Una IIFE es una función que se define y se ejecuta inmediatamente.

```
(function() {
    console.log("Esta función se ejecuta inmediatamente.");
})();
```

6.2. Parámetros y Argumentos

Las funciones en JavaScript pueden aceptar parámetros que actúan como variables locales dentro de la función.

```
function saludar(nombre, saludo = "Hola") {
    console.log(`${saludo}, ${nombre}!`);
}

saludar("Ana"); // Salida: Hola, Ana!
saludar("Ana", "Buenos días"); // Salida: Buenos días, Ana!
```

Parámetros por Defecto: Los parámetros pueden tener valores por defecto, como en el ejemplo anterior donde **saludo** tiene un valor predeterminado de **"Hola"**.

6.3. Valor de Retorno

Una función puede devolver un valor utilizando la palabra clave `return`. Si no se especifica un valor de retorno, la función devolverá `undefined`.

```
function calcularCuadrado(x) {
    return x * x;
}

let resultado = calcularCuadrado(5);
console.log(resultado); // Salida: 25
```

6.4. Funciones como Objetos de Primera Clase

En JavaScript, decir que una **función es un "objeto de primera clase"** significa que las funciones son tratadas como cualquier otro valor u objeto. Esto permite a las funciones ser usadas y manipuladas de diversas maneras, como asignarlas a variables, pasarlas como argumentos a otras funciones, devolverlas desde funciones, y almacenarlas en estructuras de datos. Este es un concepto fundamental en lenguajes de programación funcionales.

En JavaScript, las funciones pueden ser tratadas como cualquier otro valor. En este sentido podemos:

- Asignar funciones a variables

- Pasar funciones como argumentos a otras funciones

- Retornar funciones desde otras funciones

Asignación a Variables

```
const saludar = function(nombre) {
    console.log("Hola, " + nombre);
};

saludar("Luis"); // Salida: Hola, Luis
```

Pasar Funciones como Argumentos

```
function operar(a, b, operacion) {
    return operacion(a, b);
}

const suma = (x, y) => x + y;
const resta = (x, y) => x - y;

console.log(operar(5, 3, suma)); // Salida: 8
console.log(operar(5, 3, resta)); // Salida: 2
```

Retornar Funciones desde Otras Funciones

```
function crearMultiplicador(factor) {
    return function(numero) {
        return numero * factor;
    };
}

const duplicar = crearMultiplicador(2);
console.log(duplicar(5)); // Salida: 10
```

6.4. Ámbito (Scope) y Cierre (Closures)

a) **Ámbito**: El ámbito determina la accesibilidad de variables en diferentes partes del código. En JavaScript, el ámbito de las variables puede ser global o local.

```
let globalVar = "Soy global";

function miFuncion() {
    let localVar = "Soy local";
    console.log(globalVar); // Acceso a variable global
    console.log(localVar);  // Acceso a variable local
```

```
}

miFuncion();
console.log(globalVar); // Acceso a variable global
console.log(localVar);  // Error: localVar no está definido
```

b) **Cierre (Closure):** Un cierre es una función que recuerda el entorno en el que fue creada, lo que permite acceder a variables de ese entorno incluso después de que la función externa haya terminado de ejecutarse.

```
function crearContador() {
    let contador = 0;
    return function() {
        contador += 1;
        return contador;
    };
}

const contador = crearContador();
console.log(contador()); // Salida: 1
console.log(contador()); // Salida: 2
console.log(contador()); // Salida: 3
```

6.5. Funciones Recursivas

Una función recursiva es una función que se llama a sí misma. La recursión es útil para resolver problemas que pueden ser divididos en subproblemas similares.

```
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

console.log(factorial(5)); // Salida: 120
```

6.6. Función flecha (=>)

La función flecha, introducida en ECMAScript 6 (ES6), es una sintaxis moderna y concisa para definir funciones en JavaScript. A diferencia de las funciones tradicionales, las funciones flecha utilizan una sintaxis simplificada que omite la palabra clave **function** y emplea el símbolo **"=>"** para separar los parámetros del cuerpo de la función.

La sintaxis básica de una función flecha es:

```
const nombreFuncion = (param1, param2) => {
    // Cuerpo de la función
    return resultado;
};
```

Si la función solo tiene un parámetro, se pueden omitir los paréntesis:

```
const nombreFuncion = param => {
    // Cuerpo de la función
    return resultado;
};
```

Y si la función realiza una única operación y devuelve un resultado, se pueden omitir las llaves y la palabra `return`:

```
const sumar = (a, b) => a + b;
```

Esta simplicidad hace que las funciones flecha sean ideales para funciones pequeñas y expresiones cortas.

Una de las características más importantes y diferenciadoras de las funciones flecha es cómo manejan el contexto de this. El objeto this en JavaScript hace referencia al contexto de ejecución actual, es decir, al objeto al que pertenece el código que está siendo ejecutado. El valor de this varía dependiendo de dónde y cómo se utilice. Es dinámico y puede cambiar en diferentes situaciones.En las funciones tradicionales, this se refiere al objeto que invoca la función, lo cual puede cambiar dependiendo de cómo y dónde se llama la función. Sin embargo, en una función flecha, this se fija en el contexto en el que la función fue definida, no en el contexto en el que se ejecuta. Esto hace que las funciones flecha sean particularmente útiles en el manejo de callbacks o funciones dentro de métodos de clase, donde mantener el contexto de this es crucial.

A pesar de sus ventajas, las funciones flecha tienen algunas limitaciones. Por ejemplo, no pueden ser utilizadas como constructores, ya que no tienen su propio this ni el prototipo (prototype). Tampoco poseen un objeto `arguments` local, lo que puede ser una limitación si se necesita acceso a todos los argumentos pasados a la función.

6.6. Funciones matemáticas del lenguaje.

A continuación se muestran 10 funciones matemáticas que vienen incluidas en el propio lenguaje sin necesidad de cargar librerías externas. Existen muchas más que irás descubriendo a medida que ganes experiencia como programador.

a) **Math.random():** Genera un número pseudoaleatorio entre 0 (inclusive) y 1 (exclusive).

```
let randomNum = Math.random();
console.log(randomNum); // Ejemplo: 0.7231747980472979
```

b) **Math.floor():** Redondea un número hacia abajo al entero más cercano.

```
let flooredNum = Math.floor(5.95);
console.log(flooredNum); // 5
```

c) **Math.ceil():** Redondea un número hacia arriba al entero más cercano.

```
let ceiledNum = Math.ceil(5.05);
console.log(ceiledNum); // 6
```

d) **Math.sqrt():** Devuelve la raíz cuadrada de un número.

```
let sqrtNum = Math.sqrt(64);
console.log(sqrtNum); // 8
```

e) **Math.max():** Devuelve el mayor de los números pasados como argumentos.

```
let maxNum = Math.max(1, 3, 2);
console.log(maxNum); // 3
```

f) **Math.min():** Devuelve el menor de los números pasados como argumentos.

```
let minNum = Math.min(1, 3, 2);
console.log(minNum); // 1
```

g) **Math.pow():** Devuelve la base elevada al exponente, es decir, base^exponente.

```
let power = Math.pow(2, 3);
console.log(power); // 8
```

h) **Math.abs():** Devuelve el valor absoluto de un número.

```
let absNum = Math.abs(-5);
console.log(absNum); // 5
```

i) **Math.round():** Redondea un número al entero más cercano.

```
let roundedNum = Math.round(5.5);
```

```
console.log(roundedNum); // 6
```

j) **Math.log():** Devuelve el logaritmo natural (base e) de un número.

```
let logNum = Math.log(1);
console.log(logNum); // 0
```

Ejemplos de Uso en un Programa

```
Vamos a combinar todas estas funciones en un ejemplo que genere 10
números aleatorios, realice diversas operaciones matemáticas sobre
ellos, y encuentre valores específicos.

let randomNumbers = [];

// Generar 10 números aleatorios entre 0 y 100
for (let i = 0; i < 10; i++) {
    let randomNum = Math.random() * 100;
    randomNumbers.push(randomNum);
}

console.log("Números aleatorios:", randomNumbers);

// Redondear hacia abajo
let flooredNumbers = randomNumbers.map(num => Math.floor(num));
console.log("Redondeados hacia abajo:", flooredNumbers);

// Redondear hacia arriba
let ceiledNumbers = randomNumbers.map(num => Math.ceil(num));
console.log("Redondeados hacia arriba:", ceiledNumbers);

// Calcular la raíz cuadrada
let sqrtNumbers = randomNumbers.map(num => Math.sqrt(num));
console.log("Raíces cuadradas:", sqrtNumbers);

// Encontrar el número mayor y menor
let maxNumber = Math.max(...randomNumbers);
let minNumber = Math.min(...randomNumbers);

console.log("Número mayor:", maxNumber);
console.log("Número menor:", minNumber);

// Calcular potencias
let poweredNumbers = randomNumbers.map(num => Math.pow(num, 2));
console.log("Potencias al cuadrado:", poweredNumbers);
```

```
// Calcular valores absolutos (simulado con números negativos)
let negativeNumbers = randomNumbers.map(num => num * -1);
let absNumbers = negativeNumbers.map(num => Math.abs(num));
console.log("Valores absolutos:", absNumbers);

// Redondear al número entero más cercano
let roundedNumbers = randomNumbers.map(num => Math.round(num));
console.log("Redondeados:", roundedNumbers);

// Calcular logaritmos naturales
let logNumbers = randomNumbers.map(num => Math.log(num));
console.log("Logaritmos naturales:", logNumbers);
```

En este ejemplo:

- Math.random() se utiliza para generar números aleatorios.

- Math.floor() y Math.ceil() se usan para redondear esos números hacia abajo y hacia arriba, respectivamente.

- Math.sqrt() calcula la raíz cuadrada de cada número.

- Math.max() y Math.min() encuentran el número mayor y menor de la lista.

- Math.pow() eleva cada número al cuadrado.

- Math.abs() se utiliza para obtener los valores absolutos de una lista de números negativos.

- Math.round() redondea cada número al entero más cercano.

- Math.log() calcula el logaritmo natural de cada número.

Estas funciones proporcionan una amplia gama de operaciones matemáticas que pueden ser útiles en diversas situaciones de programación.

6.6. Ejercicios

Ejercicio 1: Operaciones con arrays

Escribe una función llamada **realizarOperaciones** que tome un array de números y una cadena de texto que indique una operación ("sumar", "restar", "multiplicar", "dividir"). La función debe aplicar la operación a todos los números del array y devolver el resultado. Si la operación es "sumar", debe sumar todos los números; si es "restar", debe restarlos; y así sucesivamente.

Ejercicio 2: Palíndromos

Escribe una función llamada **esPalindromo** que tome una cadena de texto y determine si es un palíndromo (una cadena que se lee igual de adelante hacia atrás). La función debe devolver true si es un palíndromo y false en caso contrario. Ignora los espacios, signos de puntuación y diferencias entre mayúsculas y minúsculas.

Ejercicio 3: Fibonacci

Escribe una función llamada **fibonacci** que tome un número n y devuelva un array con los primeros n números de la secuencia de Fibonacci. La secuencia de Fibonacci comienza con 0 y 1, y cada número subsiguiente es la suma de los dos anteriores.

Ejercicio 4: Contar palabras

Escribe una función llamada **contarPalabras** que tome una cadena de texto y devuelva un objeto donde las claves son las palabras y los valores son el número de veces que aparece cada palabra en la cadena. Ignora la diferencia entre mayúsculas y minúsculas y los signos de puntuación.

Ejercicio 5: Ordenar objetos por propiedad

Escribe una función llamada **ordenarPorPropiedad** que tome un array de objetos y una cadena de texto que indica una propiedad. La función debe devolver el array de objetos ordenado por esa propiedad. Por ejemplo, si los objetos tienen una propiedad `nombre`, y la cadena de texto es "nombre", la función debe ordenar los objetos alfabéticamente por nombre.

7. Arrays en JavaScript

Los arrays en JavaScript son estructuras de datos que permiten almacenar una colección de elementos. Los elementos de un array pueden ser de cualquier tipo de datos, incluidos números, cadenas, objetos, y otros arrays. Los arrays son muy útiles para almacenar y manipular listas de datos.

7.1. Creación de Arrays

Puedes crear arrays de varias maneras en JavaScript:

a) **Notación de Literales de Array:** La forma más común de crear un array es utilizando la notación de corchetes `[]`.

```
const frutas = ["manzana", "banana", "cereza"];
console.log(frutas); // Salida: ["manzana", "banana", "cereza"]
```

b) **Constructor Array():** También puedes crear un array utilizando el constructor Array().

```
const numeros = new Array(1, 2, 3, 4, 5);
console.log(numeros); // Salida: [1, 2, 3, 4, 5]
```

7.2. Acceso a Elementos

Los elementos de un array se acceden mediante índices, que comienzan en 0.

```
const colores = ["rojo", "verde", "azul"];
console.log(colores[0]); // Salida: rojo
console.log(colores[2]); // Salida: azul
```

Puedes modificar un elemento de un array asignando un nuevo valor al índice correspondiente.

```
const animales = ["perro", "gato", "pájaro"];
animales[1] = "conejo";
console.log(animales); // Salida: ["perro", "conejo", "pájaro"]
```

7.3. Recorrido de arrays

Un array se puede recorrer con un bucle for que vaya desde 0 hasta la longitud del array menos 1. En casa paso del bucle haremos algo con el elemento seleccionado. En el siguiente ejemplo vamos a sumar todos los elementos de un array de números enteros.

```
var numeros = [1, 2, 3, 4, 5];
var suma = 0;

for (var i = 0; i < numeros.length; i++) {
    suma += numeros[i]; // Sumar cada elemento al total
}

console.log("La suma de los números es: " + suma);
```

Sin embargo, cuando se trata de arrays, Javascript nos ofrece métodos específicos para realizar estos recorridos sin preocuparnos por la longitud del array. Las dos variaciones más interesantes son las siguiente:

a) Usando **for...of: E**l bucle `for...of` te permite iterar directamente sobre los valores de un array sin preocuparte por los índices.

```
var numeros = [1, 2, 3, 4, 5];
var suma = 0;

for (var numero of numeros) {
    suma += numero; // Sumar cada número al total
}

console.log("La suma de los números es: " + suma);
```

b) Usando **forEach**: El método forEach permite ejecutar una función para cada elemento de un array. A continuación, un ejemplo con el mismo array:

```
var numeros = [1, 2, 3, 4, 5];
var suma = 0;

numeros.forEach(function(numero) {
    suma += numero; // Sumar cada número al total
});

console.log("La suma de los números es: " + suma);
```

Por último comentar una variante con la función flecha para el método forEach

```
var numeros = [1, 2, 3, 4, 5];
var suma = 0;

numeros.forEach(numero => {
    suma += numero; // Sumar cada número al total
```

```
});

console.log("La suma de los números es: " + suma);
```

En este ejemplo, la función flecha (=>) utiliza una sintaxis más concisa para escribir funciones en JavaScript. En lugar de escribir `function(numero) {...}`, utilizamos numero => {...}.

Si también quieres utilizar el índice del elemento, puedes hacerlo así:

```
var numeros = [1, 2, 3, 4, 5];
var suma = 0;

numeros.forEach((numero, index) => {
    suma += numero; // Sumar cada número al total
    console.log("Índice " + index + ": " + numero);
});

console.log("La suma de los números es: " + suma);
```

7.4. Métodos Comunes de Arrays

JavaScript proporciona una variedad de métodos útiles para trabajar con arrays:

a) **push():** Añade uno o más elementos al final del array.

```
const verduras = ["zanahoria", "brócoli"];
verduras.push("espinaca");
console.log(verduras); // Salida: ["zanahoria", "brócoli", "espinaca"]
```

b) **pop():** Elimina el último elemento del array y lo devuelve.

```
const colores = ["rojo", "verde", "azul"];
const ultimoColor = colores.pop();
console.log(ultimoColor); // Salida: azul
console.log(colores); // Salida: [“rojo”,"verde"]
```

c) **shift()**: Elimina el primer elemento del array y lo devuelve.

```
const colores = ["rojo", "verde", "azul"];
const primerColor = colores.shift();
console.log(primerColor); // Salida: rojo
console.log(colores); // Salida: ["verde", "azul"]
```

d) **unshift()**: Añade uno o más elementos al principio del array.

```
const numeros = [2, 3, 4];
numeros.unshift(1);
console.log(numeros); // Salida: [1, 2, 3, 4]
```

e) **splice()**: Añade o elimina elementos en cualquier posición del array.

```
const animales = ["perro", "gato", "pájaro"];
animales.splice(1, 1, "conejo", "hamster");
// A partir del índice 1, elimina 1 elemento y añade "conejo" y
"hamster"
console.log(animales); // Salida: ["perro", "conejo", "hamster",
"pájaro"]
```

f) **slice():** Crea una copia superficial de una parte del array.

```
const frutas = ["manzana", "banana", "cereza", "kiwi"];
const algunasFrutas = frutas.slice(1, 3); // Desde el índice 1 hasta el
3 (excluido)
console.log(algunasFrutas); // Salida: ["banana", "cereza"]
```

g) **split():** se utiliza para dividir una cadena de texto en un array de substrings, utilizando un separador que tú defines. Este método es útil cuando necesitas separar un string en partes más pequeñas, como palabras, letras o elementos específicos dentro de la cadena.

Por ejemplo, imaginemos que tenemos una cadena de palabras dentro de una frase y queremos dividirla en palabras:

```
let texto = "Hola mundo";
let palabras = texto.split(" ");
console.log(palabras); // ["Hola", "mundo"]
```

Imaginemos ahora que queremos dividir una cadena en caracteres individuales:

```
let palabra = "JavaScript";
let caracteres = palabra.split("");
console.log(caracteres); // ["J", "a", "v", "a", "S", "c", "r", "i",
"p", "t"]
```

h) **join():** es la función inversa de split() en JavaScript. Mientras que split() convierte una cadena en un array, join() toma un array y lo convierte en una cadena, uniendo sus elementos con un separador que tú defines.

```
    let palabras = ["Hola", "mundo"];
    let texto = palabras.join(" ");
    console.log(texto); // "Hola mundo"
```

i) **map():** Crea un nuevo array con los resultados de la función aplicada a cada elemento.

```
const numeros = [1, 2, 3];
const duplicados = numeros.map(function(numero) {
    return numero * 2;
});
console.log(duplicados); // Salida: [2, 4, 6]
```

j) **filter()**: Crea un nuevo array con todos los elementos que pasan la prueba implementada por la función proporcionada.

```
const numeros = [1, 2, 3, 4, 5];
const pares = numeros.filter(function(numero) {
    return numero % 2 === 0;
});
console.log(pares); // Salida: [2, 4]
```

k) **reduce()**: Aplica una función contra un acumulador y cada elemento del array (de izquierda a derecha) para reducirlo a un único valor.

```
const numeros = [1, 2, 3, 4];
const suma = numeros.reduce(function(acumulador, numero) {
    return acumulador + numero;
}, 0);
console.log(suma); // Salida: 10
```

7.5. Arrays Multidimensionales

Los arrays también pueden contener otros arrays, lo que permite crear estructuras de datos más complejas.

```
const matriz = [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
];
console.log(matriz[1][2]); // Salida: 6
```

7.6. Métodos Adicionales

a) **concat():** Une dos o más arrays.

```
const a = [1, 2];
    const b = [3, 4];
    const c = a.concat(b);
    console.log(c); // Salida: [1, 2, 3, 4]
```

b) **find():** Devuelve el primer elemento que cumple con la condición proporcionada.

```
const numeros = [1, 2, 3, 4, 5];
    const mayorDeTres = numeros.find(function(numero) {
        return numero > 3;
    });
    console.log(mayorDeTres); // Salida: 4
```

c) **some():** Comprueba si al menos un elemento cumple con la condición proporcionada.

```
const numeros = [1, 2, 3, 4, 5];
    const hayMayorDeCuatro = numeros.some(function(numero) {
        return numero > 4;
    });
    console.log(hayMayorDeCuatro); // Salida: true
```

d) **every():** Comprueba si todos los elementos cumplen con la condición proporcionada.

```
const numeros = [1, 2, 3, 4];
    const todosMenoresDeCinco = numeros.every(function(numero) {
        return numero < 5;
    });
    console.log(todosMenoresDeCinco); // Salida: true
```

Ejemplo completo con arrays: En este ejemplo vamos a mostrar una tabla con una paleta de colores donde vayamos degradando cada color mediante el formato RGB.

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

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Paleta RGB</title>
    </head>
<body>

    <table id="paleta"></table>

    <script>
        // Definir los arrays de componentes de color RGB
        var r = ["00", "33", "66", "99", "CC", "FF"];
        var g = ["00", "33", "66", "99", "CC", "FF"];
        var b = ["00", "33", "66", "99", "CC", "FF"];

        // Obtener la tabla por su ID
        const tabla = document.getElementById('paleta');

        // Iterar sobre todos los valores de r, g y b para crear las
combinaciones
        r.forEach(red => {
            g.forEach(green => {
                const fila = tabla.insertRow(); // Crear una nueva fila
                b.forEach(blue => {
                    const color = `#${red}${green}${blue}`; // Combinar
los valores en un código de color hexadecimal
                    const celda = fila.insertCell(); // Crear una nueva
celda
                    celda.style.backgroundColor = color; // Asignar el
color de fondo
                    celda.textContent = color; // Mostrar el código de
color
                });
            });
        });
    </script>

</body>
</html>
```

1. Arrays de Componentes RGB:

- Los arrays `r`, `g`, y `b` contienen los valores de componentes en formato hexadecimal que representan diferentes intensidades de rojo, verde y azul respectivamente. Cada array

tiene 6 valores (`"00"`, `"33"`, `"66"`, `"99"`, `"CC"`, `"FF"`), lo que permite crear 216 combinaciones de colores.

2. Generación de la Tabla:

- Se utiliza una estructura de bucles anidados (`forEach`) para iterar sobre cada combinación posible de `r`, `g`, y `b`.
- Por cada combinación de `r`, `g`, y `b`, se crea un código de color hexadecimal que se asigna como fondo de una celda (`td`), y se inserta en la tabla (`<table>`).
- El código de color también se muestra como texto dentro de cada celda.

3. Visualización de Colores:

- El resultado es una tabla donde cada celda representa un color diferente en función de las combinaciones de los valores RGB. El código hexadecimal de cada color se muestra en la celda para que sea fácil de identificar.

Este código genera una tabla grande con 216 celdas, donde cada celda muestra un color único basado en las combinaciones de los valores de los arrays `r`, `g`, y `b`. La tabla tiene tantas filas como combinaciones de `r` y `g`, y cada fila tiene 6 celdas (una por cada valor de `b`).

7.7. Ejercicios

Ejercicio 1: Filtrar y transformar un array

Escribe una función llamada `filtrarYTransformar` que tome un array de números y devuelva un nuevo array que contenga solo los números pares, cada uno multiplicado por 2.

Ejercicio 2: Intersección de arrays

Escribe una función llamada intersección que tome dos arrays y devuelva un nuevo array con los elementos que están presentes en ambos arrays.

Ejercicio 3: Matriz transpuesta

Escribe una función llamada transponerMatriz que tome una matriz (array de arrays) y devuelva su transpuesta. La transpuesta de una matriz se obtiene intercambiando filas por columnas.

Ejercicio 4: Eliminar duplicados

Escribe una función llamada eliminarDuplicados que tome un array y devuelva un nuevo array con los elementos únicos (sin duplicados) del array original.

Ejercicio 5: Aplanar un array anidado

Escribe una función llamada `aplanar` que tome un array anidado (un array que contiene otros arrays) y devuelva un nuevo array "aplanado", es decir, un array que contiene todos los elementos del array original pero sin ninguna anidación.

Ejercicio 6: Contar Ocurrencias de un Elemento

Implementa una función que tome un array y un valor como argumentos, y devuelva el número de veces que ese valor aparece en el array.

Ejemplo: cuentaOcurrencias([1, 2, 3, 1, 4, 1], 1) debería devolver 3.

Ejercicio 7: Buscar el Número Máximo

Crea una función que reciba un array de números y retorne el valor máximo. No se permite usar la función Math.max(). Si el array está vacío, devuelve “Vacío”.

Ejemplo: buscarMax([10, 20, 5, 40, 25]) debería devolver 40.

8. Objetos en JavaScript

Los objetos en JavaScript son una estructura fundamental para representar entidades y almacenar datos en pares clave-valor. Son versátiles y se utilizan ampliamente para organizar y manipular datos.

8.1. Creación de Objetos

Puedes crear objetos de diferentes maneras:

a) **Notación de Literales de Objeto:** La forma más común de crear un objeto es usando la notación de literales de objeto.

```
const persona = {
    nombre: "Ana",
    edad: 30,
    ciudad: "Madrid"
};
```

b) **Constructor Object()**

```
const persona = new Object();
persona.nombre = "Luis";
persona.edad = 25;
persona.ciudad = "Barcelona";
```

c) **Constructor de Funciones:** Puedes usar una función constructora para crear instancias de objetos.

```
function Persona(nombre, edad, ciudad) {
    this.nombre = nombre;
    this.edad = edad;
    this.ciudad = ciudad;
}

const persona1 = new Persona("Maria", 28, "Valencia");
const persona2 = new Persona("Carlos", 35, "Sevilla");
```

8.2. Propiedades y Métodos

Los objetos pueden tener **propiedades** que almacenan valores.

```
const coche = {
    marca: "Toyota",
```

```
    modelo: "Corolla",
    año: 2020
};

console.log(coche.marca); // Salida: Toyota
```

Los **métodos** son funciones asociadas a un objeto.

```
const coche = {
    marca: "Toyota",
    modelo: "Corolla",
    año: 2020,
    mostrarDetalles: function() {
        return `${this.marca} ${this.modelo} (${this.año})`;
    }
};

console.log(coche.mostrarDetalles()); // Salida: Toyota Corolla (2020)
```

8.3. Acceso y Modificación de Propiedades

En JavaScript, puedes acceder y modificar las propiedades de un objeto de dos maneras principales: mediante notación de punto (`.`) y notación de corchetes (`[]`). Ambas formas son útiles y tienen aplicaciones ligeramente diferentes dependiendo del contexto.

a) **Notación de Punto (.)**: La notación de punto es la forma más común de acceder a las propiedades de un objeto. Se utiliza cuando conoces el nombre exacto de la propiedad y este es un identificador válido en JavaScript (es decir, no contiene espacios ni caracteres especiales).

```
const coche = {
    marca: "Toyota",
    modelo: "Corolla",
    año: 2020
};

// Acceder a propiedades
console.log(coche.marca); // Salida: Toyota
console.log(coche.modelo); // Salida: Corolla

// Modificar propiedades
coche.año = 2021;
console.log(coche.año); // Salida: 2021
```

b) **Notación de Corchetes ([]):** La notación de corchetes se utiliza cuando el nombre de la propiedad se almacena en una variable o cuando el nombre de la propiedad no es un identificador válido (como nombres con espacios o caracteres especiales). También es útil cuando necesitas acceder a propiedades cuyos nombres se generan dinámicamente.

```
const persona = {
    nombre: "Ana",
    edad: 30,
    "dirección completa": "Calle Mayor 10"
};

// Acceder a propiedades
console.log(persona["nombre"]); // Salida: Ana
console.log(persona["dirección completa"]); // Salida: Calle Mayor 10

// Modificar propiedades
const propiedad = "edad";
persona[propiedad] = 31;
console.log(persona.edad); // Salida: 31
```

8.4. Adición y Eliminación de Propiedades

Puedes añadir nuevas propiedades a un objeto en cualquier momento utilizando cualquiera de las dos notaciones mencionadas.

```
const libro = {
    título: "JavaScript para Todos",
    autor: "Juan Pérez"
};

// Añadir una nueva propiedad
libro.año = 2024;
libro["editorial"] = "Editorial Tech";

console.log(libro);
// Salida:
// {
//   título: "JavaScript para Todos",
//   autor: "Juan Pérez",
//   año: 2024,
//   editorial: "Editorial Tech"
// }
```

Para eliminar una propiedad de un objeto, se usa el operador `delete`.

```
const estudiante = {
    nombre: "Pedro",
    edad: 22,
    carrera: "Ingeniería"
};

// Eliminar una propiedad
delete estudiante.carrera;

console.log(estudiante);
// Salida: { nombre: "Pedro", edad: 22 }
```

8.5. Métodos de Objetos

Los métodos son funciones definidas dentro de un objeto que operan sobre las propiedades del mismo.

```
const coche = {
    marca: "Toyota",
    modelo: "Corolla",
    año: 2020,
    mostrarDetalles: function() {
        return `${this.marca} ${this.modelo} (${this.año})`;
    }
};

console.log(coche.mostrarDetalles()); // Salida: Toyota Corolla (2020)
```

En este ejemplo, **mostrarDetalles** es un método que utiliza **“this”** para acceder a las propiedades del objeto “coche”.

8.6. Propiedades Computadas

A veces, es útil crear nombres de propiedades dinámicamente utilizando variables. Esto se puede hacer con la notación de corchetes.

```
const clave = "nombre";
const persona = {
    [clave]: "Carlos"
};

console.log(persona.nombre); // Salida: Carlos
```

```
// Añadir más propiedades con claves dinámicas
const atributo = "edad";
persona[atributo] = 40;

console.log(persona.edad); // Salida: 40
```

8.7. Uso de Object para Manipulación Avanzada

a) **Object.keys():** Devuelve un array con las claves (propiedades) del objeto.

```
const persona = {
    nombre: "Ana",
    edad: 30,
    ciudad: "Madrid"
};

console.log(Object.keys(persona)); // Salida: ["nombre", "edad",
"ciudad"];
```

b) **Object.values():** Devuelve un array con los valores de las propiedades del objeto.

```
console.log(Object.values(persona)); // Salida: ["Ana", 30, "Madrid"]
```

c) **Object.entries():** Devuelve un array de pares `[clave, valor]` del objeto.

```
console.log(Object.entries(persona));
// Salida: [["nombre", "Ana"], ["edad", 30], ["ciudad", "Madrid"]]
```

d) **Object.assign():** Permite copiar las propiedades de uno o más objetos fuente a un objeto destino.

```
const persona1 = {
    nombre: "Luis",
    edad: 28
};

const persona2 = {
    ciudad: "Barcelona",
    país: "España"
};

const personaCompleta = Object.assign({}, persona1, persona2);
```

```
console.log(personaCompleta);
// Salida: { nombre: "Luis", edad: 28, ciudad: "Barcelona", país:
"España" }
```

Aquí tienes un ejemplo que combina el acceso y la modificación de propiedades, la adición y eliminación de propiedades, y el uso de métodos de objetos:

```
const coche = {
    marca: "Ford",
    modelo: "Mustang",
    año: 2021,
    mostrarDetalles: function() {
        return `${this.marca} ${this.modelo} (${this.año})`;
    }
};

// Acceso a propiedades
console.log(coche.marca); // Salida: Ford

// Modificación de propiedades
coche.año = 2022;
console.log(coche.mostrarDetalles()); // Salida: Ford Mustang (2022)

// Añadir una nueva propiedad
coche.color = "Rojo";

// Eliminar una propiedad
delete coche.modelo;

console.log(coche);
// Salida: { marca: "Ford", año: 2022, mostrarDetalles: [Function:
mostrarDetalles], color: "Rojo" }
```

8.8. Ejercicios

Ejercicio 1: Clase "Círculo”

Crea una **clase Círculo** que represente un círculo. La clase debe tener un atributo **radio** y métodos para calcular el área (calcularArea) y el perímetro (calcularPerimetro) del círculo. Recuerda que el valor de π (pi) es Math.PI.

Pista: El área de un círculo se calcula con la fórmula π * radio^2 y el perímetro con 2 * π * radio.

Ejercicio 2: Clase "Persona"

Crea una **clase Persona** que tenga los atributos **nombre, edad y género**. Añade un método **presentarse** que muestre un mensaje en la consola con el nombre, la edad y el género de la persona. Luego, crea varios objetos de la clase Persona y llama al método presentarse para cada uno de ellos.

Pista: El método presentarse podría devolver algo como: "Hola, me llamo Juan, tengo 25 años y soy hombre."

Ejercicio 3: Clase "Libro"

Crea una **clase Libro** que tenga los atributos **título, autor y año**. Añade un método **mostrarInfo** que muestre la información del libro en la consola. Luego, crea una **clase Biblioteca** que contenga un array de libros y métodos para agregar un nuevo libro (agregarLibro) y mostrar todos los libros (mostrarLibros) en la consola.

Pista: La clase Biblioteca debería tener un array privado que almacene los objetos Libro.

Ejercicio 4: Clase "CuentaBancaria"

Crea una clase **CuentaBancaria** con atributos **titular, saldo**, y métodos **depositar(cantidad) y retirar(cantidad)**. El método **retirar** debe verificar si hay suficiente saldo antes de realizar la operación. Si no hay suficiente saldo, debe mostrar un mensaje de error en la consola. Finalmente, implementa un método **mostrarSaldo** que imprima el saldo actual.

Pista: Usa condicionales dentro del método retirar para verificar si la cantidad a retirar es menor o igual que el saldo.

Ejercicio 5: Clase "Auto"

Crea una **clase Auto** que tenga los atributos **marca, modelo, año y kilometraje**. Implementa los métodos **encender, apagar, conducir(kilometros) y mostrarEstado**. El método conducir debe incrementar el kilometraje y el método mostrarEstado debe mostrar si el auto está encendido o apagado, así como su kilometraje actual.

Pista: El auto debe tener un estado (encendido o apagado) que se actualice con los métodos encender y apagar.

9. Eventos

9.1. Introducción a los Eventos

Los eventos en JavaScript son acciones o sucesos que ocurren en el navegador y a los cuales se puede responder mediante código. Los eventos pueden ser causados por interacciones del usuario, como hacer clic en un botón, mover el ratón, escribir en un campo de texto, o incluso cargar una página web.

JavaScript sigue el modelo de eventos basado en el DOM (Document Object Model) que veremos en el siguiente capítulo. Los elementos del DOM pueden generar varios tipos de eventos, y puedes utilizar manejadores de eventos para responder a esos eventos.

9.2. Manejadores de Eventos

a) **Métodos de Asignación de Eventos.** Existen varias maneras de asignar manejadores de eventos en JavaScript:

Atributos HTML

```
<button onclick="alert('¡Botón clickeado!')">Haz clic aquí</button>
```

Propiedades del DOM: En este caso los eventos se identifican mediante la palabra **"on"** seguida del **nombre del evento.** Más adelante veremos un listado completo de eventos.

```
<button id="miBoton">Haz clic aquí</button>
<script>
  document.getElementById('miBoton').onclick = function() {
    alert('¡Botón clickeado!');
  };
</script>
```

addEventListener

```
<button id="miBoton">Haz clic aquí</button>
<script>
  document.getElementById('miBoton').addEventListener('click',
function() {
    alert('¡Botón clickeado!');
  });
</script>
```

b) **Ventajas de addEventListener**

- Permite agregar múltiples manejadores de eventos al mismo elemento.
- Facilita la eliminación de manejadores de eventos.
- Proporciona un mejor soporte para la fase de captura y burbujeo de eventos.

9.3. Tipos de Eventos

a) Eventos del Ratón

- click: Ocurre cuando el usuario hace clic con el ratón.
- dblclick: Ocurre cuando el usuario hace doble clic con el ratón.
- mousedown: Ocurre cuando se presiona un botón del ratón.
- mouseup: Ocurre cuando se suelta un botón del ratón.
- mousemove: Ocurre cuando el ratón se mueve sobre un elemento.
- mouseenter: Ocurre cuando el puntero entra en un elemento.
- mouseleave: Ocurre cuando el puntero sale de un elemento.
- mouseover: Ocurre cuando el puntero pasa sobre un elemento o sus hijos.
- mouseout: Ocurre cuando el puntero sale de un elemento o de sus hijos.
- contextmenu: Ocurre cuando se abre el menú contextual (clic derecho).

```
<div id="area" style="width: 200px; height: 200px; background-color:
lightblue;"></div>
<script>
  const area = document.getElementById('area');
  area.addEventListener('click', () => {
    alert('Área clickeada');
  });
</script>
```

b) Eventos del Teclado

- keydown: Ocurre cuando una tecla se presiona.
- keyup: Ocurre cuando una tecla se suelta.
- keypress: Ocurre cuando se presiona una tecla (obsoleto en favor de keydown y keyup).

```
<input type="text" id="campoTexto" placeholder="Escribe algo...">
<script>
  const campoTexto = document.getElementById('campoTexto');
  campoTexto.addEventListener('keydown', (event) => {
    console.log(`Tecla presionada: ${event.key}`);
  });
</script>
```

c) Eventos de la Página

- load: Ocurre cuando el documento ha terminado de cargarse completamente (incluyendo recursos).

- DOMContentLoaded: Ocurre cuando el HTML ha sido completamente cargado y parseado, sin esperar imágenes u otros recursos.
- unload: Ocurre cuando el documento o la ventana se están descargando.
- beforeunload: Ocurre antes de que la ventana se descargue (puede usarse para mostrar confirmaciones).
- resize: Ocurre cuando se cambia el tamaño de la ventana.
- scroll: Ocurre cuando se desplaza el contenido de un elemento o la ventana.

```
<script>
  window.addEventListener('load', () => {
    alert('Página cargada');
  });
</script>
```

d) Eventos de Drag & Drop (Arrastrar y soltar)

- drag: Ocurre mientras se arrastra un elemento.
- dragstart: Ocurre al comenzar a arrastrar un elemento.
- dragend: Ocurre al terminar de arrastrar un elemento.
- dragenter: Ocurre cuando un elemento arrastrado entra en un objetivo de soltar.
- dragover: Ocurre mientras un elemento arrastrado se mueve dentro del objetivo de soltar.
- dragleave: Ocurre cuando un elemento arrastrado sale de un objetivo de soltar.
- drop: Ocurre cuando se suelta un elemento arrastrado en un objetivo de soltar.

e) Eventos de Formulario

- submit: Ocurre cuando se envía un formulario.
- reset: Ocurre cuando se reinicia un formulario.
- focus: Ocurre cuando un elemento obtiene el foco.
- blur: Ocurre cuando un elemento pierde el foco.
- input: Ocurre cuando se cambia el valor de un elemento <input>, <textarea>, o <select>.
- change: Ocurre cuando se cambia el valor de un elemento <input>, <textarea>, o <select> y se confirma.
- invalid: Ocurre cuando un campo de entrada es inválido.
- select: Ocurre cuando se selecciona texto en un <input> o <textarea>.

Existe una lista mucho más amplia de eventos pero aquí hemos querido recalcar los más utilizados. En Internet puedes encontrar listas completas y actualizadas.

9.4. Propagación de Eventos

La propagación de eventos describe cómo los eventos se transfieren desde el elemento que los generó hasta otros elementos. Hay dos fases principales:

a) **Fase de Captura:** El evento se propaga desde el documento raíz hacia el elemento objetivo.

b) **Fase de Burbujeo:** El evento se propaga desde el elemento objetivo de vuelta hacia el documento raíz.

c) Prevención de la Propagación

```
<div id="padre" style="padding: 50px; background-color: lightcoral;">
  Padre
  <div id="hijo" style="padding: 20px; background-color: lightgreen;">
    Hijo
  </div>
</div>
<script>
  document.getElementById('padre').addEventListener('click', () => {
    alert('Padre clickeado');
  });

  document.getElementById('hijo').addEventListener('click', (event) => {
    alert('Hijo clickeado');
    event.stopPropagation();  // Previene la propagación hacia el padre
  });
</script>
```

9.5. Delegación de Eventos

La delegación de eventos es una técnica que consiste en manejar eventos en un elemento padre en lugar de asignar manejadores de eventos a cada elemento hijo individualmente. Es especialmente útil cuando se tienen muchos elementos similares o dinámicamente generados.

```
<ul id="lista">
  <li>Elemento 1</li>
  <li>Elemento 2</li>
  <li>Elemento 3</li>
</ul>
<script>
  document.getElementById('lista').addEventListener('click', (event) =>
{
    if (event.target.tagName === 'LI') {
      alert(`Elemento clickeado: ${event.target.textContent}`);
    }
  });
</script>
```

9.6. Eventos Personalizados

Puedes crear y despachar tus propios eventos personalizados utilizando el constructor CustomEvent.

```
<button id="miBoton">Despachar Evento</button>
<script>
  const miEvento = new CustomEvent('miEventoPersonalizado', {
    detail: { mensaje: 'Hola, mundo!' }
  });

  document.getElementById('miBoton').addEventListener('click', () => {
    document.dispatchEvent(miEvento);
  });

  document.addEventListener('miEventoPersonalizado', (event) => {
    alert(event.detail.mensaje);
  });
</script>
```

9.7. Recopilando información de eventos

En JavaScript, los eventos son acciones o sucesos que ocurren en el navegador y que el código puede manejar o responder. Por ejemplo, cuando un usuario hace clic en un botón, mueve el mouse, o envía un formulario, estos son eventos que pueden desencadenar funciones específicas en tu código.

¿Cómo obtener información de los eventos en JavaScript?

a) **Escuchar el evento:** Primero necesitas "escuchar" el evento. Esto se hace utilizando un event listener o manejador de eventos (`event handler`). Un event listener es una función que se ejecuta en respuesta a un evento.

b) **Acceder al objeto event**: Cuando se desencadena un evento, el navegador crea un objeto event que contiene información sobre el evento. Puedes acceder a este objeto en la función manejadora del evento.

Supongamos que quieres obtener información sobre un clic en un botón:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de Evento</title>
</head>
<body>
    <button id="myButton">Haz clic aquí</button>
```

```
    <script>
        // Seleccionamos el botón por su id
        var button = document.getElementById('myButton');

        // Agregamos un event listener al botón
        button.addEventListener('click', function(event) {
            // Aquí accedemos al objeto event
            console.log('El botón fue clicado!');
            console.log('Evento:', event);

            // Ejemplos de información que puedes obtener del objeto
event:
            console.log('Tipo de evento:', event.type); // "click"
            console.log('Elemento objetivo:', event.target); // El botón
que fue clicado
            console.log('Coordenadas del clic:', event.clientX,
event.clientY); // Coordenadas del clic en la ventana
        });
    </script>
</body>
</html>
```

Desglose del código:

1. document.getElementById('myButton'): Selecciona el botón con el ID `myButton`.

2. button.addEventListener('click', function(event) { ... }): Añade un *event listener* para el evento `click`. Cada vez que el botón es clicado, se ejecuta la función proporcionada.

3. event: Este es el objeto de evento que se pasa automáticamente a la función manejadora. Contiene mucha información útil sobre el evento.

Algunas propiedades comunes del objeto `event`:

- type: Tipo de evento (por ejemplo, "click", "mouseover", "keydown").
- target: El elemento que desencadenó el evento.
- currentTarget: El elemento al cual el manejador del evento fue adjuntado.
- clientX` y `clientY: Las coordenadas X e Y del cursor del mouse en el momento del evento, relativas a la ventana.
- preventDefault(): Método que detiene la acción predeterminada del evento (por ejemplo, evitar que un formulario se envíe).
- stopPropagation(): Método que detiene la propagación del evento hacia otros elementos.

Ahora vamos a profundizar en cómo obtener y utilizar las coordenadas donde se realiza un clic dentro de la ventana del navegador y dentro de un elemento específico.

Cuando un usuario hace clic en algún lugar de la página, el evento `click` contiene información sobre las coordenadas X e Y de ese clic. Estas coordenadas pueden ser obtenidas de diferentes maneras dependiendo de tu necesidad:

1. Coordenadas relativas a la ventana (`clientX` y `clientY`)**: Indican la posición del clic en relación con la esquina superior izquierda de la ventana del navegador.

2. Coordenadas relativas al documento (`pageX` y `pageY`)**: Indican la posición del clic en relación con la esquina superior izquierda del documento completo. Es útil cuando la página tiene desplazamiento (scroll).

3. Coordenadas relativas al elemento (`offsetX` y `offsetY`)**: Indican la posición del clic en relación con la esquina superior izquierda del elemento en el que se hizo clic.

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de Coordenadas de Clic</title>
</head>
<body>
    <button id="myButton" style="margin-top: 50px; padding: 20px;">Haz
clic aquí</button>

    <script>
        // Seleccionamos el botón por su id
        var button = document.getElementById('myButton');

        // Agregamos un event listener al botón
        button.addEventListener('click', function(event) {
            // Obtenemos diferentes coordenadas del clic
            var clientX = event.clientX;
            var clientY = event.clientY;
            var pageX = event.pageX;
            var pageY = event.pageY;
            var offsetX = event.offsetX;
            var offsetY = event.offsetY;

            // Mostramos las coordenadas en la consola
            console.log('Coordenadas relativas a la ventana (clientX,
clientY):', clientX, clientY);
            console.log('Coordenadas relativas al documento (pageX,
pageY):', pageX, pageY);
            console.log('Coordenadas relativas al elemento (offsetX,
offsetY):', offsetX, offsetY);
```

```
            // Ejemplo de uso: mostrar un mensaje en el botón con las
coordenadas
            button.textContent = 'Clic en: (' + clientX + ', ' + clientY
+ ')';
        });
    </script>
</body>
</html>
```

Veamos qué hace este código:

1. Coordenadas relativas a la ventana (clientX, clientY): Proporcionan la posición del clic en la ventana del navegador. Es útil si necesitas saber dónde dentro de la ventana ocurrió el clic.

2. Coordenadas relativas al documento (pageX, pageY): Indican la posición del clic en relación con todo el documento. Esto tiene en cuenta cualquier desplazamiento de la página. Es útil para saber la posición real en la página completa, no solo en lo que se ve en la ventana.

3. Coordenadas relativas al elemento (`offsetX`, `offsetY`): Dan la posición del clic en relación con el elemento específico en el que se hizo clic. Esto es útil cuando deseas saber dónde dentro de un botón, imagen, o cualquier otro elemento ocurrió el clic.

Posibles usos de estas coordenadas:

- Crear efectos visuales: Puedes usar estas coordenadas para generar animaciones que se inicien en el punto exacto donde se hizo clic.

- Mostrar información contextual: Como un menú emergente que aparezca exactamente donde el usuario hizo clic.

- Grabar interacciones del usuario: Para analizar cómo interactúan con diferentes partes de tu página.

Las coordenadas del clic en un evento click pueden proporcionar una gran cantidad de información útil que puedes utilizar para mejorar la interacción del usuario con tu sitio o aplicación web. Dependiendo del tipo de interacción que estés buscando implementar, puedes optar por las coordenadas relativas a la ventana, al documento o al elemento específico.

9.8. Ejercicios

Ejercicio 1: Cambio de Color al Hacer Clic

Crea una página con un botón que cambie de color cada vez que se haga clic en él. Los colores deben alternarse entre rojo, azul y verde en ese orden.

Ejercicio 2. Mostrar/Ocultar Texto

Crea una página con un párrafo de texto y un botón que, al hacer clic, oculte el texto si está visible, o lo muestre si está oculto.

Ejercicio 3: Contador de Teclas Presionadas

Crea una página que cuente el número de veces que se ha presionado una tecla y muestre el conteo en pantalla.

Ejercicio 4: Arrastrar y Soltar (Drag and Drop)

Crea una página con dos áreas. La primera área debe contener varios elementos que puedan ser arrastrados y la segunda área debe ser un contenedor donde los elementos pueden soltarse. Al soltar un elemento, este debe desaparecer de la primera área y aparecer en la segunda.

10. El árbol DOM

10.1. Introducción

DOM (Document Object Model) es una interfaz de programación para documentos HTML y XML. Representa la estructura del documento como un árbol de nodos, donde cada nodo es un objeto que representa una parte del documento. Manipular el DOM te permite modificar la estructura, el estilo y el contenido de una página web en tiempo real.

Cuando el navegador carga una página web, crea un árbol DOM que refleja la estructura de la página. Cada nodo del árbol representa un elemento HTML, un atributo o un texto dentro del documento.

Ejemplo de HTML y Árbol DOM

Considera el siguiente fragmento de código HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi Página</title>
</head>
<body>
    <header>
        <h1>Bienvenido a Mi Página</h1>
    </header>
    <main>
        <section>
            <h2>Sección 1</h2>
            <p>Este es el primer párrafo en la sección 1.</p>
        </section>
        <section>
            <h2>Sección 2</h2>
            <p>Este es el primer párrafo en la sección 2.</p>
        </section>
    </main>
    <footer>
        <p>Derechos reservados 2024</p>
    </footer>
</body>
</html>
```

Este HTML se traduce en un árbol DOM en memoria que se ve algo así:
- Document

- html
  - head
    - meta (charset="UTF-8")
    - title (Mi Página)
  - body
    - header
      - h1 (Bienvenido a Mi Página)
    - main
      - section
        - h2 (Sección 1)
        - p (Este es el primer párrafo en la sección 1.)
      - section
        - h2 (Sección 2)
        - p (Este es el primer párrafo en la sección 2.)
    - footer
      - p (Derechos reservados 2024)

Donde cada parte tiene un significado concreto:

a) **Document:** Es el nodo raíz del árbol DOM, que representa el documento completo.

b) **html:** Es el primer nodo hijo del Document, que representa el elemento <html>.

c) **head:** Nodo hijo de <html>, que contiene meta-información y otros elementos relacionados con la cabecera del documento.

- **meta:** Nodo hijo de <head>, que define la codificación de caracteres.
- **title:** Nodo hijo de <head>, que define el título del documento.

d) **body:** Nodo hermano de <head>, que contiene el contenido visible de la página.

- header: Nodo hijo de <body>, que contiene el encabezado de la página.
  - h1: Nodo hijo de <header>, que contiene el título principal de la página.

- main: Nodo hermano de <header>, que contiene el contenido principal.
  - section: Primer nodo hijo de <main>, que representa una sección.
    - h2: Nodo hijo de <section>, que representa un subtítulo dentro de la primera sección.
    - p: Nodo hijo de <section>, que representa un párrafo en la primera sección.

  - section: Segundo nodo hijo de <main>, que representa otra sección.
    - h2: Nodo hijo de <section>, que representa un subtítulo dentro de la segunda sección.
    - p: Nodo hijo de <section>, que representa un párrafo en la segunda sección.

- footer: Nodo hermano de <main>, que contiene el pie de página.
  - p: Nodo hijo de <footer>, que contiene información de derechos reservados.

En las herramientas de desarrollo de tu navegador (como las DevTools en Chrome o Firefox), puedes ver este árbol DOM en el panel de "Elementos". Aquí, cada elemento HTML se muestra como un nodo en un árbol jerárquico que refleja su estructura en el código fuente.

10.2. Selección de elementos del Árbol DOM

Una vez que el árbol DOM está construido, puedes manipularlo con JavaScript para cambiar el contenido, los atributos o el estilo de los elementos en la página. Por ejemplo:

Para manipular el DOM, primero necesitas acceder a los elementos HTML de la página. Esto se puede hacer utilizando varios métodos proporcionados por el objeto document.

a) **getElementById():** Devuelve el elemento que tiene el atributo `id` especificado. Cada id HTML debe ser único y de esta forma encontraremos elementos individuales.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <h1 id="titulo">Hola Mundo</h1>
    <script>
        const titulo = document.getElementById("titulo");
        console.log(titulo.textContent); // Salida: Hola Mundo
    </script>
</body>
</html>
```

b) **getElementsByClassName():** Devuelve una colección de elementos con la clase especificada. Esta colección estará formada por todos los nodos que tengan la clase especificada y podrá recorrerse como si fuera, por ejemplo, un array.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <p class="parrafo">Primer párrafo</p>
```

```
    <p class="parrafo">Segundo párrafo</p>
    <script>
        const parrafos = document.getElementsByClassName("parrafo");
        console.log(parrafos[0].textContent); // Salida: Primer párrafo
    </script>
</body>
</html>
```

c) **getElementsByTagName():** Devuelve una **colección de elementos** con el nombre de etiqueta especificado.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo getElementsByTagName</title>
</head>
<body>
  <h1>Título principal</h1>
  <p>Este es un párrafo 1.</p>
  <p>Este es un párrafo 2.</p>
  <p>Este es un párrafo 3.</p>

  <script>
    // Obtener todos los elementos <p> de la página
    let parrafos = document.getElementsByTagName("p");

    // Iterar sobre los elementos y mostrar su contenido
    for(let i = 0; i < parrafos.length; i++) {
      console.log(parrafos[i].textContent);
    }
  </script>
</body>
</html>
```

d) **getElementsByName():** se utiliza para seleccionar y obtener todos los elementos en un documento HTML que tienen un atributo `name` específico. Este método devuelve una colección de todos los elementos que coinciden con el valor del atributo name dado.

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de getElementsByName</title>
```

```
</head>
<body>
    <form>
        <input type="text" name="usuario" value="Usuario1">
        <input type="text" name="usuario" value="Usuario2">
        <input type="password" name="password" value="123456">
        <input type="submit" value="Enviar">
    </form>

    <script>
        // Obtener todos los elementos con el atributo name="usuario"
        var usuarios = document.getElementsByName("usuario");

        // Recorrer y mostrar el valor de cada elemento con
name="usuario"
        for (var i = 0; i < usuarios.length; i++) {
            console.log(usuarios[i].value);
        }
    </script>
</body>
</html>
```

Normalmente se usa para seleccionar campos de un formulario que comparten el mismo nombre, lo cual es útil cuando se manejan elementos como grupos de radios o checkboxes, donde varios elementos pueden tener el mismo name.

e) **querySelector():** Devuelve el **primer elemento** que coincide con el selector CSS especificado.Para especificar el elemento nos valdremos de la sintaxis de CSS.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <p class="parrafo">Texto en párrafo</p>
    <script>
        const parrafo = document.querySelector(".parrafo");
        console.log(parrafo.textContent); // Salida: Texto en párrafo
    </script>
</body>
</html>
```

El método querySelector en JavaScript permite seleccionar elementos del DOM utilizando una variedad de selectores. A continuación se muestra una lista de los tipos de selectores que puedes utilizar.

Selectores de tipo: document.querySelector('div'); // Selecciona el primer

Selectores de clase: document.querySelector('.mi-clase'); // Selecciona el primer elemento con la clase "mi-clase"

Selectores de ID: document.querySelector('#mi-id'); // Selecciona el elemento con el ID "mi-id"

Selectores de atributos: document.querySelector('[type="text"]'); // Selecciona el primer elemento con type="text"

Selectores de descendientes: document.querySelector('div p'); // Selecciona el primer <p> que es descendiente de un

Selectores de hijo directo: document.querySelector('div > p'); // Selecciona el primer <p> que es hijo directo de un

Selectores de hermano adyacente: document.querySelector('h1 + p'); // Selecciona el primer <p> que es hermano adyacente de un <h1>

Selectores de hermano general: document.querySelector('h1 ~ p'); // Selecciona todos los <p> que son hermanos de un <h1>

Selectores combinados: document.querySelector('div.mi-clase#mi-id'); // Selecciona el primer que tiene la clase "mi-clase" y el ID "mi-id"

Pseudo-clases: document.querySelector('a:hover'); // Selecciona el primer <a> que está siendo "hovered"

Pseudo-elementos: document.querySelector('p::first-line'); // Selecciona la primera línea del primer <p>

Estos son algunos ejemplos de selectores que puedes usar con `querySelector`. La clave es que `querySelector` acepta cualquier selector CSS válido.

f) **querySelectorAll():** Devuelve una **colección de todos los elementos** que coinciden con el selector CSS especificado.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
```

```
</head>
<body>
    <p class="parrafo">Primer párrafo</p>
    <p class="parrafo">Segundo párrafo</p>
    <script>
        const parrafos = document.querySelectorAll(".parrafo");
        parrafos.forEach(parrafo => {
            console.log(parrafo.textContent);
        });
        // Salida:
        // Primer párrafo
        // Segundo párrafo
    </script>
</body>
</html>
```

10.3. Manipulación de Elementos

Una vez que tienes acceso a un elemento, puedes modificar su contenido, atributos y estilos.

a) **Modificar el Contenido:** Puedes cambiar el contenido de un elemento utilizando textContent o innerHTML. Estas propiedades también sirven para visualizar el contenido de un determinado nodo. La diferencia entre ambos es que con textContent modificamos solo el texto del elemento mientras que con innerHTML modificamos su contenido HTML.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <div id="miDiv">Contenido original</div>
    <script>
        const miDiv = document.getElementById("miDiv");
        miDiv.textContent = "Contenido modificado"; // Cambia solo el
texto
        // miDiv.innerHTML = "<strong>Contenido modificado</strong>"; //
Cambia el contenido HTML
    </script>
</body>
</html>
```

b) **Modificar Atributos:** Puedes modificar los atributos de los elementos con setAttribute() y getAttribute().

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <img id="miImagen" src="imagen.jpg" alt="Imagen original">
    <script>
        const imagen = document.getElementById("miImagen");
        imagen.setAttribute("src", "nueva-imagen.jpg");
        imagen.setAttribute("alt", "Imagen modificada");
    </script>
</body>
</html>
```

En el siguiente ejemplo vemos código en JavaScript que crea dos botones, uno para aumentar el tamaño de una imagen en 5 píxeles y otro para disminuirlo en la misma proporción:

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Aumentar y Disminuir Tamaño de Imagen</title>
    <style>
        #miImagen {
            width: 100px; /* Tamaño inicial de la imagen */
        }
    </style>
    <script src=”utiles.js”>
</head>
<body>

    <img id="miImagen" src="https://via.placeholder.com/150" alt="Imagen
de ejemplo">
    <br>
    <button onclick="aumentar()">Aumentar tamaño</button>
```

```
    <button onclick="disminuir()">Disminuir tamaño</button>
</body>
</html>
```

utiles.js

```
        function aumentar() {
            const img = document.getElementById("miImagen");
            const width = img.clientWidth;
            img.style.width = (width + 5) + "px";
        }

        function disminuir() {
            const img = document.getElementById("miImagen");
            const width = img.clientWidth;
            img.style.width = (width - 5) + "px";
        }
```

c) **Modificar Estilos:** Puedes cambiar el estilo de un elemento a través de la propiedad style. Los elementos de esta propiedad se escribirán en formato camel case.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <div id="miCaja" style="width: 100px; height: 100px;
background-color: red;"></div>
    <script>
        const miCaja = document.getElementById("miCaja");
        miCaja.style.backgroundColor = "blue";
        miCaja.style.width = "200px";
    </script>
</body>
</html>
```

También podemos modificar los estilos mediante la propiedad className. Esta es una manera efectiva de aplicar múltiples estilos predefinidos a un elemento HTML. Se vuelve especialmente útil cuando quieres aplicar un conjunto completo de estilos (especificados en una clase CSS) en lugar de modificar estilos individuales uno por uno.

A continuación, veremos un ejemplo sencillo que cambia un estilo inicial por otro al desencadenarse un evento:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Ejemplo de Cambio de Estilo con className</title>
    <style>
        /* Definimos algunas clases CSS */
        .estiloOriginal {
            color: black;
            font-size: 16px;
            background-color: lightgray;
            padding: 10px;
            border: 2px solid black;
        }

        .nuevoEstilo {
            color: white;
            font-size: 20px;
            background-color: blue;
            padding: 20px;
            border-radius: 10px;
            border: none;
        }
    </style>
</head>
<body>
    <!-- Elemento al que se le aplicarán los estilos -->
    <div id="miDiv" class="estiloOriginal">Este es un texto con
estilo.</div>
    <!-- Botón que cambiará el estilo del elemento -->
    <button onclick="cambiarEstilo()">Cambiar Estilo</button>

    <script>
        function cambiarEstilo() {
            // Selecciona el elemento por su ID
            var elemento = document.getElementById("miDiv");

            // Cambia la clase del elemento
            elemento.className = "nuevoEstilo";
        }
    </script>
</body>
</html>
```

10.4. Añadir y Eliminar Elementos

Puedes crear nuevos elementos usando document.createElement() y luego añadirlos al DOM con **appendChild()** o **insertBefore()**.

Normalmente para crear un elemento se siguen los siguientes cuatro pasos, aunque solo dos de ellos son obligatorios:

a) **Crear el elemento:** Utiliza el método document.createElement para crear un nuevo elemento.

```
let nuevoElemento = document.createElement('div');
```

b) **Configurar el elemento (opcional):** Establece atributos, clases, IDs, contenido de texto, y otros aspectos del nuevo elemento.

```
nuevoElemento.id = 'mi-id';
nuevoElemento.className = 'mi-clase';
nuevoElemento.textContent = 'Hola, mundo!';
// También puedes agregar atributos personalizados
nuevoElemento.setAttribute('data-personalizado', 'valor');
```

c) **Añadir el elemento al DOM:** Utiliza un método de manipulación del DOM para insertar el nuevo elemento en el documento. Comúnmente se usan métodos como **appendChild, insertBefore, append,** entre otros.

```
let elementoPadre = document.getElementById('elemento-padre');
elementoPadre.appendChild(nuevoElemento);
```

d) **Opcionalmente, añadir manejadores de eventos:** Añade cualquier manejador de eventos que necesite el nuevo elemento para que pueda responder a interacciones del usuario.

```
nuevoElemento.addEventListener('click', function() {
    alert('¡Elemento clickeado!');
});
```

Siguiendo estos pasos, puedes crear y manipular elementos en el DOM de manera efectiva y dinámica.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
```

```
    <title>Ejemplo DOM</title>
</head>
<body>
    <ul id="miLista">
        <li>Elemento 1</li>
    </ul>
    <script>
        const lista = document.getElementById("miLista");
        const nuevoElemento = document.createElement("li");
        nuevoElemento.textContent = "Elemento 2";
        lista.appendChild(nuevoElemento);
    </script>
</body>
</html>
```

Puedes eliminar elementos del DOM con removeChild() o remove().

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo DOM</title>
</head>
<body>
    <ul id="miLista">
        <li>Elemento 1</li>
        <li id="eliminarEste">Elemento 2</li>
    </ul>
    <script>
        const elementoAEliminar =
document.getElementById("eliminarEste");
        elementoAEliminar.remove(); // O usa
lista.removeChild(elementoAEliminar);
    </script>
</body>
</html>
```

En el siguiente ejemplo veremos un script que añade y elimina elementos de una lista de frutas.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
```

```
initial-scale=1.0">
    <title>Lista de Frutas</title>
</head>
<body>
    <h1>Agregar Frutas a la Lista</h1>
    <input type="text" id="fruitInput" placeholder="Escribe el nombre de
una fruta">
    <button id="addButton">Agregar Fruta</button>
    <ul id="fruitList"></ul>

    <script>
        // Seleccionamos el input, el botón y la lista
        const fruitInput = document.getElementById('fruitInput');
        const addButton = document.getElementById('addButton');
        const fruitList = document.getElementById('fruitList');

        // Función para agregar la fruta a la lista
        function addFruit() {
            const fruitName = fruitInput.value.trim();

            // Solo agregamos si el input no está vacío
            if (fruitName !== "") {
                const listItem = document.createElement('li');
                listItem.textContent = fruitName;

                // Creamos el botón de eliminar
                const deleteButton = document.createElement('button');
                deleteButton.textContent = 'Eliminar';
                deleteButton.className = 'deleteButton';

                // Añadimos el botón de eliminar al elemento de lista
                listItem.appendChild(deleteButton);

                // Añadimos el elemento de lista a la lista desordenada
                fruitList.appendChild(listItem);

                // Limpiamos el input después de agregar la fruta
                fruitInput.value = "";

                // Añadimos el evento para eliminar el elemento
                deleteButton.addEventListener('click', function() {
                    fruitList.removeChild(listItem);
                });
            }
        }
```

```
    // Escuchamos el evento keyup para capturar cuando se presiona Enter
        fruitInput.addEventListener('keyup', function(event) {
            if (event.key === 'Enter') {
                addFruit();
            }
        });

        // Escuchamos el evento click del botón para agregar la fruta
        addButton.addEventListener('click', addFruit);
    </script>
</body>
</html>
```

1. Botón de eliminar:
   - const deleteButton = document.createElement('button'): Creamos un botón que servirá para eliminar el elemento de la lista.
   - deleteButton.textContent = 'Eliminar': Establecemos el texto del botón como "Eliminar".
   - deleteButton.className = 'deleteButton': Asignamos una clase al botón para poder darle estilo.

2. Añadir el botón de eliminar al elemento de la lista:
   - listItem.appendChild(deleteButton): Agregamos el botón de eliminar al `<li>` que representa la fruta en la lista.

3. Función de eliminar:
   - deleteButton.addEventListener('click', function() { ... }): Escuchamos el evento `click` en el botón de eliminar.
   - fruitList.removeChild(listItem): Cuando se hace clic en el botón de eliminar, se remueve el `<li>` correspondiente de la lista.

Este script ahora permite agregar frutas a la lista y, además, eliminar cualquier fruta que se haya añadido, proporcionando una funcionalidad completa para manejar la lista de frutas.

Finalmente veamos un ejemplo de cómo **seleccionar y modificar** el contenido de los elementos de una **tabla**:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Ejemplo de Selección en Tabla</title>
    <style>
        table, th, td {
            border: 1px solid black;
```

```
            border-collapse: collapse;
            padding: 10px;
        }
        th {
            background-color: lightgray;
        }
    </style>
</head>
<body>
    <!-- Tabla de ejemplo -->
    <table id="miTabla">
        <thead>
            <tr>
                <th>Encabezado 1</th>
                <th>Encabezado 2</th>
                <th>Encabezado 3</th>
                <th>Encabezado 4</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Fila 1, Col 1</td>
                <td>Fila 1, Col 2</td>
                <td>Fila 1, Col 3</td>
                <td>Fila 1, Col 4</td>
            </tr>
            <tr>
                <td>Fila 2, Col 1</td>
                <td>Fila 2, Col 2</td>
                <td>Fila 2, Col 3</td>
                <td>Fila 2, Col 4</td>
            </tr>
            <tr>
                <td>Fila 3, Col 1</td>
                <td>Fila 3, Col 2</td>
                <td>Fila 3, Col 3</td>
                <td>Fila 3, Col 4</td>
            </tr>
        </tbody>
    </table>

    <!-- Botones para interactuar con la tabla -->
    <button onclick="seleccionarFila()">Seleccionar Fila 2</button>
    <button onclick="seleccionarCelda()">Seleccionar Celda
[2,3]</button>
    <button onclick="seleccionarContenido()">Mostrar Contenido de
```

```
[3,4]</button>
    <button onclick="seleccionarHijosFila()">Seleccionar Hijos de Fila
1</button>
    <button onclick="seleccionarAbueloContenido()">Seleccionar Abuelo
del Contenido en Celda [2,3]</button>

    <script>
        // Seleccionar y resaltar la segunda fila
        function seleccionarFila() {
            var fila = document.querySelector("#miTabla tbody").rows[1];
            fila.style.backgroundColor = "yellow";
        }

        // Seleccionar y resaltar la celda en la segunda fila y tercera
columna
        function seleccionarCelda() {
            var celda = document.querySelector("#miTabla
tbody").rows[1].cells[2];
            celda.style.backgroundColor = "lightblue";
        }

        // Mostrar el contenido de la celda en la tercera fila y cuarta
columna
        function seleccionarContenido() {
            var contenido = document.querySelector("#miTabla
tbody").rows[2].cells[3].textContent;
            alert("El contenido de la celda [3, 4] es: " + contenido);
        }

        // Seleccionar todos los elementos hijos (celdas) de la primera
fila
        function seleccionarHijosFila() {
            var fila = document.querySelector("#miTabla tbody").rows[0];
            var celdas = fila.children; // Obtener todos los elementos
hijos (celdas)
            for (var i = 0; i < celdas.length; i++) {
                celdas[i].style.backgroundColor = "lightgreen"; //
Cambiar el fondo de cada celda
            }
        }

        // Seleccionar el abuelo del contenido de una celda específica
(fila <tr>)
        function seleccionarAbueloContenido() {
            var celda = document.querySelector("#miTabla
tbody").rows[1].cells[2]; // Celda [2,3]
```

```
            var contenido = celda.firstChild; // Contenido de la celda
            var abuelo = contenido.parentElement.parentElement; //
Abuelo del contenido es la fila <tr>
            abuelo.style.border = "3px solid red"; // Cambiar el borde
de la fila
        }
    </script>
</body>
</html>
```

Veamos ahora un ejemplo de como crear una tabla de 3x3 de forma dinámica mediante los métodos anteriormente citados.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Tabla Dinámica 3x3 con Métodos DOM</title>
    </head>
<body>

    <table id="tabla"></table>

    <script>
        // Obtener el elemento de la tabla por su ID
        const tabla = document.getElementById('tabla');

        // Crear la tabla dinámica 3x3 usando métodos DOM
        for (let i = 0; i < 3; i++) {
            // Crear un elemento de fila (tr)
            const fila = document.createElement('tr');

            for (let j = 0; j < 3; j++) {
                // Crear un elemento de celda (td)
                const celda = document.createElement('td');

                // Asignar el contenido de la celda
                celda.textContent = `Fila ${i + 1}, Columna ${j + 1}`;

                // Añadir la celda a la fila
                fila.appendChild(celda);
            }
```

```
            // Añadir la fila a la tabla
            tabla.appendChild(fila);
        }
    </script>

</body>
</html>
```

Otra alternativa sería crear los elementos dinámicamente con los métodos que ofrece el objeto table del siguiente modo:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Tabla Dinámica 3x3</title>
   </head>
<body>

    <table id="tabla"></table>

    <script>
        // Obtener el elemento de la tabla por su ID
        const tabla = document.getElementById('tabla');

        // Crear la tabla dinámica 3x3
        for (let i = 0; i < 3; i++) {
            // Crear una nueva fila
            const fila = tabla.insertRow();

            for (let j = 0; j < 3; j++) {
                // Crear una nueva celda en la fila actual
                const celda = fila.insertCell();

                // Asignar el contenido de la celda
                celda.textContent = `Fila ${i + 1}, Columna ${j + 1}`;
            }
        }
    </script>

</body>
</html>
```

10.5. Ejercicios

Ejercicio 1: Crear y añadir un elemento

Escribe una función llamada `añadirParrafo` que tome un texto como argumento y añada un nuevo párrafo con ese texto al final del cuerpo del documento (`<body>`).

Ejercicio 2: Cambiar el estilo de un elemento

Escribe una función llamada `cambiarColorFondo` que tome un id de un elemento y un color como argumentos, y cambie el color de fondo del elemento con ese id al color especificado.

Ejercicio 3: Mostrar/Ocultar un elemento
Escribe una función llamada mostrarOcultar que tome un id de un elemento y, si el elemento está visible, lo oculte, y si está oculto, lo muestre.

Ejercicio 4: Añadir una lista de elementos
Escribe una función llamada añadirLista que tome un array de textos como argumento y añada una lista desordenada (<ul>) con esos textos como elementos de lista (`<li>`) al final del cuerpo del documento (<body>).

Ejercicio 5: Actualizar el contenido de un elemento

Escribe una función llamada actualizarContenido que tome un id de un elemento y un nuevo contenido como argumentos, y actualice el contenido HTML del elemento con ese id al nuevo contenido.

Ejercicio 6: Disponemos de una tabla de 3x3 con sus celdas puestas a cero tal y como se ilustra a continuación:

0	0	0
0	0	0
0	0	0

Se pide:

a) Al hacer click en una celda ésta debe incrementar su valor en una unidad. Si llega al valor 10 el fondo de la celda debe ponerse de color amarillo.

b) Añadir un <select> con tres opciones (Subir, Bajar y Resetear). Si la opción **Subir** está seleccionada debe hacer lo que pone el apartado a. Si está seleccionada **Bajar** debe de restar una unidad y si lo seleccionado es **Resetear** entonces debe poner todos los valores a 0.

11. Formularios.

Los formularios en HTML son estructuras de código utilizadas para recopilar información de los usuarios. Permiten a los usuarios introducir datos que luego pueden ser enviados a un servidor para su procesamiento. Los formularios son esenciales para muchas aplicaciones web, como el registro de usuarios, el inicio de sesión, la búsqueda en una base de datos, la recopilación de comentarios, entre otros.

Es importante reseñar que los campos de un formulario tienen un atributo **value** que nos servirá para almacenar o para leer el valor que contienen.

11.1. Etiqueta <form>

El contenedor principal de los elementos de formulario. Atributos importantes:

- **action:** URL a la que se enviarán los datos del formulario. En el caso del cliente es posible que los datos sean tratados en el mismo equipo por lo que posteriormente, una vez manipulados, se enviarán al servidor.

- **method:** Método HTTP utilizado para enviar los datos (`GET` o `POST`).

```
<form id="myForm" action="/submit" method="post">
        <!-- Elementos del formulario -->
    </form>
```

```
document.getElementById('myForm').addEventListener('submit',
function(event) {
        event.preventDefault(); // Evita el envío del formulario
        alert('Formulario enviado!');
    });
```

11.2. Campo de texto (<input type="text">):

Los campos de texto en un formulario HTML sirven para permitir a los usuarios introducir datos que serán enviados al servidor para su procesamiento o bien serán validados en el cliente. Estos campos son esenciales para la interacción y recopilación de información en aplicaciones web.

```
 <label for="username">Nombre de usuario:</label>
    <input type="text" id="username" name="username">
```

```
document.getElementById('username').addEventListener('input', function()
{
        console.log('Nombre de usuario:', this.value);
```

```
    });
```

11.3. Contraseña (<input type="password">):

Es un campo de tipo text con los típicos asteriscos para que no se vea la contraseña.

```
<label for="password">Contraseña:</label>
    <input type="password" id="password" name="password">
```

```
document.getElementById('password').addEventListener('input', function()
{
    console.log('Contraseña:', this.value);
});
```

11.4. Área de texto (<textarea>)

Para entrada de texto de múltiples líneas. Es una alternativa al cuadro de texto cuando queremos explayarnos en la información a introducir.

```
<label for="bio">Biografía:</label>
    <textarea id="bio" name="bio"></textarea>
```

```
 document.getElementById('bio').addEventListener('input', function() {
        console.log('Biografía:', this.value);
    });
```

11.5. Correo electrónico (`<input type="email">`)

Para entrada de direcciones de correo electrónico. Es como un campo de texto pero aporta validación para direcciones de correo.

```
 <label for="email">Correo electrónico:</label>
    <input type="email" id="email" name="email">
```

```
document.getElementById('email').addEventListener('input', function() {
        console.log('Correo electrónico:', this.value);
    });
```

11.6. Campos numéricos (<input type=”number”>)

Se utiliza para permitir que los usuarios ingresen un valor numérico en un formulario. Este tipo de campo es especialmente útil cuando se espera que los usuarios proporcionen

números, como una edad, cantidad, precio, etc. Además, este campo puede tener atributos adicionales para controlar el rango de valores permitidos, los pasos de incremento, y más.

```
<form>
  <label for="age">Edad:</label>
  <input type="number" id="age" name="age" min="1" max="120" step="1"
required>
  <input type="submit" value="Enviar">
</form>
```

11.7. Botones de radio (<input type="radio">)

Para seleccionar una opción de un conjunto. Normalmente se utilizan para pedir datos al usuario de forma exscluyente como podría ser una encuesta de una sola opción. La forma de hacer que sean excluyentes es asignarle el mismo **name** a cada elemento.

```
 <p>Género:</p>
    <input type="radio" id="male" name="gender" value="male">
    <label for="male">Masculino</label>
    <input type="radio" id="female" name="gender" value="female">
    <label for="female">Femenino</label>
```

```
document.getElementsByName('gender').forEach(radio => {
        radio.addEventListener('change', function() {
            console.log('Género seleccionado:', this.value);
        });
    });
```

Para seguir entendiendo este componente veamos el código que implementa una encuesta con tres opciones (Sí, No, NS/NC) y actualiza los votos, el total de votos y los porcentajes cada vez que el usuario vote.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Encuesta</title>
</head>
<body>

    <div class="container">
        <h2>¿Estás de acuerdo?</h2>
```

```
        <button onclick="votar('si')">Sí</button>
        <button onclick="votar('no')">No</button>
        <button onclick="votar('nsnc')">NS/NC</button>

        <div class="result">
            <p>Votos Sí: <span id="votosSi">0</span></p>
            <p>Votos No: <span id="votosNo">0</span></p>
            <p>Votos NS/NC: <span id="votosNsnc">0</span></p>
            <p>Total de votos: <span id="totalVotos">0</span></p>
            <p>Porcentaje Sí: <span id="porcentajeSi"
class="percentage">0%</span></p>
            <p>Porcentaje No: <span id="porcentajeNo"
class="percentage">0%</span></p>
            <p>Porcentaje NS/NC: <span id="porcentajeNsnc"
class="percentage">0%</span></p>
        </div>
    </div>

    <script>
        // Variables para contar los votos
        let votosSi = 0;
        let votosNo = 0;
        let votosNsnc = 0;

        // Función para actualizar los resultados
        function actualizarResultados() {
            const totalVotos = votosSi + votosNo + votosNsnc;

            // Mostrar el número de votos
            document.getElementById('votosSi').innerText = votosSi;
            document.getElementById('votosNo').innerText = votosNo;
            document.getElementById('votosNsnc').innerText = votosNsnc;
            document.getElementById('totalVotos').innerText =
totalVotos;

            // Calcular y mostrar los porcentajes
          const porcentajeSi = totalVotos > 0 ? ((votosSi / totalVotos)
* 100).toFixed(2) : 0;
          const porcentajeNo = totalVotos > 0 ? ((votosNo / totalVotos)
* 100).toFixed(2) : 0;
          const porcentajeNsnc = totalVotos > 0 ? ((votosNsnc /
totalVotos) * 100).toFixed(2) : 0;

            document.getElementById('porcentajeSi').innerText =
porcentajeSi + '%';
```

```
            document.getElementById('porcentajeNo').innerText =
porcentajeNo + '%';
            document.getElementById('porcentajeNsnc').innerText =
porcentajeNsnc + '%';
        }

        // Función para votar
        function votar(opcion) {
            if (opcion === 'si') {
                votosSi++;
            } else if (opcion === 'no') {
                votosNo++;
            } else if (opcion === 'nsnc') {
                votosNsnc++;
            }

            actualizarResultados();
        }
    </script>

</body>
</html>
```

En este ejemplo hay tres botones que representan las opciones de la encuesta: "Sí", "No", y "NS/NC". Cada vez que se hace clic en uno de los botones, se incrementa el contador correspondiente y se actualizan los totales y porcentajes.

La función actualizarResultados() se encarga de actualizar el DOM con los nuevos valores y calcular los porcentajes. Si no se han emitido votos, los porcentajes se muestran como `0%`.

Propuesta de mejora: Podríamos poner al lado de cada voto una barra que nos representará el porcentaje. Para ello podríamos colocar una imagen de 1 px de anchura y en cada voto modificamos el atributo width para hacer más ancha la imagen. También podemos utilizar componentes para ello más avanzados.

11.8. Casillas de verificación (<input type="checkbox">):

Para seleccionar múltiples opciones y que éstas no sean excluyentes..

```
<p>Aficiones:</p>
    <input type="checkbox" id="hobby1" name="hobbies" value="reading">
    <label for="hobby1">Lectura</label>
    <input type="checkbox" id="hobby2" name="hobbies" value="sports">
    <label for="hobby2">Deportes</label>
```

```
   document.querySelectorAll('input[name="hobbies"]').forEach(checkbox
=> {
        checkbox.addEventListener('change', function() {
            console.log('Afición seleccionada:', this.value, 'Estado:',
this.checked);
        });
    });
```

11.9. Lista desplegable (<select>):

Para seleccionar una opción de una lista desplegable. Es sin duda uno de los elementos más utilizados cuando se quiere dar a elegir entre una serie de valores tan grande que difícilmente cabría en la página, por lo que se pueden mostrar en una lista donde seleccionaremos un elemento. Un ejemplo sería la lista de las provincias españolas ó una lista donde el usuario eligiese su nacionalidad.

```
<label for="country">País:</label>
    <select id="country" name="country">
        <option value="us">Estados Unidos</option>
        <option value="es">España</option>
        <option value="mx">México</option>
    </select>
```

```
document.getElementById('country').addEventListener('change', function()
{
        console.log('País seleccionado:', this.value);
    });
```

Podemos añadir elementos al Select modificando su lista de opciones.

```
<select id="miSelect">
  <option value="1">Opción 1</option>
</select>

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

  // Crear nueva opción
  const nuevaOpcion = new Option("Opción 2", "2");

  // Insertarla al final
  select.options.add(nuevaOpcion);
</script>
```

También podemos insertar en una posición concreta:

```
<select id="miSelect">
  <option value="1">Opción 1</option>
  <option value="3">Opción 3</option>
</select>

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

  // Insertar en la posición 1
  select.options[1] = new Option("Opción 2", "2");
</script>
```

Veamos el clásico ejemplo de selects dependientes, es decir, el típico ejemplo de seleccionar en un primer select una provincia y que cargue en el segundo solo los pueblos de esa provincia. En este ejemplo lo haremos con un primer select que tiene una lista de categorías y de forma dinámica cargaremos el segundo con la lista de subcategorías asociada.

```
<div>
  <label for="categoria">Categoría:</label>
  <select id="categoria" onchange="cargarSubcategorias()">
    <option value="">Selecciona una categoría</option>
    <option value="tecnologia">Tecnología</option>
    <option value="hogar">Hogar</option>
    <option value="ropa">Ropa</option>
  </select>
</div>

<div>
  <label for="subcategoria">Subcategoría:</label>
  <select id="subcategoria">
    <option value="">Selecciona una subcategoría</option>
  </select>
</div>
```

```
// Datos de ejemplo: categorías y sus subcategorías
const subcategorias = {
  tecnologia: ["Smartphones", "Computadoras", "Accesorios"],
  hogar: ["Muebles", "Electrodomésticos", "Decoración"],
  ropa: ["Camisas", "Pantalones", "Zapatos"]
};

// Función que se ejecuta cuando se selecciona una categoría
```

```
function cargarSubcategorias() {
  const categoria = document.getElementById("categoria").value;
  const subcategoriaSelect = document.getElementById("subcategoria");

  // Limpiar las opciones anteriores
  subcategoriaSelect.innerHTML = '<option value="">Selecciona una
subcategoría</option>';

  // Si se seleccionó una categoría válida, cargar las subcategorías
correspondientes
  if (categoria) {
    const subcats = subcategorias[categoria];

    subcats.forEach(subcat => {
      const opcion = document.createElement("option");
      opcion.value = subcat.toLowerCase();
      opcion.textContent = subcat;
      subcategoriaSelect.appendChild(opcion);
    });
  }
}
```

Analicemos en detalle este código:

1. HTML: Se tienen dos <select>, uno para las categorías **(id="categoria")** y otro para las subcategorías **(id="subcategoria")**. El primer <select> tiene un evento **onchange** que llama a la función cargarSubcategorias() cada vez que se selecciona una categoría.

2. JavaScript: Definimos un objeto subcategorias que asocia cada categoría con un array de subcategorías. La función cargarSubcategorias() obtiene la categoría seleccionada y el elemento <select> de subcategorías. Se limpia el `<select>` de subcategorías y si la categoría seleccionada es válida, se cargan las subcategorías correspondientes en el select> de subcategorías.

Cuando el usuario selecciona una categoría en el primer <select>, el segundo <select> se actualiza automáticamente con las subcategorías correspondientes. Si el usuario cambia la categoría seleccionada, las subcategorías se vuelven a cargar según la nueva selección.

Este enfoque es completamente en JavaScript puro y es fácil de adaptar a cualquier estructura de categorías y subcategorías.

11.10. Lista Múltiple

Se trata de un elemento <select> que permite seleccionar múltiples opciones. Por ejemplo, imaginemos una pregunta que te diga cuáles son tus frutas preferidas y te permita elegir más de una. El código para manejar este elemento sería 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>Lista de Selección Múltiple</title>
</head>
<body>
    <form>
        <label for="frutas">Selecciona tus frutas favoritas:</label>
        <select id="frutas" name="frutas" multiple>
            <option value="manzana">Manzana</option>
            <option value="banana">Banana</option>
            <option value="naranja">Naranja</option>
            <option value="uva">Uva</option>
            <option value="mango">Mango</option>
        </select>
        <input type="button" onclick="mostrarSeleccion()">Mostrar
Selección</button>
    </form>

    <p id="resultado"></p>

    <script>
        function mostrarSeleccion() {
            // Obteniendo la referencia al elemento select
            const selectElement = document.getElementById('frutas');

            // Array para almacenar las opciones seleccionadas
            let seleccionadas = [];

            // Iterando sobre las opciones del select
            for (let option of selectElement.options) {
                // Si la opción está seleccionada, se agrega al array
                if (option.selected) {
                    seleccionadas.push(option.value);
                }
            }

            // Mostrando la selección en el párrafo con id="resultado"
            document.getElementById('resultado').textContent =
                seleccionadas.length > 0 ? `Has seleccionado:
${seleccionadas.join(', ')}` : "No has seleccionado nada";
        }
```

```
    </script>
</body>
</html>
```

En este código debemos fijarnos en los siguientes aspectos:
- Se utiliza document.getElementById('frutas') para obtener el elemento <select>.
- Se recorre cada opción del <select> usando un bucle for...of.
- Si una opción está seleccionada (option.selected es true), se agrega a un array llamado seleccionadas.
- Finalmente, se muestra el resultado de las selecciones en un elemento <p>.

Con este código, los usuarios pueden seleccionar varias frutas de la lista, y cuando hagan clic en el botón, se mostrará cuáles han sido seleccionadas.

11.11. Botón de envío (<button type="submit">) y reset (<button type=”reset”>)

Para enviar el formulario utilizaremos el click en un botón de tipo submit..

```
   <button type="submit">Enviar</button>
   <button type="reset">Eliminar datos</button>
```

```
document.querySelector('button[type="submit"]').addEventListener('click'
, function(event) {
        event.preventDefault(); // Evita el envío del formulario
        alert('Botón de envío presionado!');
    });
```

El botón reset en un formulario HTML se utiliza para restablecer todos los campos del formulario a sus valores iniciales. Esto significa que si un usuario ha llenado o modificado algún campo del formulario, al presionar el botón reset, todos los campos volverán a los valores predeterminados que tenían cuando se cargó la página.

11.12. Subida de ficheros (<input type=”file”>)

El campo de tipo `file` en HTML se utiliza para permitir a los usuarios seleccionar archivos desde su dispositivo (como imágenes, documentos, videos, etc.) y subirlos al servidor. Este campo es comúnmente usado en formularios donde se necesita cargar archivos, cómo al subir una foto de perfil, un currículum, o cualquier tipo de archivo adjunto.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Subir Archivo</title>
</head>
<body>
    <h2>Subir Archivo</h2>
    <form id="fileForm">
        <label for="archivo">Selecciona un archivo:</label>
        <input type="file" id="archivo" name="archivo"
onchange="mostrarNombreArchivo()">
        <br><br>
        <p id="nombreArchivo"></p>
        <button type="button" onclick="subirArchivo()">Subir</button>
    </form>

    <script src="app.js"></script>
</body>
</html>
```

```
function mostrarNombreArchivo() {
    const archivoInput = document.getElementById('archivo');
    const nombreArchivo = archivoInput.files[0].name;
    document.getElementById('nombreArchivo').textContent = `Archivo
seleccionado: ${nombreArchivo}`;
}

function subirArchivo() {
    const archivoInput = document.getElementById('archivo');
    const archivo = archivoInput.files[0];

    if (archivo) {
        const formData = new FormData();
        formData.append('archivo', archivo);

        // Aquí podrías enviar el archivo a un servidor utilizando fetch
o XMLHttpRequest
        // Este es un ejemplo de cómo podrías hacerlo con fetch:

        fetch('ruta_del_servidor_para_subir_archivo', {
            method: 'POST',
            body: formData
        })
        .then(response => response.json())
        .then(data => {
```

```
            console.log('Archivo subido exitosamente:', data);
            alert('Archivo subido con éxito.');
        })
        .catch(error => {
            console.error('Error al subir el archivo:', error);
            alert('Hubo un error al subir el archivo.');
        });
    } else {
        alert('Por favor, selecciona un archivo.');
    }
}
```

Los navegadores limitan la capacidad de los scripts para acceder a la información de los archivos por razones de seguridad. Los archivos seleccionados solo pueden ser manipulados por el cliente y enviados al servidor, pero no se puede acceder a la ruta completa del archivo por JavaScript.

Carga múltiple: Puedes permitir la selección de múltiples archivos usando el atributo multiple en el campo `<input type="file">`.

11.13. Campos de fecha y hora

Los campos **date, time, y datetime-local** en HTML se utilizan para capturar fechas, horas y combinaciones de fecha y hora en formularios. Estos tipos de entrada proporcionan una interfaz gráfica amigable (como calendarios y selectores de tiempo) en navegadores que los soportan, facilitando la entrada de datos por parte del usuario.

a) **Campo date:** Permite al usuario seleccionar una fecha específica (día, mes y año).

b) **Campo time:** Permite al usuario seleccionar una hora específica (horas y minutos, y opcionalmente, segundos).
c) **Campo datetime-local:** Combina la funcionalidad de `date` y `time`, permitiendo al usuario seleccionar tanto una fecha como una hora.

Aquí tienes el código HTML con un pequeño script de JavaScript que obtiene los valores de los campos `date`, `time`, y `datetime-local` cuando se envía el formulario, y los muestra en la consola.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Ejemplo de Campos de Fecha y Hora</title>
</head>
```

```
<body>
    <h2>Reservar una Cita</h2>
    <form id="formCita">
        <label for="fecha">Fecha de la cita:</label>
        <input type="date" id="fecha" name="fecha">
        <br><br>

        <label for="hora">Hora de la cita:</label>
        <input type="time" id="hora" name="hora">
        <br><br>

        <label for="fechaHora">Fecha y hora de inicio:</label>
        <input type="datetime-local" id="fechaHora" name="fechaHora">
        <br><br>

        <button type="submit">Reservar</button>
    </form>

    <script>
        // Obtener el formulario por su ID
        const formCita = document.getElementById('formCita');

        // Escuchar el evento 'submit' del formulario
        formCita.addEventListener('submit', function(event) {
            // Evitar el envío del formulario
            event.preventDefault();

            // Obtener los valores de los campos
            const fecha = document.getElementById('fecha').value;
            const hora = document.getElementById('hora').value;
            const fechaHora =
document.getElementById('fechaHora').value;

            // Mostrar los valores en la consola
            console.log('Fecha de la cita:', fecha);
            console.log('Hora de la cita:', hora);
            console.log('Fecha y hora de inicio:', fechaHora);
        });
    </script>
</body>
</html>
```

A medida que van evolucionando los navegadores estos van facilitando más controles de interface de usuario que nos permiten un mayor control de los datos a introducir en un formulario.

11.14. Manipulación de Formularios con JavaScript

JavaScript se utiliza para interactuar con los formularios, realizar validaciones y manejar eventos. Aquí se presentan algunos ejemplos comunes de manipulación de formularios con JavaScript.

a) **Captura de Datos del Formulario:** Para capturar y procesar los datos del formulario, podemos usar JavaScript para escuchar el **evento submit.**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Formulario con JavaScript</title>
</head>
<body>
    <form id="myForm">
        <label for="username">Nombre de usuario:</label>
        <input type="text" id="username" name="username"><br>
        <label for="email">Correo electrónico:</label>
        <input type="email" id="email" name="email"><br>
        <button type="submit">Enviar</button>
    </form>

    <script>
        document.getElementById('myForm').addEventListener('submit',
function(event) {
            event.preventDefault(); // Evita el envío del formulario
            const username = document.getElementById('username').value;
            const email = document.getElementById('email').value;
            console.log('Nombre de usuario:', username);
            console.log('Correo electrónico:', email);
        });
    </script>
</body>
</html>
```

11.15. Introducción a la Validación de Formularios

La validación de formularios es un proceso crucial en el desarrollo web para asegurar que los datos ingresados por los usuarios cumplan con ciertos criterios antes de ser enviados al servidor. La validación puede llevarse a cabo tanto del lado del cliente (en el navegador) como del lado del servidor. La validación del lado del cliente proporciona una respuesta inmediata a los usuarios y reduce la carga del servidor.

a) Validación del Lado del Cliente

La validación del lado del cliente se realiza utilizando JavaScript y se ejecuta antes de que el formulario sea enviado al servidor. Esto mejora la experiencia del usuario al proporcionar retroalimentación inmediata.

b) Validación del Lado del Servidor

La validación del lado del servidor se realiza una vez que los datos han sido enviados al servidor. Aunque es más segura que la validación del lado del cliente, se recomienda usar ambos tipos de validación para garantizar la máxima seguridad y usabilidad.

c) Validación HTML5

HTML5 introduce varios atributos de validación que pueden ser utilizados sin necesidad de JavaScript:

- **required:** Hace que un campo sea obligatorio.
- **minlength y maxlength:** Especifican la longitud mínima y máxima de un campo de texto.
- **pattern:** Define una expresión regular que el valor del campo debe cumplir.
- **type:** Especifica el tipo de datos esperados (e.g., `email`, `number`, `url`).

```
<form id="miFormulario">
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" minlength="8"
required>
  <button type="submit">Enviar</button>
</form>
```

11.16. Validación con JavaScript

La validación de formularios con JavaScript consiste en verificar que los datos ingresados en un formulario cumplen con ciertos requisitos antes de enviarlo al servidor. Esto se hace para evitar errores, mejorar la experiencia del usuario y reducir la carga en el servidor, ya que los errores pueden corregirse inmediatamente sin necesidad de una solicitud al servidor.

Esta validación se puede hacer desde varios enfoques:

a) Validación Básica

```
<form id="miFormulario">
```

```
  <label for="email">Email:</label>
  <input type="email" id="email" name="email">
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password">
  <button type="submit">Enviar</button>
  <div id="errores" style="color: red;"></div>
</form>
<script>
  document.getElementById('miFormulario').addEventListener('submit',
function(event) {
    let errores = [];
    const email = document.getElementById('email').value;
    const password = document.getElementById('password').value;

    if (!email) {
      errores.push('El email es obligatorio.');
    }
    if (!password || password.length < 8) {
      errores.push('La contraseña debe tener al menos 8 caracteres.');
    }

    if (errores.length > 0) {
      event.preventDefault(); // Previene el envío del formulario
      document.getElementById('errores').innerHTML =
errores.join('<br>');
    }
  });
</script>
```

b) Validación Avanzada con Expresiones Regulares

```
<form id="miFormulario">
  <label for="username">Nombre de Usuario:</label>
  <input type="text" id="username" name="username" required>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" required>
  <button type="submit">Enviar</button>
  <div id="errores" style="color: red;"></div>
</form>
<script>
  document.getElementById('miFormulario').addEventListener('submit',
function(event) {
    let errores = [];
```

```
    const username = document.getElementById('username').value;
    const email = document.getElementById('email').value;
    const password = document.getElementById('password').value;

    const usernamePattern = /^[a-zA-Z0-9]{5,15}$/; // Solo letras y
números, entre 5 y 15 caracteres
    if (!usernamePattern.test(username)) {
      errores.push('El nombre de usuario debe tener entre 5 y 15
caracteres y solo contener letras y números.');
    }

    const emailPattern =
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
    if (!emailPattern.test(email)) {
      errores.push('El email no es válido.');
    }

    if (!password || password.length < 8) {
      errores.push('La contraseña debe tener al menos 8 caracteres.');
    }

    if (errores.length > 0) {
      event.preventDefault();
      document.getElementById('errores').innerHTML =
errores.join('<br>');
    }
  });
</script>
```

c) Feedback Visual

Proporcionar retroalimentación visual puede mejorar significativamente la experiencia del usuario. Puedes usar estilos CSS y JavaScript para resaltar campos con errores.

```
<style>
  .error {
    border-color: red;
  }
</style>
<form id="miFormulario">
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" required>
  <button type="submit">Enviar</button>
```

```
  <div id="errores" style="color: red;"></div>
</form>
<script>
  document.getElementById('miFormulario').addEventListener('submit',
function(event) {
    let errores = [];
    const email = document.getElementById('email');
    const password = document.getElementById('password');

    if (!email.value) {
      errores.push('El email es obligatorio.');
      email.classList.add('error');
    } else {
      email.classList.remove('error');
    }

    if (!password.value || password.value.length < 8) {
      errores.push('La contraseña debe tener al menos 8 caracteres.');
      password.classList.add('error');
    } else {
      password.classList.remove('error');
    }

    if (errores.length > 0) {
      event.preventDefault();
      document.getElementById('errores').innerHTML =
errores.join('<br>');
    }
  });
</script>
```

11.17. Validación con APIs del Navegador

Los navegadores modernos ofrecen APIs como `setCustomValidity` y `checkValidity` para manejar la validación de formularios de manera más flexible.

```
<form id="miFormulario">
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>
  <label for="password">Contraseña:</label>
  <input type="password" id="password" name="password" required>
  <button type="submit">Enviar</button>
</form>
<script>
  document.getElementById('miFormulario').addEventListener('submit',
```

```
function(event) {
    const email = document.getElementById('email');
    const password = document.getElementById('password');

    if (!email.checkValidity()) {
      email.setCustomValidity('Por favor, ingrese un email válido.');
      email.reportValidity();
      event.preventDefault();
    } else {
      email.setCustomValidity('');
    }

    if (password.value.length < 8) {
      password.setCustomValidity('La contraseña debe tener al menos 8
caracteres.');
      password.reportValidity();
      event.preventDefault();
    } else {
      password.setCustomValidity('');
    }
  });
</script>
```

11.18 Ejercicios

Ejercicio 1: Validación de Formulario de Registro

Crea un formulario de registro con los campos "Nombre", "Correo Electrónico", "Contraseña" y "Confirmar Contraseña". Usa JavaScript para validar que:

- Todos los campos estén llenos.
- El campo de "Correo Electrónico" contenga un formato de correo válido.
- La "Contraseña" y "Confirmar Contraseña" coincidan.

Si alguna validación falla, muestra un mensaje de error debajo del campo correspondiente.

Pista: Usa expresiones regulares para validar el correo y compara los valores de los campos de contraseña.

Ejercicio 2: Formulario de Encuesta

Crea un formulario de encuesta con una pregunta y las opciones SI, NO, NS/NC. Al hacer clic en el botón "Votar", usa JavaScript para recopilar las respuestas y mostrarlas en un div debajo del formulario junto con el porcentaje y el total de votos así como una barra de progreso que muestre el porcentaje.

Ejercicio 3: Calculadora de IMC

Crea un formulario que permita a los usuarios ingresar su peso y altura. Usa JavaScript para calcular el Índice de Masa Corporal (IMC) al hacer clic en un botón "Calcular". Muestra el resultado del IMC en la página e indica si el usuario tiene un peso bajo, normal, sobrepeso o es obeso según su IMC.

Pista: La fórmula del IMC es `IMC = peso (kg) / altura (m)^2`.

Ejercicio 4: Formulario de Contacto con Contador de Caracteres

Crea un formulario de contacto con un campo de "Mensaje". Agrega un contador de caracteres que muestre cuántos caracteres quedan antes de alcanzar el límite de 200 caracteres. Si el usuario excede el límite, muestra un mensaje de error y evita que se pueda enviar el formulario.

Pista: Usa el evento `input` en el campo de mensaje para actualizar el contador dinámicamente.

Ejercicio 5: Formulario Dinámico de Reservas

Crea un formulario de reservas de hotel con campos para seleccionar la fecha de entrada, fecha de salida y número de personas. Usa JavaScript para:

- Calcular y mostrar el número de noches entre las fechas de entrada y salida.
- Actualizar el costo total de la reservación basado en un precio fijo por noche y el número de personas.

Muestra el costo total en tiempo real conforme el usuario actualiza las fechas o el número de personas.

Pista: Usa el objeto `Date` para calcular la diferencia de días entre las fechas y `addEventListener` para escuchar los cambios en los campos del formulario.

12. Fundamentos de la asincronía en Javascript.

JavaScript es un lenguaje de programación de un solo hilo, lo que significa que solo puede realizar una tarea a la vez. Sin embargo, con la asincronía, JavaScript puede manejar múltiples tareas simultáneamente sin bloquear el hilo principal, proporcionando una experiencia de usuario más fluida y receptiva.

12.1. Callbacks:

Los callbacks son funciones que se pasan como argumentos a otras funciones y se ejecutan después de que se completa una operación asincrónica.

```
console.log('Inicio');

   setTimeout(() => {
       console.log('Tarea asincrónica completada');
   }, 2000);

   console.log('Fin');
```

En este ejemplo, setTimeout es una función asincrónica que ejecuta el callback después de 2 segundos, permitiendo que el código continúe ejecutándose sin esperar.

12.2. Promesas.

Las promesas representan un valor que puede estar disponible ahora, en el futuro o nunca. Proporcionan un enfoque más manejable y legible que los callbacks, especialmente cuando se encadenan múltiples operaciones asincrónicas.

```
  const promesa = new Promise((resolve, reject) => {
       setTimeout(() => {
           resolve('Promesa resuelta');
       }, 2000);
   });

   promesa.then((mensaje) => {
       console.log(mensaje);
   }).catch((error) => {
       console.error(error);
   });
```

Aquí, la promesa se resuelve después de 2 segundos, y el método `then` maneja el resultado.

12.3. Async/Await.

Introducidos en ECMAScript 2017 (ES8), **async** y **await** proporcionan una forma más simple y legible de trabajar con código asincrónico. Permiten escribir código asíncrono que se parece y se comporta como código síncrono.

```
const esperar = () => {
        return new Promise((resolve) => {
            setTimeout(() => {
                resolve('Esperar completado');
            }, 2000);
        });
    };

    const funcionAsincrona = async () => {
        console.log('Inicio');
        const resultado = await esperar();
        console.log(resultado);
        console.log('Fin');
    };

    funcionAsincrona();
```

En este ejemplo, `await` se usa para esperar a que la promesa se resuelva antes de continuar con la ejecución del código. Esto hace que el código sea más fácil de leer y entender en comparación con el uso de `then`.

12.4. Casos de Uso Comunes.

a) **Solicitudes HTTP:** Realizar solicitudes a servidores remotos para obtener o enviar datos.

```
const obtenerDatos = async () => {
        try {
            const respuesta = await
fetch('https://api.example.com/datos');
            const datos = await respuesta.json();
            console.log(datos);
        } catch (error) {
            console.error('Error:', error);
        }
    };

    obtenerDatos();
```

b) **Operaciones de Entrada/Salida (I/O):** Leer o escribir archivos, interactuar con bases de datos, etc.

```
const fs = require('fs').promises;
const leerArchivo = async () => {
        try {
            const datos = await fs.readFile('archivo.txt', 'utf8');
            console.log(datos);
        } catch (error) {
            console.error('Error al leer el archivo:', error);
        }
    };

    leerArchivo();
```

c) **Manejando Errores Asincrónicos.**

Es crucial manejar errores en código asincrónico para evitar comportamientos inesperados y depuración difícil. `try/catch` es comúnmente utilizado con `async/await`:

```
const funcionAsincrona = async () => {
    try {
        const resultado = await esperar();
        console.log(resultado);
    } catch (error) {
        console.error('Error:', error);
    }
};

funcionAsincrona();
```

13. Fundamentos de AJAX

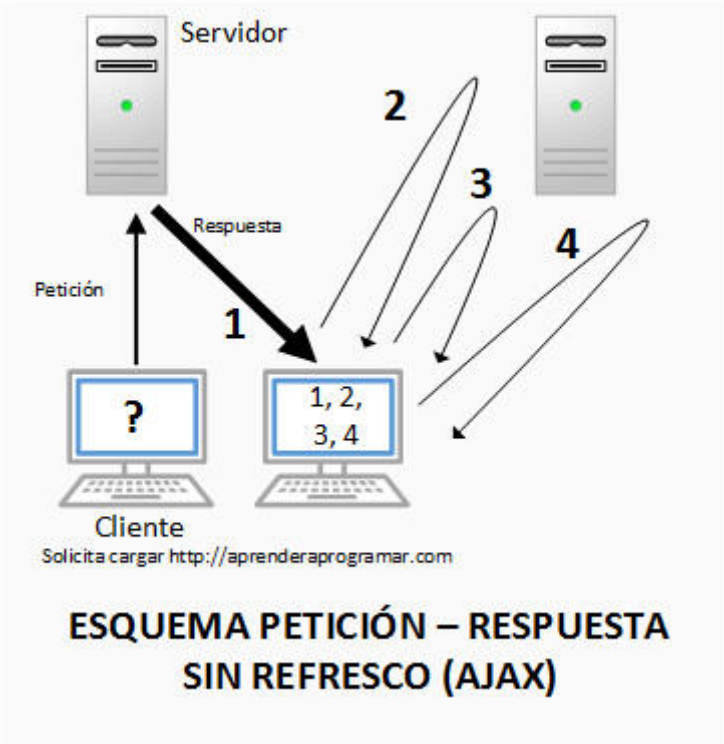
AJAX (Asynchronous JavaScript and XML) es una técnica de desarrollo web que permite a las aplicaciones web enviar y recibir datos del servidor de forma asíncrona, sin tener que recargar toda la página. Esto mejora la experiencia del usuario al hacer que las aplicaciones sean más rápidas y receptivas.

Funcionamiento:

1. Solicitud al servidor: Se utiliza JavaScript para enviar una solicitud HTTP (normalmente GET o POST) al servidor.

2. Respuesta del servidor: El servidor procesa la solicitud y devuelve una respuesta, que puede estar en varios formatos, como XML, JSON, HTML o texto plano.

3. Actualización de la página web: JavaScript maneja la respuesta y actualiza solo la parte necesaria de la página sin recargarla completamente.



Ventajas:

- Mejora de la experiencia del usuario: Al actualizar solo partes específicas de la página, se reduce el tiempo de carga y la interacción es más fluida.

- Eficiencia en el tráfico: Solo se envían y reciben datos relevantes, lo que minimiza el tráfico innecesario entre el cliente y el servidor.

Imagina que estás llenando un formulario y necesitas seleccionar una ciudad después de elegir un país. AJAX podría enviar una solicitud al servidor para obtener las ciudades sin recargar la página, actualizando solo el campo de selección de ciudades mediante dos <selects> dependientes.

Aunque originalmente utilizaba XML, hoy en día es más común que las aplicaciones utilicen JSON como formato de intercambio de datos, debido a su simplicidad y compatibilidad con JavaScript.

AJAX por tanto no es una tecnología específica, sino una combinación de varias tecnologías:

- HTML/XHTML y CSS: Para la presentación.
- DOM: Para la interacción dinámica con la página.
- JavaScript: Para realizar solicitudes asíncronas.
- XML/JSON: Para intercambiar datos entre el servidor y el cliente.

13.1. Implementación de AJAX.

Hay varias formas de implementar AJAX en JavaScript:

a) **XMLHttpRequest (XHR):** El método original para realizar solicitudes AJAX.

```
const xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://api.example.com/data', true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4 && xhr.status === 200) {
            const response = JSON.parse(xhr.responseText);
            console.log(response);
        }
    };

    xhr.send();
```

En este ejemplo:

- **XMLHttpRequest** se utiliza para crear un objeto de solicitud.

- **open** inicializa la solicitud.

- **onreadystatechange** es un manejador de eventos que se llama cuando cambia el estado de la solicitud.

- **send** envía la solicitud al servidor.

b) **Fetch API:** La Fetch API es una interfaz moderna de JavaScript para realizar solicitudes HTTP desde el navegador. Ofrece una forma más sencilla y flexible de interactuar con recursos en la web en comparación con la antigua API XMLHttpRequest.

Características Principales de la Fetch API:

Promesas: Fetch utiliza promesas, lo que facilita el manejo de operaciones asíncronas en comparación con los callbacks tradicionales de XMLHttpRequest. Una promesa (o promise) en JavaScript es un objeto que representa la eventual finalización (o falla) de una operación asíncrona y su valor resultante. Es una forma de manejar tareas asíncronas, como solicitudes AJAX, de una manera más organizada y predecible en lugar de depender de callbacks o funciones de retroceso. Las promesas permiten una sintaxis más clara y manejable con .then() y .catch() para el tratamiento de respuestas y errores.

Interfaz Limpia y Sencilla: La API está diseñada para ser simple y fácil de usar. Se puede configurar con diversas opciones para hacer solicitudes GET, POST, PUT, DELETE, entre otras.

Soporte para CORS: La Fetch API maneja de forma nativa las solicitudes cross-origin (CORS), lo cual es útil para trabajar con APIs externas.

```
fetch('https://api.example.com/data') //Colocamos la web que nos
devuelve los datos
        .then(response => {
            if (!response.ok) {
                throw new Error('Network response was not ok');
            }
            return response.json();
        })
        .then(data => console.log(data))
.catch(error => console.error('Existe un problema con su operación:',
error));
```

En este ejemplo:

- **fetch** realiza una solicitud HTTP y devuelve una **promesa.**

- **response.json()** convierte la respuesta en un objeto JavaScript.

- **then** maneja la respuesta exitosa.

- **catch** maneja cualquier error que ocurra durante la solicitud.

13.2. Técnicas Avanzadas con Fetch API

a) **Solicitudes POST:** Para enviar datos al servidor, como formularios.

```
const data = { username: 'example' };

    fetch('https://api.example.com/submit', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
    })
    .then(response => response.json())
    .then(data => console.log('Success:', data))
    .catch(error => console.error('Error:', error));
```

En este ejemplo:

- **method:** “POST” especifica el método HTTP.

- **headers** define el tipo de contenido de la solicitud.

- **body** contiene los datos que se enviarán, convertidos a JSON.

b) **Manejo de Errores:** Es importante manejar errores en solicitudes HTTP para una mejor experiencia de usuario.

```
 fetch('https://api.example.com/data')
        .then(response => {
            if (!response.ok) {
                throw new Error('Network response was not ok');
            }
            return response.json();
        })
        .then(data => console.log(data))
        .catch(error => {
            console.error('There has been a problem with your fetch
operation:', error);
            // Aquí puedes agregar lógica para manejar el error, como
mostrar un mensaje al usuario.
        });
```

13.3. Uso de Librerías AJAX

Además de XMLHttpRequest y Fetch API, existen varias librerías que simplifican el uso de AJAX:

a) **jQuery:** es una forma muy común y sencilla de hacer solicitudes asíncronas en aplicaciones web. jQuery, una popular librería de JavaScript, simplifica mucho la escritura y gestión de las operaciones AJAX, proporcionando una API más limpia y fácil de usar en comparación con `XMLHttpRequest`.

   jQuery proporciona varios métodos para trabajar con AJAX, como $.ajax(), $.get(), $.post(), y otros que hacen el manejo de peticiones asíncronas más accesible.

   Métodos comunes de AJAX en jQuery:

   1. $.ajax(): Es el método más versátil y permite configurar la solicitud con diversas opciones.

   2. $.get(): Método simplificado para hacer solicitudes HTTP GET.

   3. $.post(): Método simplificado para hacer solicitudes HTTP POST.

   4. $.data(): Método para repoblar el contenido de una etiqueta con lo que nos devuelva el servidor.

   El método **$.ajax()** es el más completo, ya que permite un control detallado sobre la solicitud.

```
$.ajax({
  url: "https://api.ejemplo.com/datos", // URL a la que se envía la
solicitud
  type: "GET", // Tipo de solicitud (GET, POST, etc.)
  dataType: "json", // El tipo de dato que esperas de la respuesta
(JSON, XML, HTML, etc.)
  success: function (respuesta) {
    console.log(respuesta); // Manejo de la respuesta exitosa
  },
  error: function (xhr, status, error) {
    console.log("Error: " + error); // Manejo de errores
  }
});
```

En este ejemplo, la solicitud se realiza a la URL dada. Si la solicitud es exitosa, el callback success maneja la respuesta. Si hay un error, el callback `error` se ejecuta.

El método **$.get()** es una versión más simple de $.ajax() para hacer solicitudes HTTP GET.

```
$.get("https://api.ejemplo.com/datos", function (respuesta) {
  console.log(respuesta); // Se maneja la respuesta aquí
});
```

Este método simplifica la solicitud GET en una sola línea. La URL es el primer argumento, y el segundo es la función callback que maneja la respuesta.

El método **$.post()** es muy útil cuando necesitas enviar datos al servidor, como en formularios.

```
$.post("https://api.ejemplo.com/enviar", { nombre: "Juan", edad: 30 },
function (respuesta) {
  console.log(respuesta); // Respuesta manejada aquí
});
```

En este ejemplo, se envían los datos { nombre: "Juan", edad: 30 } al servidor usando POST, y se maneja la respuesta en el callback.

Opciones comunes en $.ajax():

- **url:** La URL a la que se enviará la solicitud.

- **type:** El tipo de solicitud HTTP (GET, POST, PUT, DELETE).

- **data:** Los datos que se enviarán al servidor.

- **dataType:** El tipo de datos esperados en la respuesta (json, xml, html, etc.).

- **success:** Una función callback que se ejecuta cuando la solicitud tiene éxito.

- **error:** Una función callback que se ejecuta si ocurre un error.

Ejemplo con un formulario y POST:

En este ejemplo, los datos de un formulario se envían a un servidor utilizando una solicitud POST con `$.ajax()`. `e.preventDefault()` evita que el formulario envíe los datos de la manera tradicional (con recarga de página).

```
<form id="miFormulario">
  Nombre: <input type="text" name="nombre" /><br />
  Edad: <input type="number" name="edad" /><br />
  <input type="submit" value="Enviar" />
</form>

<script>
```

```
  $("#miFormulario").submit(function (e) {
    e.preventDefault(); // Evita que el formulario se envíe de manera
tradicional

    $.ajax({
      url: "https://api.ejemplo.com/enviar", // URL de destino
      type: "POST",
      data: $(this).serialize(), // Convierte los datos del formulario a
un formato adecuado
      success: function (respuesta) {
        console.log("Datos enviados correctamente: ", respuesta);
      },
      error: function (xhr, status, error) {
        console.log("Error al enviar los datos: " + error);
      }
    });
  });
</script>
```

El método $.load() realiza una solicitud GET al servidor y carga el contenido HTML recibido en un elemento seleccionado. Es útil cuando quieres actualizar una parte específica de tu página sin tener que escribir manualmente el código AJAX completo.

```
$(selector).load(url, [data], [callback]);
```

- **selector:** El selector jQuery del elemento donde se insertará el contenido.

- **url:** La URL a la que se realizará la solicitud. Puede ser una URL de un archivo HTML o un script del servidor que devuelve HTML.

- **data** (opcional): Un objeto de datos o una cadena de consulta que se enviará al servidor como parámetros de la solicitud.

- **callback** (opcional): Una función que se ejecutará una vez que la solicitud y el contenido hayan sido cargados. Esta función recibe dos parámetros: `response` (la respuesta del servidor) y `status` (el estado de la solicitud).

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

<script>
  $("#miDiv").load("contenido.html");
</script>
```

En este ejemplo, el contenido del archivo contenido.html se cargará dentro del div con el ID miDiv.

Ventajas de usar AJAX con jQuery:

1. Sintaxis más simple y clara: jQuery reduce mucho el código que necesitas escribir en comparación con `XMLHttpRequest`.

2. Compatibilidad entre navegadores: jQuery maneja las diferencias entre los navegadores automáticamente, asegurando que las solicitudes funcionen de manera consistente.

3. Manejo de eventos simplificado: jQuery proporciona eventos y callbacks más fáciles de usar para manejar el éxito, error y otros estados de las solicitudes AJAX.

Usar jQuery para AJAX es muy útil cuando se busca simplificar las solicitudes asíncronas en aplicaciones web. Aunque en la actualidad la API fetch es una alternativa moderna, jQuery sigue siendo muy utilizada debido a su simplicidad y soporte a navegadores más antiguos.

b) **Axios:** es una librería basada en promesas para realizar solicitudes HTTP, más moderna y con soporte para Node.js.

```
axios.get('https://api.example.com/data')
        .then(response => {
            console.log(response.data);
        })
        .catch(error => {
            console.error('Error:', error);
        });
```

Para solicitudes POST con Axios:

```
const data = { username: 'example' };

   axios.post('https://api.example.com/submit', data)
        .then(response => {
            console.log('Success:', response.data);
        })
        .catch(error => {
            console.error('Error:', error);
        });
```

13.4. Ejercicios

Ejercicio 1: Carga de Contenido Dinámico

Crea una página web con un botón "Cargar Noticias". Al hacer clic en el botón, usa AJAX para hacer una solicitud a un archivo JSON que contenga una lista de noticias (título,

resumen y enlace). Muestra las noticias en la página sin recargarla. Utiliza la API JSONPlaceholder.

Ejercicio 2: Formulario de Búsqueda en Vivo

Crea un formulario de búsqueda en una página web con un campo de texto "Buscar". A medida que el usuario escribe en el campo, usa AJAX para enviar la consulta a un servidor que devuelve resultados de búsqueda en JSON. Muestra los resultados en tiempo real debajo del campo de búsqueda.

Pista: Usa el evento input para detectar cuando el usuario escribe, y fetch o XMLHttpRequest para hacer la solicitud al servidor.

Ejercicio 3: Autocompletado de Campo de Formulario

Implementa un campo de texto para ingresar nombres de ciudades. Usa AJAX para hacer una solicitud a una API (por ejemplo, de OpenWeatherMap o una similar) que devuelva sugerencias de ciudades a medida que el usuario escribe. Muestra las sugerencias en una lista desplegable debajo del campo de texto.

Pista: Usa el evento `input` para capturar el texto y realizar la solicitud AJAX, actualizando la lista de sugerencias con cada cambio.

Ejercicio 4: Envío de Formulario sin Recargar la Página

Crea un formulario de contacto con campos para "Nombre", "Correo Electrónico" y "Mensaje". Usa AJAX para enviar los datos del formulario a un servidor cuando el usuario haga clic en "Enviar". Muestra un mensaje de éxito o error en la misma página sin recargar.

Pista: Captura el evento `submit` del formulario, evita el comportamiento por defecto con `event.preventDefault()`, y luego usa `fetch` o `XMLHttpRequest` para enviar los datos del formulario.

Ejercicio 5: Actualización Dinámica de Datos

Crea una página que muestre una lista de usuarios con sus nombres y correos electrónicos. Usa AJAX para actualizar la lista cada 30 segundos, obteniendo los datos de un servidor que devuelve la lista de usuarios en JSON. Muestra la lista en la página, actualizándose automáticamente sin recargar la página.

Pista: Usa `setInterval` para programar la actualización automática, y `fetch` para realizar la solicitud AJAX periódica para actualizar el DOM con los nuevos datos.

14. El Formato JSON

14.1. Definición

JSON (JavaScript Object Notation) es un formato de texto ligero utilizado para intercambiar datos. Se basa en un subconjunto de la notación de objetos de JavaScript y es ampliamente utilizado debido a su simplicidad y legibilidad. JSON es independiente del lenguaje, lo que significa que puede ser utilizado y generado por cualquier lenguaje de programación.

Fué introducido por Douglas Crockford a principios de los años 2000 como una alternativa más simple a XML para la transmisión de datos entre un cliente y un servidor.

Características principales:

- Ligero: JSON es un formato de texto simple y fácil de leer.
- Estructurado: Utiliza una estructura basada en pares clave-valor y listas ordenadas.
- Legible: Es fácil de entender tanto para humanos como para máquinas.
- Soporte universal: JSON es soportado nativamente en la mayoría de los lenguajes de programación.

14.2. Estructura de JSON

En JSON, un objeto es una colección de pares clave-valor. Se escribe entre llaves {}.

```
{
"nombre": "Juan",
"edad": 30,
"esEstudiante": true
}
```

Un array en JSON es una lista ordenada de valores. Se escribe entre corchetes `[]`.

```
{
"nombres": ["Juan", "María", "Carlos"]
}
```

Los valores en JSON pueden ser de los siguientes tipos: string, number, boolean, array, object y null.

```
{
"cadena": "Hola",
"numero": 25,
"booleano": true,
"nulo": null
```

```
}
```

14.3. Uso de JSON en JavaScript

a) Convertir objetos JavaScript a JSON: Para convertir un objeto JavaScript a una cadena JSON, se utiliza el método JSON.stringify().

```
const persona = { nombre: "Juan", edad: 30 };
      const jsonString = JSON.stringify(persona);
      console.log(jsonString); // {"nombre":"Juan","edad":30};
```

b) Convertir JSON a objetos JavaScript: Para convertir una cadena JSON a un objeto JavaScript, se utiliza el método JSON.parse().

```
      const jsonString = '{"nombre":"Juan","edad":30}';
      const persona = JSON.parse(jsonString);
      console.log(persona.nombre); // Juan
```

JSON es comúnmente utilizado en aplicaciones web para enviar y recibir datos a través de solicitudes HTTP. Por ejemplo, al hacer una solicitud `fetch` a una API, la respuesta suele estar en formato JSON como ya vimos en apartados anteriores.

14.4. Buenas prácticas con JSON

a) Validación de JSON: Siempre debes validar JSON antes de utilizarlo, especialmente si proviene de una fuente externa. Hay varias herramientas en línea y librerías en JavaScript para validar JSON.

b) Manejo de errores: Siempre rodea el uso de `JSON.parse()` con un bloque `try-catch` para manejar posibles errores en caso de que el JSON no sea válido.

```
try {
        const persona = JSON.parse('{"nombre":"Juan","edad":30}');
      } catch (error) {
        console.error('JSON inválido:', error);
      }
```

c) Minificación y legibilidad: Durante el desarrollo, es útil trabajar con JSON en un formato legible, con indentación y saltos de línea. Sin embargo, al enviar datos a través de la red, es común minificar el JSON para reducir el tamaño del payload.

```
const jsonString = JSON.stringify(persona, null, 2); // null para el
reemplazo, 2 para la cantidad de espacios
```

14.5. Comparación con otros formatos de datos

a) **JSON vs. XML:** JSON es más simple, legible y ligero comparado con XML. Es más legible y fácil de entender. Es nativo en JavaScript, mientras que XML requiere procesamiento adicional.

b) **JSON vs. YAML:** JSON es más sencillo y directo que YAML, que tiene una sintaxis más flexible pero puede ser más propenso a errores de sintaxis.

14.6. JSON en el mundo real

a) **APIs y JSON:** Es el formato preferido para el intercambio de datos en aplicaciones web y servicios web. La mayoría de las APIs RESTful utilizan JSON como formato de respuesta.

b) **Archivos de configuración:** se usa comúnmente en archivos de configuración debido a su simplicidad y legibilidad. Ejemplos incluyen archivos `package.json` en proyectos Node.js.

c) **Almacenamiento de datos:** JSON es utilizado en bases de datos NoSQL como MongoDB, donde los datos son almacenados en un formato muy similar a JSON (BSON).

14.7. Ejercicios

Ejercicio 1: Parseo y acceso a datos JSON

Disponemos de un archivo JSON que contiene información sobre estudiantes de una universidad. El formato del JSON es el siguiente:

```
{
  "estudiantes": [
    { "nombre": "Juan", "edad": 21, "carrera": "Ingeniería" },
    { "nombre": "María", "edad": 22, "carrera": "Matemáticas" },
    { "nombre": "Carlos", "edad": 20, "carrera": "Física" }
  ]
}
```

Escribe un programa en JavaScript que:

- Parsee el archivo JSON.
- Acceda a la lista de estudiantes.
- Imprima en la consola el nombre y la carrera de cada estudiante.

Ejercicio 2: Generar JSON a partir de objetos

Crea una aplicación que permita a los usuarios ingresar información sobre diferentes libros. Cada libro tiene un título, autor y año de publicación.

La aplicación debe tener los siguientes requisitos:

- Debe recibir la información del libro a través de un formulario.
- Genere un objeto en JavaScript con la información del libro.
- Convierta ese objeto a formato JSON y lo muestre en la consola.

Ejercicio 3: Manipulación de datos JSON

Tienes el siguiente JSON que contiene una lista de productos de una tienda:

```
{
  "productos": [
    { "nombre": "Laptop", "precio": 800, "cantidad": 10 },
    { "nombre": "Teléfono", "precio": 600, "cantidad": 25 },
    { "nombre": "Tablet", "precio": 400, "cantidad": 15 }
  ]
}
```

Escribe un programa en JavaScript que:

- Parsee el JSON.
- Calcule el valor total de inventario de cada producto (precio * cantidad).
- Imprima el valor total de inventario para cada producto en una etiqueta .

Ejercicio 4: Envío de datos con JSON a través de AJAX

Imagina que tienes un formulario donde los usuarios ingresan su nombre, correo electrónico y un mensaje. La información debe enviarse al servidor utilizando AJAX, con el contenido codificado en JSON.

Escribe un programa que:

- Capture los datos del formulario.
- Cree un objeto JSON con los datos ingresados.
- Envíe el objeto JSON a una URL de tu elección usando AJAX ($.ajax() de jQuery o fetch).
- Maneje la respuesta del servidor e imprima un mensaje de éxito o error en la consola.

Ejercicio 5: Filtrar datos de un JSON

Tenemos un archivo JSON con la información de varias películas en una base de datos:

```
{
  "peliculas": [
```

```
    { "titulo": "Matrix", "genero": "Ciencia ficción", "año": 1999 },
    { "titulo": "El Padrino", "genero": "Drama", "año": 1972 },
    { "titulo": "Star Wars", "genero": "Ciencia ficción", "año": 1977 },
    { "titulo": "Titanic", "genero": "Romance", "año": 1997 }
  ]
}
```

Escribe un programa en JavaScript que:
- Parsee el JSON.
- Filtre y muestre solo las películas del género "Ciencia ficción".
- Imprima en la consola el título y año de las películas que cumplan con ese criterio.

15. Librerías adicionales

15.1. JQuery

jQuery es una biblioteca de JavaScript ligera y rápida, diseñada para simplificar el uso de JavaScript en la creación de sitios web interactivos.
Fue creada para facilitar tareas comunes como manipulación del DOM, manejo de eventos, efectos animados y manejo de AJAX.

Ventajas de usar jQuery:
- Sintaxis simple y fácil de aprender.
- Compatible con todos los navegadores modernos.
- Permite una manipulación más sencilla del DOM.
- Facilita la creación de efectos animados.
- Amplia comunidad y soporte, con abundante documentación y plugins disponibles.

Cómo incluir jQuery:
Puedes incluir jQuery en tu proyecto a través de una CDN (Content Delivery Network) o descargando la biblioteca localmente.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Manual de jQuery</title>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>

</body>
</html>
```

Estructura básica de jQuery:

```
$(document).ready(function(){
    // Aquí va el código jQuery
});
```

La función `$(document).ready()` asegura que el código jQuery se ejecute una vez que el documento esté completamente cargado.

Manipulación del DOM con jQuery:

jQuery utiliza la sintaxis de selectores de CSS para identificar elementos en el DOM.

```
// Seleccionar todos los párrafos
$('p');

// Seleccionar un elemento con un ID específico
$('#miElemento');

// Seleccionar elementos por clase
$('.miClase');
```

Manipulación del contenido:
- **text():** Cambia o recupera el texto de un elemento.
- **html():** Cambia o recupera el HTML de un elemento.
- **val():** Cambia o recupera el valor de un input.

```
// Cambiar el texto de un párrafo
$('p').text('Nuevo texto');

// Obtener el texto de un párrafo
var texto = $('p').text();

// Cambiar el contenido HTML de un div
$('#miDiv').html('<strong>Texto en negrita</strong>');
```

Manipulación de atributos:
- **attr():** Cambia o recupera el valor de un atributo.
- **removeAttr():** Elimina un atributo.

```
// Cambiar el atributo src de una imagen
$('img').attr('src', 'nueva-imagen.jpg');

// Eliminar el atributo disabled de un botón
$('button').removeAttr('disabled');
```

Manipulación de clases:
- **addClass():** Añade una clase a un elemento.
- **removeClass():** Elimina una clase de un elemento.
- **toggleClass():** Alterna una clase en un elemento.

```
// Añadir una clase a un párrafo
$('p').addClass('nuevoEstilo');

// Alternar una clase en un div
$('#miDiv').toggleClass('activo');
```

Manejo de eventos en jQuery:

- **click():** Maneja el evento de clic.
- **hover():** Maneja los eventos de pasar el mouse por encima y salir.
- **keyup():** Maneja el evento de soltar una tecla.

```
// Cambiar el color de fondo al hacer clic en un botón
$('#miBoton').click(function(){
    $('body').css('background-color', 'blue');
});

// Mostrar un mensaje cuando el mouse está sobre un div
$('#miDiv').hover(function(){
    alert('Estás sobre el div');
});
```

Efectos en jQuery: jQuery ofrece una serie de efectos para animar elementos en el DOM de una forma sencilla y eficiente.

- **hide() y show():** Ocultan o muestran elementos.
- **fadeIn() y fadeOut():** Desvanecen elementos.
- **slideDown() y slideUp():** Deslizan elementos hacia abajo o hacia arriba.

```
// Ocultar un párrafo con un desvanecimiento
$('p').fadeOut('slow');

// Mostrar un div con un deslizamiento
$('#miDiv').slideDown('fast');
```

Manejo de AJAX con jQuery:

jQuery simplifica las solicitudes AJAX para interactuar con servidores sin recargar la página.

```
$.ajax({
    url: 'datos.json',
    method: 'GET',
    success: function(data) {
        console.log(data);
    }
});
```

15.2. Moment

Moment.js es una biblioteca de JavaScript utilizada para manejar y manipular fechas y horas de manera fácil y eficiente. Es especialmente útil para formatear, validar, analizar y manipular fechas en diversas zonas horarias.

Incluir Moment.js en tu proyecto:

Puedes incluir Moment.js en tu proyecto utilizando una CDN o instalándolo con npm.

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.1/moment.min.
js"></script>
```

Usos Básicos de Moment.js:

- Crear un objeto de fecha con la fecha y hora actuales:

```
var ahora = moment();
```

- Crear un objeto de fecha para una fecha específica:

```
var fechaEspecifica = moment('2024-08-15');
```

Formateo de fechas:

var formateada = ahora.format('YYYY-MM-DD'); // '2024-08-15'
var formatoCompleto = ahora.format('dddd, MMMM Do YYYY, h:mm:ss a'); // 'Thursday, August 15th 2024, 12:00:00 am'

Manipulación de fechas:

- Añadir o restar tiempo:

```
var enUnaSemana = ahora.add(7, 'days');
var haceUnMes = ahora.subtract(1, 'months');
```

- Cambiar componentes específicos de la fecha:

```
var cambiarDia = ahora.set('date', 1); // Cambia el día al 1
```

Comparación de fechas:

```
var fecha1 = moment('2024-08-01');
var fecha2 = moment('2024-08-15');
var esAntes = fecha1.isBefore(fecha2); // true
var esDespues = fecha1.isAfter(fecha2); // false
var esIgual = fecha1.isSame(fecha2); // false
```

Funciones Útiles de Moment.js:

- Convertir a un objeto de fecha JavaScript:

```
var fechaJavaScript = ahora.toDate();
```

- Convertir a una cadena ISO:

```
var fechaISO = ahora.toISOString(); // '2024-08-15T00:00:00.000Z'
```

- Calcular la diferencia entre dos fechas:

```
var diferenciaDias = fecha2.diff(fecha1, 'days'); // 14 días
```

- Trabajar con duraciones:

```
var duracion = moment.duration(2, 'hours');
var sumaDuracion = ahora.add(duracion); // Añade 2 horas a la fecha
actual
```

15.3. Chart

Chart.js es una librería de JavaScript de código abierto que permite crear gráficos animados y visualmente atractivos en el navegador utilizando el elemento `<canvas>` de HTML5. Es fácil de usar, altamente personalizable y soporta varios tipos de gráficos, como gráficos de línea, barras, radar, torta, burbuja, dispersión, y más.

Incluir Chart.js en tu proyecto:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

Creación de un Gráfico Básico:

```
<canvas id="miGrafico" width="400" height="400"></canvas>
```

Creación de un gráfico de barras:

```
// Obtener el contexto del canvas
var ctx = document.getElementById('miGrafico').getContext('2d');

// Crear el gráfico
var miGrafico = new Chart(ctx, {
    type: 'bar', // Tipo de gráfico
    data: {
        labels: ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo'], //
Etiquetas del eje X
        datasets: [{
            label: 'Ventas',
            data: [12, 19, 3, 5, 2], // Datos del eje Y
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)',
                'rgba(54, 162, 235, 0.2)',
                'rgba(255, 206, 86, 0.2)',
                'rgba(75, 192, 192, 0.2)',
                'rgba(153, 102, 255, 0.2)'
            ],
            borderColor: [
                'rgba(255, 99, 132, 1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 206, 86, 1)',
                'rgba(75, 192, 192, 1)',
                'rgba(153, 102, 255, 1)'
            ],
            borderWidth: 1
        }]
    },
    options: {
        scales: {
            y: {
                beginAtZero: true
            }
        }
    }
});
```

Personalización y Opciones:

Tipos de gráficos soportados:
- bar: Gráfico de barras
- line: Gráfico de líneas
- pie: Gráfico de torta
- doughnut: Gráfico de dona

- radar: Gráfico de radar
- polarArea: Gráfico de área polar
- bubble: Gráfico de burbujas
- scatter: Gráfico de dispersión

Puedes personalizar prácticamente cualquier aspecto del gráfico, como colores, tamaños de fuentes, leyendas, y mucho más.

```
var miGrafico = new Chart(ctx, {
    type: 'line',
    data: { /* datos */ },
    options: {
        responsive: true,
        plugins: {
            legend: {
                display: true,
                position: 'top',
            },
            tooltip: {
                enabled: true
            }
        },
        scales: {
            x: {
                display: true,
                title: {
                    display: true,
                    text: 'Meses'
                }
            },
            y: {
                display: true,
                title: {
                    display: true,
                    text: 'Cantidad'
                },
                beginAtZero: true
            }
        }
    }
});
```

Actualización de gráficos: Puedes actualizar el gráfico dinámicamente si cambian los datos:

```
miGrafico.data.datasets[0].data = [nuevosDatos];
miGrafico.update();
```

16. Frameworks

En el desarrollo web moderno, Vue.js, React, y Angular se han consolidado como los tres frameworks más populares y poderosos para construir interfaces de usuario dinámicas y aplicaciones web de una sola página (SPA). Cada uno de ellos tiene sus propias fortalezas, enfoques únicos y comunidades robustas. A continuación, exploraremos una visión general de cada uno, sus características clave, y cómo se comparan entre sí.

16.1. Vue.js

Vue.js es un framework progresivo de JavaScript que se centra en la capa de vista. Fue creado por Evan You en 2014 y ha ganado popularidad rápidamente debido a su simplicidad, flexibilidad y capacidad de integración con otros proyectos.

Características Principales:

- **Facilidad de uso:** Vue.js es intuitivo y fácil de aprender, especialmente para desarrolladores con conocimientos en HTML, CSS y JavaScript.

- **Reactividad:** Vue ofrece un sistema reactivo eficiente que actualiza automáticamente la vista cuando los datos cambian.

- **Componentes:** Los componentes en Vue.js permiten una organización modular y reutilizable del código.

- **Progresividad:** Vue puede ser adoptado de forma incremental, desde mejorar partes de una aplicación hasta construir una SPA completa.

Cuándo usar Vue.js:

- Ideal para proyectos pequeños y medianos donde la simplicidad y la rapidez de desarrollo son cruciales.

- Cuando se necesita integrar gradualmente nuevas funcionalidades en aplicaciones existentes.

16.2. React

React es una biblioteca de JavaScript para construir interfaces de usuario, desarrollada por Facebook en 2013. A menudo es considerado un framework debido a su ecosistema y a las herramientas complementarias que se utilizan con él. React es conocido por su enfoque en la construcción de componentes reutilizables y su eficiencia en la actualización del DOM.

Características Principales:

- **Virtual DOM:** React utiliza un DOM virtual para mejorar el rendimiento al minimizar las operaciones costosas en el DOM real.

- **JSX:** Una extensión de JavaScript que permite escribir estructuras HTML dentro de archivos JavaScript, facilitando la creación de componentes.

- **Unidireccionalidad de datos:** Los datos en React fluyen en una sola dirección, lo que facilita la gestión del estado y la depuración.

- **Ecosistema robusto:** Aunque React se centra en la vista, se complementa con bibliotecas como Redux para la gestión del estado y React Router para el enrutamiento.

Cuándo usar React:

- Ideal para aplicaciones donde se requiere alta interactividad y actualización rápida del DOM, como en redes sociales o plataformas de contenido dinámico.

- Adecuado para grandes aplicaciones donde la gestión del estado es crucial.

16.3. Angular

Angular es un framework de desarrollo web completo y robusto, creado por Google. La versión original, AngularJS, fue lanzada en 2010, pero la versión completamente reescrita, Angular (a partir de Angular 2+), fue lanzada en 2016. Angular está diseñado para desarrollar aplicaciones web a gran escala y es una solución completa para crear SPAs.

Características Principales:

- **Arquitectura MVC:** Angular sigue el patrón Modelo-Vista-Controlador, facilitando la separación de preocupaciones en el desarrollo.

- **TypeScript:** Angular se construye sobre TypeScript, un superconjunto de JavaScript que permite escribir código más seguro y escalable.

- **Inyección de dependencias:** Angular proporciona un potente sistema de inyección de dependencias que facilita la creación de aplicaciones modulares y escalables.

- **Ecosistema completo:** Angular ofrece una solución integral con herramientas integradas para enrutamiento, formularios, HTTP, validación, y más.

Cuándo usar Angular:

- Ideal para aplicaciones empresariales y proyectos grandes que requieren una solución completa desde el inicio.

- Cuando se necesita un framework que gestione todas las partes de la aplicación, desde el enrutamiento hasta la gestión de formularios y validación.

16.4. Comparación entre frameworks

a) **Curva de aprendizaje**: Vue.js es generalmente el más fácil de aprender, seguido por React. Angular, con su enfoque en TypeScript y su arquitectura más compleja, tiene una curva de aprendizaje más pronunciada.

b) **Ecosistema:** React y Angular tienen ecosistemas más desarrollados, con una gran cantidad de bibliotecas y herramientas complementarias. Vue.js, aunque más joven, también está creciendo rápidamente en este aspecto.

c) **Flexibilidad vs. Solución completa:** React y Vue.js son más flexibles y pueden integrarse fácilmente en proyectos existentes. Angular es más adecuado cuando se necesita una solución todo en uno.

En resumen, la elección entre Vue.js, React, y Angular depende en gran medida del tipo de proyecto, el equipo de desarrollo y las necesidades específicas de la aplicación. Cada uno tiene su lugar y puede ser la mejor opción dependiendo del contexto.

17. Proyectos y Ejercicios Prácticos

17.1. Calculadora

En este proyecto, vamos a desarrollar una calculadora básica. La calculadora permite realizar operaciones aritméticas simples como suma, resta, multiplicación y división, además de otras funcionalidades como el uso de porcentajes, decimales y la capacidad de borrar la pantalla o eliminar el último dígito ingresado.

Pantalla de visualización: El campo de texto muestra las operaciones y resultados de manera interactiva.

Botones de operaciones y números: Los usuarios pueden ingresar números y operadores como "+", "-", "*", "/" y "%".

Operaciones básicas: La calculadora procesa operaciones aritméticas utilizando la función eval() de JavaScript.
Borrado: Incluye un botón de reinicio total ("C") y uno para eliminar el último carácter ("←").

Aquí tienes un ejemplo de una calculadora simple usando HTML, CSS y JavaScript. Esta calculadora permite realizar operaciones básicas como suma, resta, multiplicación y división.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Calculadora</title>
</head>
<body>

    <div class="calculadora">
        <input type="text" id="resultado" disabled>
        <div class="botones">
            <button onclick="limpiar()">C</button>
            <button onclick="borrar()">←</button>
            <button onclick="agregarSimbolo('%')">%</button>
            <button class="operador"
onclick="agregarSimbolo('/')">/</button>

            <button onclick="agregarSimbolo('7')">7</button>
            <button onclick="agregarSimbolo('8')">8</button>
            <button onclick="agregarSimbolo('9')">9</button>
```

```
            <button class="operador"
onclick="agregarSimbolo('*')">*</button>

            <button onclick="agregarSimbolo('4')">4</button>
            <button onclick="agregarSimbolo('5')">5</button>
            <button onclick="agregarSimbolo('6')">6</button>
            <button class="operador"
onclick="agregarSimbolo('-')">-</button>

            <button onclick="agregarSimbolo('1')">1</button>
            <button onclick="agregarSimbolo('2')">2</button>
            <button onclick="agregarSimbolo('3')">3</button>
            <button class="operador"
onclick="agregarSimbolo('+')">+</button>

            <button onclick="agregarSimbolo('0')" style="grid-column:
span 2;">0</button>
            <button onclick="agregarSimbolo('.')">.</button>
            <button class="operador" onclick="calcular()">=</button>
        </div>
    </div>

    <script>
        // Agrega el símbolo (número u operador) a la pantalla
        function agregarSimbolo(simbolo) {
            document.getElementById("resultado").value += simbolo;
        }

        // Limpia la pantalla de la calculadora
        function limpiar() {
            document.getElementById("resultado").value = "";
        }

        // Elimina el último carácter de la pantalla
        function borrar() {
            let resultado = document.getElementById("resultado").value;
            document.getElementById("resultado").value =
resultado.slice(0, -1);
        }

        // Realiza el cálculo usando la función eval (cuidado con la
seguridad)
        function calcular() {
            let resultado = document.getElementById("resultado").value;
            try {
                document.getElementById("resultado").value =
```

```
eval(resultado);
            } catch (error) {
                document.getElementById("resultado").value = "Error";
            }
        }
    </script>

</body>
</html>
```

- agregarSimbolo(simbolo): Añade el número u operador al campo de texto.
- limpiar(): Limpia toda la pantalla.
- borrar(): Borra el último carácter ingresado.
- calcular(): Evalúa la expresión matemática y muestra el resultado. Utiliza eval(), que es útil para este ejemplo, pero debe usarse con precaución en aplicaciones más grandes por razones de seguridad.

17.2. ToDo List

Desarrolla una aplicación web de "To-Do List" utilizando HTML, CSS y JavaScript. La aplicación debe permitir al usuario agregar nuevas tareas, marcar tareas como completadas y eliminar tareas. La lista de tareas debe mostrarse en la misma página sin recargar.

Requisitos:
1. Agregar Tareas: El usuario debe poder escribir una tarea en un campo de texto y agregarla a la lista haciendo clic en un botón o presionando "Enter".
2. Eliminar Tareas: Cada tarea en la lista debe tener un botón que permita eliminarla.
3. Marcar Tareas como Completadas: Al hacer clic en una tarea, esta debe marcarse como completada, aplicando un estilo de tachado.
4. Interactividad: La aplicación debe actualizar la lista de tareas en tiempo real sin necesidad de recargar la página.

Opcional:
- Guarda las tareas en el localStorage del navegador para que persistan al recargar la página.
- Permite editar las tareas existentes.
- Añade categorías o prioridades a las tareas.

HTML (`index.html`)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

```
    <title>To-Do List</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>To-Do List</h1>
        <input type="text" id="new-task" placeholder="Agregar una nueva
tarea...">
        <button id="add-task">Agregar</button>

        <ul id="task-list">
            <!-- Las tareas se agregarán aquí -->
        </ul>
    </div>
    <script src="app.js"></script>
</body>
</html>
```

CSS (`styles.css`)

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
}

.container {
    background: #fff;
    padding: 20px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    border-radius: 8px;
    width: 300px;
    text-align: center;
}

h1 {
    margin-bottom: 20px;
}

input[type="text"] {
    width: calc(100% - 20px);
```

```
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid #ddd;
    border-radius: 4px;
}

button {
    padding: 10px 20px;
    border: none;
    background-color: #5cb85c;
    color: white;
    border-radius: 4px;
    cursor: pointer;
    margin-bottom: 20px;
}

button:hover {
    background-color: #4cae4c;
}

ul {
    list-style-type: none;
    padding: 0;
}

li {
    background-color: #eee;
    padding: 10px;
    margin-bottom: 5px;
    border-radius: 4px;
    display: flex;
    justify-content: space-between;
    align-items: center;
}

li.completed {
    text-decoration: line-through;
    color: gray;
}

.delete {
    background: #d9534f;
    color: white;
    border: none;
    padding: 5px 10px;
    border-radius: 4px;
```

```
    cursor: pointer;
}

.delete:hover {
    background: #c9302c;
}
```

JavaScript (`app.js`)

```
// Obtener referencias a los elementos del DOM
const taskInput = document.getElementById('new-task');
const addTaskButton = document.getElementById('add-task');
const taskList = document.getElementById('task-list');

// Función para agregar una nueva tarea
function addTask() {
    const taskText = taskInput.value.trim();

    if (taskText === '') {
        alert('Por favor, ingresa una tarea.');
        return;
    }

    const li = document.createElement('li');
    li.textContent = taskText;

    const deleteButton = document.createElement('button');
    deleteButton.textContent = 'Eliminar';
    deleteButton.classList.add('delete');
    deleteButton.onclick = function() {
        taskList.removeChild(li);
    };

    li.appendChild(deleteButton);

    li.onclick = function() {
        li.classList.toggle('completed');
    };

    taskList.appendChild(li);
    taskInput.value = '';
}

// Evento para agregar tarea cuando se hace clic en el botón
addTaskButton.addEventListener('click', addTask);
```

```
// Evento para agregar tarea al presionar Enter
taskInput.addEventListener('keypress', function(e) {
    if (e.key === 'Enter') {
        addTask();
    }
});
```

17.3. Juego simple (Piedra, Papel o Tijera)

Desarrolla una aplicación web simple que permita jugar "Piedra, Papel o Tijera" contra la computadora. El usuario debe poder seleccionar su opción (piedra, papel o tijera) y la computadora debe generar una elección aleatoria. Después de cada ronda, se debe mostrar el resultado (victoria, derrota o empate) y actualizar el marcador.

Requisitos:

a) Selección del Jugador: El usuario debe poder elegir entre piedra, papel o tijera haciendo clic en botones correspondientes.

b) Selección Aleatoria de la Computadora: La computadora debe elegir aleatoriamente entre piedra, papel o tijera.

c) Determinación del Ganador: La aplicación debe comparar las elecciones y determinar el ganador según las reglas:
   - Piedra vence a Tijera
   - Tijera vence a Papel
   - Papel vence a Piedra

d) Mostrar Resultado: Después de cada ronda, se debe mostrar un mensaje indicando si el jugador ganó, perdió o empató.

e) Actualizar Marcador: Lleva un conteo de cuántas veces ha ganado el jugador, la computadora y cuántos empates ha habido.

Opcional:

- Agrega animaciones o efectos visuales al seleccionar una opción o mostrar el resultado.

- Implementa un sistema de rondas, por ejemplo, el primero en ganar 5 veces.

- Guarda el historial de resultados en localStorage para que persistan al recargar la página.

HTML (index.html)

```
<!DOCTYPE html>
```

```
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Piedra, Papel o Tijera</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="container">
        <h1>Piedra, Papel o Tijera</h1>
        <div class="choices">
            <button class="choice" data-choice="piedra">✊
Piedra</button>
            <button class="choice" data-choice="papel">✋ Papel</button>
            <button class="choice" data-choice="tijera">✌
Tijera</button>
        </div>

        <div id="result">
            <p id="player-choice">Tu elección: <span>-</span></p>
            <p id="computer-choice">Elección de la computadora:
<span>-</span></p>
            <h2 id="outcome">Resultado: -</h2>
        </div>

        <div id="score">
            <p>Jugador: <span id="player-score">0</span></p>
            <p>Computadora: <span id="computer-score">0</span></p>
            <p>Empates: <span id="ties">0</span></p>
        </div>
    </div>
    <script src="app.js"></script>
</body>
</html>
```

CSS (`styles.css`)

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
```

```
    margin: 0;
}

.container {
    background: #fff;
    padding: 20px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    border-radius: 8px;
    text-align: center;
    width: 300px;
}

h1 {
    margin-bottom: 20px;
}

.choices {
    margin-bottom: 20px;
}

.choice {
    padding: 10px 20px;
    margin: 0 5px;
    border: none;
    background-color: #007bff;
    color: white;
    border-radius: 4px;
    cursor: pointer;
}

.choice:hover {
    background-color: #0056b3;
}

#result {
    margin-bottom: 20px;
}

#outcome {
    margin: 20px 0;
    color: #007bff;
}

#score {
    margin-top: 20px;
    border-top: 1px solid #ddd;
```

```
    padding-top: 10px;
}
```

JavaScript (`app.js`)

```
// Referencias a elementos del DOM
const choices = document.querySelectorAll('.choice');
const playerChoiceText =
document.getElementById('player-choice').querySelector('span');
const computerChoiceText =
document.getElementById('computer-choice').querySelector('span');
const outcomeText = document.getElementById('outcome');
const playerScoreText = document.getElementById('player-score');
const computerScoreText = document.getElementById('computer-score');
const tiesText = document.getElementById('ties');

let playerScore = 0;
let computerScore = 0;
let ties = 0;

// Función para obtener la elección aleatoria de la computadora
function getComputerChoice() {
    const choices = ['piedra', 'papel', 'tijera'];
    const randomIndex = Math.floor(Math.random() * choices.length);
    return choices[randomIndex];
}

// Función para determinar el ganador
function determineWinner(playerChoice, computerChoice) {
    if (playerChoice === computerChoice) {
        ties++;
        return "Empate";
    }

    if (
        (playerChoice === 'piedra' && computerChoice === 'tijera') ||
        (playerChoice === 'tijera' && computerChoice === 'papel') ||
        (playerChoice === 'papel' && computerChoice === 'piedra')
    ) {
        playerScore++;
        return "¡Ganaste!";
    } else {
        computerScore++;
        return "Perdiste...";
```

```
    }
}

// Función principal que se ejecuta cuando el jugador hace una elección
function playRound(playerChoice) {
    const computerChoice = getComputerChoice();

    playerChoiceText.textContent = playerChoice;
    computerChoiceText.textContent = computerChoice;

    const outcome = determineWinner(playerChoice, computerChoice);
    outcomeText.textContent = `Resultado: ${outcome}`;

    playerScoreText.textContent = playerScore;
    computerScoreText.textContent = computerScore;
    tiesText.textContent = ties;
}

// Añadir eventos a los botones
choices.forEach(choice => {
    choice.addEventListener('click', () => {
        playRound(choice.getAttribute('data-choice'));
    });
});
```

- Selección de Jugador: El usuario selecciona su opción (piedra, papel o tijera) haciendo clic en uno de los botones.
- Selección de la Computadora: La computadora elige aleatoriamente entre piedra, papel o tijera.
- Resultado: La aplicación determina quién gana y muestra el resultado en la pantalla junto con la actualización del marcador.

18. Proyectos Intermedios

18.1. Aplicación de notas

Vamos a crear una aplicación de notas en javascript donde aparezca una tabla con el nombre, apellidos y tres notas obtenidas así como la media que debe calcularse. Además tendremos los botones de alta, baja y modificación de notas.

Entiendo, eliminaremos el uso de `prompt` y haremos que tanto la adición como la modificación de notas se realicen a través del formulario directamente en la página. También ajustaremos el flujo para que el formulario pueda usarse tanto para agregar como para modificar las notas.

a) HTML

```
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Aplicación de Notas</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <h1>Aplicación de Notas</h1>

    <form id="notasForm">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" required>
        <br><br>
        <label for="apellidos">Apellidos:</label>
        <input type="text" id="apellidos" name="apellidos" required>
        <br><br>
        <label for="nota1">Nota 1:</label>
        <input type="number" id="nota1" name="nota1" min="0" max="10"
step="0.01" required>
        <br><br>
        <label for="nota2">Nota 2:</label>
        <input type="number" id="nota2" name="nota2" min="0" max="10"
step="0.01" required>
        <br><br>
        <label for="nota3">Nota 3:</label>
        <input type="number" id="nota3" name="nota3" min="0" max="10"
step="0.01" required>
        <br><br>
        <button type="button" onclick="agregarOModificarNota()">Guardar
Nota</button>
    </form>

    <br>

    <table id="notasTable">
        <thead>
            <tr>
                <th>Nombre</th>
                <th>Apellidos</th>
                <th>Nota 1</th>
                <th>Nota 2</th>
                <th>Nota 3</th>
```

```
                <th>Media</th>
                <th>Acciones</th>
            </tr>
        </thead>
        <tbody id="notasBody">
            <!-- Las filas de datos se agregarán aquí dinámicamente -->
        </tbody>
    </table>

    <script src="app.js"></script>
</body>
</html>
```

b) Javascript

```
let notas = [];
let editIndex = -1; // Variable para identificar si se está editando una
nota

function agregarOModificarNota() {
    const nombre = document.getElementById("nombre").value;
    const apellidos = document.getElementById("apellidos").value;
    const nota1 = parseFloat(document.getElementById("nota1").value);
    const nota2 = parseFloat(document.getElementById("nota2").value);
    const nota3 = parseFloat(document.getElementById("nota3").value);

    if (!isNaN(nota1) && !isNaN(nota2) && !isNaN(nota3)) {
        const media = calcularMedia(nota1, nota2, nota3);
        const nuevaNota = { nombre, apellidos, nota1, nota2, nota3,
media };

        if (editIndex === -1) {
            // Si no estamos editando, agregamos una nueva nota
            notas.push(nuevaNota);
        } else {
            // Si estamos editando, actualizamos la nota existente
            notas[editIndex] = nuevaNota;
            editIndex = -1; // Reiniciamos el índice de edición
        }

        renderizarTabla();
        document.getElementById("notasForm").reset(); // Limpiar el
formulario
    } else {
        alert("Por favor, ingrese valores numéricos válidos para las
```

```
notas.");
    }
}

function eliminarNota(index) {
    notas.splice(index, 1);
    renderizarTabla();
}

function modificarNota(index) {
    const nota = notas[index];
    document.getElementById("nombre").value = nota.nombre;
    document.getElementById("apellidos").value = nota.apellidos;
    document.getElementById("nota1").value = nota.nota1;
    document.getElementById("nota2").value = nota.nota2;
    document.getElementById("nota3").value = nota.nota3;

    editIndex = index; // Guardamos el índice de la nota que se está
editando
}

function calcularMedia(nota1, nota2, nota3) {
    return ((nota1 + nota2 + nota3) / 3).toFixed(2);
}

function renderizarTabla() {
    const notasBody = document.getElementById("notasBody");
    notasBody.innerHTML = "";

    notas.forEach((nota, index) => {
        const fila = document.createElement("tr");

        fila.innerHTML = `
            <td>${nota.nombre}</td>
            <td>${nota.apellidos}</td>
            <td>${nota.nota1}</td>
            <td>${nota.nota2}</td>
            <td>${nota.nota3}</td>
            <td>${nota.media}</td>
            <td>
                <button
onclick="modificarNota(${index})">Modificar</button>
                <button
onclick="eliminarNota(${index})">Eliminar</button>
            </td>
        `;
```

```
        notasBody.appendChild(fila);
    });
}
```

Formulario para Agregar y Modificar: El formulario permite tanto agregar como modificar notas. La función agregarOModificarNota maneja ambos casos: si estamos agregando una nueva nota o si estamos editando una existente.

Variable editIndex: Se utiliza editIndex para almacenar el índice de la nota que se está editando. Si editIndex es -1, se agrega una nueva nota; si no, se actualiza la nota existente.
Función modificarNota(index): Cuando se hace clic en el botón "Modificar", los datos de la nota seleccionada se cargan en el formulario. El formulario puede ser editado y, al hacer clic en "Guardar Nota", los cambios se guardan en la tabla.

18.2. Consumo de una API REST

El consumo de una API en JavaScript se refiere a la interacción con una API (Interfaz de Programación de Aplicaciones) desde una aplicación web o una página web usando JavaScript. Este proceso generalmente involucra hacer peticiones a un servidor para obtener datos, enviarlos o realizar otras acciones según lo que permita la API.

Para realizar este proceso debemos tener en cuenta los siguientes pasos:

a) **Seleccionar la API y entender su documentación:** Antes de consumir una API, es esencial comprender cómo funciona, qué endpoints ofrece, qué tipo de datos se envían y reciben, y qué métodos HTTP (como GET, POST, PUT, DELETE, etc.) se deben usar.

b) **Realizar una petición a la API:** Para interactuar con una API, se realiza una solicitud HTTP a uno de sus endpoints. JavaScript tiene varias formas de hacer esto, siendo las más comunes `XMLHttpRequest`, `fetch` y bibliotecas como `Axios`.

c) **Manejar la respuesta:** Una vez que se envía una solicitud, la API devuelve una respuesta que generalmente incluye un código de estado (como 200 para éxito, 404 para no encontrado, etc.) y los datos en el formato especificado (usualmente JSON).

d) **Procesar los datos recibidos:** Después de recibir la respuesta, es necesario procesar los datos para usarlos en la aplicación, como mostrarlos en la interfaz de usuario o usarlos en alguna lógica interna.

Veamos un ejemplo con la API falsa de JSONPlaceHolder. Supongamos que queremos obtener una lista de publicaciones (posts) desde esta API y mostrarlas en una capa DIV.

Claro, aquí tienes un ejemplo de cómo consumir la API de JSONPlaceholder y mostrar cada publicación en un `<div>` dentro de tu HTML en lugar de la consola.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Ejemplo API JSONPlaceholder</title>
    <style>
        /* Estilos básicos para las publicaciones */
        .post {
            border: 1px solid #ccc;
            padding: 10px;
            margin: 10px 0;
            border-radius: 5px;
        }
    </style>
</head>
<body>
    <h1>Lista de Publicaciones</h1>
    <div id="posts-container"></div>

    <script src="app.js"></script>
</body>
</html>
```

```javascript
// URL del endpoint de la API para obtener las publicaciones
const apiURL = 'https://jsonplaceholder.typicode.com/posts';

// Seleccionamos el contenedor donde mostraremos las publicaciones
const postsContainer = document.getElementById('posts-container');

// Hacer una petición GET a la API
fetch(apiURL)
  .then(response => {
    // Verificar si la respuesta fue exitosa (código 200)
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    // Convertir la respuesta en un objeto JSON
    return response.json();
  })
  .then(data => {
    // Iterar sobre cada publicación y crear un div para mostrarla
    data.forEach(post => {
      // Crear un div para cada publicación
```

```
      const postDiv = document.createElement('div');
      postDiv.classList.add('post');

      // Añadir el contenido de la publicación al div
      postDiv.innerHTML = `
        <h2>${post.title}</h2>
        <p>${post.body}</p>
        <small>ID: ${post.id}</small>
      `;

      // Añadir el div al contenedor en el HTML
      postsContainer.appendChild(postDiv);
    });
  })
  .catch(error => {
    // Manejo de errores en caso de fallo en la petición
    console.error('There has been a problem with your fetch operation:',
error);
  });
```

- HTML Contenedor: El elemento `<div id="posts-container"></div>` en el HTML es donde se insertarán las publicaciones dinámicamente.

- JavaScript: Se selecciona el contenedor donde se van a añadir los `<div>` de las publicaciones.
- fetch(apiURL): Realiza la petición a la API.
- data.forEach(post => {...})`: Itera sobre cada publicación recibida. Por cada publicación, se crea un nuevo ` ` con la clase `post`, que contiene el título (`<h2>`), el cuerpo (`<p>`) y el ID (`<small>`).
- postsContainer.appendChild(postDiv): Añade cada `<div>` al contenedor en el HTML.

- Estilos (Opcional): El CSS en el `<style>` proporciona un diseño básico para las publicaciones, con bordes, margen y padding.

Cuando cargues la página en un navegador, debería obtener las publicaciones de la API y mostrarlas como una lista de tarjetas dentro de tu contenedor.

18.3. Aplicación de chat simple

19. Introducción e instalación de Vue.js

19.1. ¿Qué es Vue.js?

Vue.js es un framework de JavaScript progresivo diseñado para construir interfaces de usuario interactivas y aplicaciones web de una sola página (SPA). Fue creado por Evan You y lanzado por primera vez en 2014. Vue se distingue por su enfoque en la simplicidad y la facilidad de integración, permitiendo a los desarrolladores crear aplicaciones de manera rápida y eficiente.

Características:

- **Reactividad:** Vue.js utiliza un sistema de reactividad que actualiza automáticamente la interfaz de usuario cuando los datos cambian. Esto significa que los desarrolladores pueden concentrarse en los datos y las funcionalidades sin preocuparse por la sincronización manual entre los datos y la vista.

- **Componentes:** Vue.js permite construir interfaces usando componentes reutilizables. Los componentes encapsulan tanto la lógica como el HTML y CSS, facilitando la organización y el mantenimiento del código.

- **Directivas:** Vue usa directivas especiales en la plantilla para aplicar comportamientos reactivos tales como v-bind, v-model, v-for, v-if y v-on.

- **Facilidad de Integración:** Vue puede ser añadido a proyectos existentes de forma gradual, lo que permite integrar funcionalidades de Vue en aplicaciones existentes sin necesidad de una reescritura completa.

19.2. Historia y Filosofía

Vue.js fue creado por Evan You, quien anteriormente había trabajado en Google y participado en el desarrollo de AngularJS. Inspirado en la simplicidad y la elegancia de AngularJS, You buscó construir un framework que combinara lo mejor de los dos mundos: la facilidad de uso de AngularJS y la flexibilidad de otros enfoques.

- 2014: Lanzamiento inicial de Vue.js.
- 2016: Vue.js 2.0 se publica, trayendo mejoras significativas en la reactividad y el rendimiento.
- 2018: Vue.js 2.x se convierte en una de las bibliotecas más populares para el desarrollo frontend.
- 2020: Vue.js 3.0 es lanzado, con mejoras en el rendimiento, un nuevo sistema de composición y soporte para TypeScript.

La filosofía de Vue.js se centra en ofrecer una experiencia de desarrollo que sea tan potente como accesible. El objetivo es proporcionar una herramienta que permita a los desarrolladores construir aplicaciones complejas sin que la complejidad del propio framework sea una barrera.

19.3. Características Clave

Vue.js se distingue por varias características clave que facilitan el desarrollo y la gestión de aplicaciones web:

- **Sistema de Componentes:** Vue.js promueve una arquitectura basada en componentes, lo que facilita la creación de aplicaciones modulares y reutilizables. Cada componente encapsula su propia lógica y estilo, promoviendo una estructura más organizada y mantenible.

- **Reactividad y Data Binding:** Vue utiliza un sistema de reactividad que actualiza automáticamente el DOM cuando los datos cambian. Esto se logra mediante un sistema de "data binding" bidireccional que mantiene sincronizados los datos y la vista.

- **Directivas Declarativas:** Vue usa directivas para manipular el DOM de manera declarativa. Las directivas son atributos especiales en las plantillas que proporcionan funcionalidades reactivas, como el enlace de datos y la iteración sobre listas.

- **Sistema de Composición:** Con Vue 3, el sistema de composición permite organizar la lógica de los componentes de una manera más flexible y reutilizable mediante el uso de "composables" y la Composition API.

- **Integración Gradual:** Vue.js se puede incorporar de manera incremental en proyectos existentes. Esto permite a los desarrolladores usar Vue solo para partes específicas de una aplicación sin necesidad de reescribir todo el código.

- **Documentación Extensa y Comunidad Activa:** Vue cuenta con una documentación detallada y una comunidad activa que contribuye con plugins, extensiones y soporte. Esto facilita el aprendizaje y la resolución de problemas.

19.4. Instalación y Configuración

¡Por supuesto! A continuación se desarrolla el punto 2 del manual sobre Vue.js, que cubre la instalación y configuración del entorno para trabajar con Vue.js.

a) Usando CDN

Para comenzar rápidamente con Vue.js, una forma sencilla es incluirlo directamente en tu proyecto mediante un CDN (Content Delivery Network). Esta opción es ideal para prototipos o pruebas rápidas.

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

```
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <title>Mi Aplicación Vue</title>
        <script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
    </head>
    <body>
        <div id="app">
            {{ mensaje }}
        </div>
        <script>
            new Vue({
                el: '#app',
                data: {
                    mensaje: '¡Hola Vue!'
                }
            });
        </script>
    </body>
    </html>
```

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>: Incluye Vue.js desde un CDN. En este caso, se está cargando Vue 2.x.

new Vue({ ... }): Crea una nueva instancia de Vue que controla el elemento con el ID app. En este ejemplo, se define un objeto data con una propiedad `mensaje` que se muestra en la vista.

b) Instalación con npm

Para proyectos más complejos, se recomienda utilizar un gestor de paquetes como npm para instalar Vue.js. Esto permite una gestión más avanzada de dependencias y es adecuado para aplicaciones grandes.

Abre una terminal y crea una nueva carpeta para tu proyecto, luego inicializa un proyecto npm:

```
mkdir mi-proyecto-vue
cd mi-proyecto-vue
npm init -y
```

Ejecuta el siguiente comando para instalar Vue.js:

```
npm install vue
```

Crea una estructura básica de archivos:

- index.html
- main.js

index.html:

```
<!DOCTYPE html>
   <html lang="es">
   <head>
       <meta charset="UTF-8">
       <meta name="viewport" content="width=device-width,
initial-scale=1.0">
       <title>Mi Aplicación Vue</title>
   </head>
   <body>
       <div id="app">
           {{ mensaje }}
       </div>
       <script src="./main.js"></script>
   </body>
   </html>
```

main.js:

```
   import Vue from 'vue';

   new Vue({
       el: '#app',
       data: {
           mensaje: '¡Hola Vue!'
       }
   });
```

Servir el Proyecto:

Utiliza un servidor de desarrollo para servir el proyecto. Puedes usar herramientas como http-server o configurar un entorno con Webpack o Vite para desarrollo.

```
npx http-server
```

Abre tu navegador en `http://localhost:8080` para ver tu aplicación en funcionamiento.

c) Configuración de un Proyecto Vue con Vue CLI

Vue CLI es una herramienta que proporciona una configuración inicial para proyectos Vue, incluyendo herramientas para desarrollo, pruebas y construcción. Es ideal para configurar proyectos Vue de manera rápida y profesional.

1. Instalar Vue CLI:

```
npm install -g @vue/cli
```

2. Crear un Nuevo Proyecto:

```
vue create mi-proyecto-vue
```

Sigue las instrucciones en pantalla para seleccionar las configuraciones deseadas (como Babel, ESLint, etc.).

3. Navegar a la Carpeta del Proyecto y Ejecutar el Servidor de Desarrollo:

```
cd mi-proyecto-vue
npm run serve
```

Esto iniciará un servidor de desarrollo y podrás ver tu aplicación en `http://localhost:8080`.

2.4. Estructura de Archivos en un Proyecto Vue

Cuando usas Vue CLI, tu proyecto tendrá una estructura de archivos organizada:

- public/: Contiene el archivo `index.html` y otros activos públicos.
- src/: Contiene el código fuente de tu aplicación:
  - main.js:: El archivo principal que crea la instancia Vue y monta la aplicación.
  - App.vue: Componente raíz de la aplicación.
  - components: Carpeta para componentes Vue individuales.
- node_modules: Carpeta que contiene las dependencias instaladas.

```
mi-proyecto-vue/
│
├── public/
│   └── index.html
│
├── src/
│   ├── main.js
│   ├── App.vue
│   └── components/
│       └── HelloWorld.vue
│
├── package.json
└── vue.config.js
```

20. Conceptos Básicos en Vue

20.2. Creación de la Instancia Vue

En Vue.js, todo comienza con la creación de una instancia de Vue. Esta instancia es el núcleo de cualquier aplicación Vue y es responsable de controlar una parte del DOM y manejar los datos y la lógica de la aplicación.

```
new Vue({
    el: '#app',
    data: {
        mensaje: '¡Hola Vue!'
    }
});
```

- **el:** Especifica el elemento del DOM que será controlado por Vue. En este caso, Vue tomará el control del elemento con el ID `app`.

- **data:** Contiene las propiedades que Vue usará para la reactividad. Aquí se define la propiedad `mensaje` que puede ser usada en la vista.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Mi Aplicación Vue</title>
</head>
<body>
    <div id="app">
        {{ mensaje }}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
    <script src="./main.js"></script>
</body>
</html>
```

Es importante entender la idea de que **“el”** hace referencia al id de la etiqueta donde se desplegará nuestra aplicación u objeto en Vue, en este caso **app**. Fuera de esta etiqueta el objeto no tendrá ninguna relevancia.

20.3. Data y Methods

En Vue, **data y methods** son dos propiedades fundamentales para gestionar el estado y la lógica de los componentes.

- **data:** Es un objeto que contiene las propiedades reactivas del componente. Estas propiedades están vinculadas a la vista y actualizan automáticamente el DOM cuando cambian.

- **methods:** Es un objeto que contiene las funciones que se pueden invocar desde el DOM o desde otros métodos dentro del componente.

```
new Vue({
    el: '#app',
    data: {
        mensaje: '¡Hola Vue!',
        contador: 0
    },
    methods: {
        incrementar() {
            this.contador++;
        }
    }
});
```

```
<div id="app">
    <p>{{ mensaje }}</p>
    <button v-on:click="incrementar">Incrementar</button>
    <p>Contador: {{ contador }}</p>
</div>
```

incrementar es un método definido en methods que incrementa la propiedad contador cuando se hace clic en el botón. La directiva **v-on:click** se utiliza para enlazar el método con el evento click.

Es interesante observar cómo VUe utiliza la **interpolación** para escribir los datos en HTML mediante la cadena **{{ mensaje }}** donde la variable mensaje perteneciente al objeto se despliega mostrando su valor.

20.4. Directivas Básicas

En Vue.js, las directivas son atributos especiales en las plantillas que se utilizan para enlazar el DOM con el modelo de datos. Son una forma declarativa de decirle a Vue cómo debe comportarse un elemento del DOM en función de los datos de la aplicación. Las directivas proporcionan una forma de aplicar comportamiento y lógica a los elementos HTML de manera reactiva y sencilla.

Las directivas en Vue.js se representan con un prefijo v-, que indica que son directivas específicas de Vue. Estas directivas pueden modificar el DOM, enlazar datos, manejar eventos y controlar la renderización de elementos. Las directivas se aplican integradas en el código HTML.

a) **v-bind:** Se utiliza para enlazar atributos del DOM a datos en la instancia Vue. Permite la interpolación de datos en atributos HTML y la actualización automática de estos atributos cuando los datos cambian.

```
<img v-bind:src="imagenUrl" alt="Descripción">
```

También podemos usar su abreviatura de dos puntos:

```
    <img :src="imagenUrl" alt="Descripción">
```

```
new Vue({
    el: '#app',
    data: {
        imagenUrl: 'https://example.com/imagen.jpg'
    }
});
```

De esta forma tan sencilla conseguimos modificar el valor de un atributo conectándolo a una variable. Pensemos por ejemplo en una etiqueta **<img src=XXX>** donde XXX estaría vinculado a una ruta de una imagen. Simplemente cambiando el valor de esta variable haríamos que la imagen aparezca en nuestro navegador.

b) **v-model:** Utilizado para crear un enlace bidireccional entre el valor de un campo de formulario y una propiedad de datos. Es comúnmente usado en formularios para sincronizar el valor del campo con el estado de los datos.

```
<input v-model="texto">
<p>Texto ingresado: {{ texto }}</p>
```

```
new Vue({
    el: '#app',
    data: {
        texto: ''
    }
});
```

Esta directiva nos permite desentendernos, por ejemplo, del value de un campo de texto y hacer que fuera actualizándose a medida que la variable a la que está vinculado el campo

fuera modificada. Es decir, en Vue prestamos atención al cambio de valor de una variable del objeto.

c) **v-for:** Utilizado para iterar sobre una lista de elementos y renderizar un bloque para cada elemento. Es comúnmente usado para generar listas o tablas dinámicas.

```
<ul>
    <li v-for="item in items" :key="item">{{ item }}</li>
</ul>
```

```
new Vue({
    el: '#app',
    data: {
        items: ['Manzana', 'Banana', 'Cereza']
    }
});
```

d) **v-if y v-else:** Renderizan bloques de código condicionalmente. Es decir, si se cumple una condición el elemento que se encuentre dentro de esta directiva se hará visible. En caso contrario estará oculto. Esto nos proporciona una forma sencilla de controlar los elementos HTML que aparecerán en nuestro proyecto.

```
<p v-if="mostrar">Este texto se muestra solo si 'mostrar' es
verdadero.</p>
<p v-else>Este texto se muestra si 'mostrar' es falso.</p>
```

```
new Vue({
    el: '#app',
    data: {
        mostrar: true
    }
});
```

e) **v-on:** en Vue.js se utiliza para agregar controladores de eventos a los elementos HTML. Es una de las directivas más comunes y poderosas de Vue.js, ya que te permite manejar eventos como clics, cambios de input, movimientos del ratón, y más, directamente en tu plantilla.

```
<button v-on:click="metodo">Haz clic aquí</button>
```

También disponemos de su abreviatura:

```
<button @click="metodo">Haz clic aquí</button>
```

A continuación veamos un ejemplo que combina algunas de estas directivas para crear la clásica aplicación de lista de tareas o Todo-List

```
<div id="app">
  <h1>Lista de Tareas</h1>

  <!-- Input para agregar una nueva tarea -->
  <input
    v-model="nuevaTarea"
    @keyup.enter="agregarTarea"
    placeholder="Agregar nueva tarea"
  >

  <!-- Botón para agregar la tarea -->
  <button @click="agregarTarea">Agregar</button>

  <!-- Lista de tareas -->
  <ul>
    <li v-for="(tarea, index) in tareas" :key="index">
      <!-- Mostrar la tarea -->
      <span :class="{ completada: tarea.completada }">{{ tarea.texto
}}</span>

      <!-- Botón para marcar como completada -->
      <button @click="marcarComoCompletada(index)">
        {{ tarea.completada ? "Desmarcar" : "Completar" }}
      </button>

      <!-- Botón para eliminar la tarea -->
      <button @click="eliminarTarea(index)">Eliminar</button>
    </li>
  </ul>

  <!-- Mensaje si no hay tareas -->
  <p v-if="tareas.length === 0">No tienes tareas pendientes.</p>
</div>

<script>
  new Vue({
    el: '#app',
    data: {
      nuevaTarea: '',
      tareas: []
    },
    methods: {
```

```
      agregarTarea() {
        if (this.nuevaTarea.trim() !== '') {
          this.tareas.push({ texto: this.nuevaTarea, completada: false
});
          this.nuevaTarea = '';
        }
      },
      marcarComoCompletada(index) {
        this.tareas[index].completada = !this.tareas[index].completada;
      },
      eliminarTarea(index) {
        this.tareas.splice(index, 1);
      }
    }
  });
</script>

<style>
  .completada {
    text-decoration: line-through;
    color: gray;
  }
</style>
```

Veamos este código con mas detenimiento:

1. v-model: Se utiliza en el input de texto para enlazar el valor del input con la variable nuevaTarea. Esto permite que el valor del input se actualice automáticamente en el estado de la aplicación.

2. v-for: Se utiliza para iterar sobre la lista de tareas y renderizar cada una de ellas como un elemento de lista (<li>).

3. v-bind (abreviado como `:`): Se utiliza para enlazar la clase CSS completada dinámicamente al que muestra el texto de la tarea, dependiendo de si la tarea está completada o no.

4. v-on (abreviado como @): Se usa en varios lugares para manejar eventos, como

- @keyup.enter en el input para agregar la tarea cuando se presiona la tecla Enter.
- @click en los botones para agregar, completar o eliminar tareas.

5. v-if: Se utiliza para mostrar un mensaje de "No tienes tareas pendientes" solo si la lista de tareas está vacía.

Veamos a continuación otro ejemplo. Imaginemos que tenemos una lista de alumnos con sus notas. Queremos representar estos alumnos es una tabla y dispondremos de un formulario para insertar, modificar y borrar alumnos. los datos a contemplar serían los siguientes: nombre, apellidos, telefono, email y nota.

```
<div id="app">
  <h1>Gestión de Alumnos</h1>

  <!-- Formulario para agregar o modificar un alumno -->
  <form @submit.prevent="submitForm">
    <div>
      <label for="nombre">Nombre:</label>
      <input type="text" v-model="newAlumno.nombre" required>
    </div>
    <div>
      <label for="apellidos">Apellidos:</label>
      <input type="text" v-model="newAlumno.apellidos" required>
    </div>
    <div>
      <label for="telefono">Teléfono:</label>
      <input type="tel" v-model="newAlumno.telefono" required>
    </div>
    <div>
      <label for="email">Email:</label>
      <input type="email" v-model="newAlumno.email" required>
    </div>
    <div>
      <label for="nota">Nota:</label>
      <input type="number" v-model="newAlumno.nota" min="0" max="10"
required>
    </div>
    <button type="submit">{{ editMode ? "Modificar Alumno" : "Agregar
Alumno" }}</button>
  </form>

  <!-- Tabla de alumnos -->
  <table border="1">
    <thead>
      <tr>
        <th>Nombre</th>
        <th>Apellidos</th>
        <th>Teléfono</th>
        <th>Email</th>
        <th>Nota</th>
        <th>Acciones</th>
      </tr>
```

```
    </thead>
    <tbody>
      <tr v-for="(alumno, index) in alumnos" :key="index">
        <td>{{ alumno.nombre }}</td>
        <td>{{ alumno.apellidos }}</td>
        <td>{{ alumno.telefono }}</td>
        <td>{{ alumno.email }}</td>
        <td>{{ alumno.nota }}</td>
        <td>
          <button @click="editAlumno(index)">Editar</button>
          <button @click="deleteAlumno(index)">Eliminar</button>
        </td>
      </tr>
    </tbody>
  </table>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
new Vue({
  el: '#app',
  data: {
    alumnos: [],
    newAlumno: {
      nombre: '',
      apellidos: '',
      telefono: '',
      email: '',
      nota: null,
    },
    editIndex: -1,
    editMode: false,
  },
  methods: {
    submitForm() {
      if (this.editMode) {
        // Si estamos en modo edición, actualizamos el alumno
        this.alumnos.splice(this.editIndex, 1, this.newAlumno);
        this.editMode = false;
      } else {
        // Si no, agregamos el nuevo alumno
        this.alumnos.push(this.newAlumno);
      }
      // Limpiamos el formulario
      this.resetForm();
    },
```

```
    editAlumno(index) {
      this.newAlumno = Object.assign({}, this.alumnos[index]);
      this.editIndex = index;
      this.editMode = true;
    },
    deleteAlumno(index) {
      this.alumnos.splice(index, 1);
    },
    resetForm() {
      this.newAlumno = {
        nombre: '',
        apellidos: '',
        telefono: '',
        email: '',
        nota: null,
      };
      this.editIndex = -1;
    }
  }
});
</script>
```

El formulario de alumnos permite agregar o modificar alumnos con campos para nombre, apellidos, teléfono, email y nota. El botón cambia de texto si se está editando o agregando un nuevo alumno.

La tabla de alumnos muestra la lista de alumnos con todas sus propiedades y permite realizar acciones de editar o eliminar.

Los modos de edición y adición: Si un alumno es editado, el formulario se rellena con sus datos. Si se hace clic en "Modificar Alumno", se actualizan los datos, y si no, se añade un nuevo alumno.

Las operaciones básicas: Uso de métodos para añadir, editar y eliminar alumnos de la lista.

20.5. Computed Properties y Watchers. Mounted.

a) **Computed Properties:** Son propiedades calculadas que dependen de otras propiedades reactivas. Se actualizan automáticamente cuando cambian las propiedades de las que dependen.

```
new Vue({
    el: '#app',
    data: {
        numero: 10
```

```
    },
    computed: {
        cuadrado() {
            return this.numero * this.numero;
        }
    }
});
```

```
<p>Numero: {{ numero }}</p>
<p>Cuadrado: {{ cuadrado }}</p>
```

b) **Watchers:** Son funciones que observan los cambios en las propiedades y ejecutan código adicional cuando cambian.

```
new Vue({
    el: '#app',
    data: {
        numero: 10
    },
    watch: {
        numero(nuevoValor, valorAnterior) {
            console.log(`Número cambiado de ${valorAnterior} a
${nuevoValor}`);
        }
    }
});
```

```
<p>Numero: {{ numero }}</p>
```

c) **Mounted:** En Vue.js, el ciclo de vida de una instancia Vue comprende varias etapas, desde su creación hasta su destrucción. Una de las etapas más importantes es cuando la instancia Vue se monta en el DOM (Document Object Model). Aquí es donde entra en juego el hook mounted`**.

   mounted se utiliza para ejecutar código que necesita que la instancia Vue esté completamente montada en el DOM. Normalmente se ejecuta al comienzo de nuestra aplicación y nos servirá para inicializar datos del objeto Vue.

Veamos a continuación un ejemplo donde ponemos en práctica lo aprendido hasta ahora mediante una aplicación que nos sirva para adivinar un número entre 1 y 100 elegido aleatoriamente por nuestra aplicación.

```
<!DOCTYPE html>
```

```
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Juego de Adivinanza</title>
</head>
<body>

<div id="app" class="container">
    <h1>¡Adivina el Número!</h1>
    <input type="number" v-model="intento" placeholder="Tu número">
    <button @click="adivinar">Adivinar</button>
    <p v-if="mensaje">{{ mensaje }}</p>
    <p v-if="ganado">¡Felicidades! Adivinaste el número en {{ intentos
}} intentos.</p>
    <button v-if="ganado" @click="reiniciar">Jugar de nuevo</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
new Vue({
    el: '#app',
    data: {
        numeroSecreto: null,
        intento: null,
        mensaje: '',
        intentos: 0,
        ganado: false
    },
    mounted() {
        this.reiniciar();
    },
    methods: {
        reiniciar() {
            this.numeroSecreto = Math.floor(Math.random() * 100) + 1;
            this.intento = null;
            this.mensaje = '';
            this.intentos = 0;
            this.ganado = false;
        },
        adivinar() {
            if (this.intento === null || this.intento === '') {
                this.mensaje = 'Por favor, ingresa un número.';
                return;
            }
```

```
            this.intentos++;
            const numero = parseInt(this.intento);

            if (numero === this.numeroSecreto) {
                this.mensaje = '';
                this.ganado = true;
            } else if (numero < this.numeroSecreto) {
                this.mensaje = 'El número es mayor.';
            } else {
                this.mensaje = 'El número es menor.';
            }
        }
    }
});
</script>

</body>
</html>
```

La aplicación tiene una estructura simple con un input para que el usuario introduzca su número, un botón para enviar el intento, y un mensaje que se muestra dependiendo de la respuesta. Si el usuario adivina el número, se muestra un mensaje de felicitación y un botón para reiniciar el juego.

En data se definen las variables principales:

- numeroSecreto: Almacena el número aleatorio generado por el ordenador.
- intento: Almacena el número introducido por el usuario.
- mensaje: Muestra mensajes sobre si el número es mayor, menor o si no se ha introducido nada.
- intentos: Cuenta el número de intentos realizados.
- anado: Indica si el usuario ha adivinado el número.
- mounted: Se utiliza para iniciar el juego generando el número secreto cuando la aplicación se carga.

En methods definimos los métodos principales de nuestro objeto Vue.

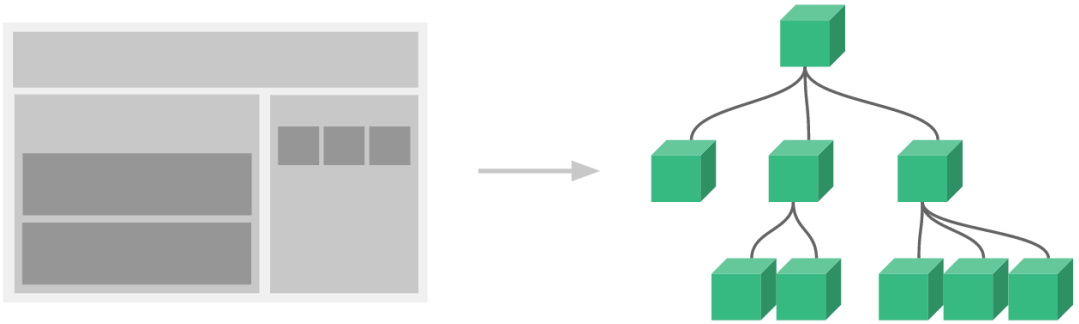
- **reiniciar:** Reinicia el juego, generando un nuevo número y reseteando las variables.
- **adivinar:** Contiene la lógica que compara el número introducido por el usuario con el número secreto y muestra los mensajes correspondientes.

20.6. Componentes

En Vue.js, un componente es una instancia de Vue con una plantilla, lógica de JavaScript y estilos. Los componentes son una forma de encapsular la funcionalidad y la interfaz de

usuario en unidades reutilizables que luego podremos ir colocando como una etiqueta HTML en distintas partes de nuestra aplicación.

En este sentido podemos disponer de una interfaz donde cada parte de la pantalla sea un componente. Podríamos tener un componente menú, otro que fuera barra lateral, etc. Dentro de cada uno de ellos podríamos tener a su vez varios componentes, por ejemplo uno que me mostrara los diez últimos artículos publicados que dando una figura en forma de árbol como se muestra en la siguiente ilustración.



```
<template>
  <div>
    <p>{{ mensaje }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mensaje: 'Hola desde el componente'
    };
  }
};
</script>

<style scoped>
p {
  color: blue;
}
</style>
```

- **<template>:** Contiene el HTML del componente.

- **<script>:** Define la lógica del componente, como datos, métodos y ciclo de vida.

- **<style scoped>**: Añade estilos específicos para el componente. El atributo scoped asegura que los estilos no afecten a otros componentes.

a) Componentes Globales y Locales

Los componentes globales se registran una vez y están disponibles en toda la aplicación Vue.

```
// main.js o archivo de entrada principal
Vue.component('componente-global', {
  template: '<div>Componente Global</div>'
});
```

Luego los llamaremos mediante su correspondiente etiqueta:
<componente-global></componente-global>

Los Componentes Locales se registran dentro de un componente y solo están disponibles en ese componente específico.

```
<template>
  <div>
    <componente-local></componente-local>
  </div>
</template>

<script>
import ComponenteLocal from './ComponenteLocal.vue';

export default {
  components: {
    ComponenteLocal
  }
};
</script>
```

b) Props y Eventos

Props: Permite pasar datos desde un componente padre a un componente hijo.

```
<template>
  <div>
```

```
    <p>{{ mensaje }}</p>
  </div>
</template>

<script>
export default {
  props: ['mensaje']
};
```

Luego esa propiedad podremos definirla como si fuera un atributo de su correspondiente etiqueta: **<componente-hijo mensaje="Hola desde el padre"></componente-hijo>**

Los Eventos permiten a los componentes hijos comunicar eventos al componente padre.

```
<template>
  <button @click="emitirEvento">Emitir Evento</button>
</template>

<script>
export default {
  methods: {
    emitirEvento() {
      this.$emit('evento-personalizado', 'Datos del evento');
    }
  }
};
</script>
```

Escuchar eventos:

```
<componente-hijo
@evento-personalizado="manejarEvento"></componente-hijo>

<script>
export default {
  methods: {
    manejarEvento(datos) {
      console.log(datos); // 'Datos del evento'
    }
  }
};
</script>
```

c) Slots y Scoped Slots

Los Slots permiten pasar contenido dinámico a un componente.

```
<template>
  <div>
    <slot></slot>
  </div>
</template>

<componente-slot>
  <p>Contenido del slot</p>
</componente-slot>
```

Los **Scoped Slots** permiten al componente hijo exponer datos al componente padre a través del slot.

```
<template>
  <div>
    <slot :mensaje="mensaje"></slot>
  </div>
</template>

<script>
export default {
  data() {
    return {
      mensaje: 'Hola desde el scoped slot'
    };
  }
};
</script>

<componente-scoped-slot v-slot:default="{ mensaje }">
  <p>{{ mensaje }}</p>
</componente-scoped-slot>
```

20.7. Componentes Dinámicos y Asíncronos

Componentes Dinámicos: Permiten renderizar componentes diferentes en función de la lógica.

```
<template>
  <component :is="componenteActual"></component>
</template>
```

```
<script>
import ComponenteA from './ComponenteA.vue';
import ComponenteB from './ComponenteB.vue';

export default {
  data() {
    return {
      componenteActual: ComponenteA
    };
  }
};
</script>
```

Componentes Asíncronos: Se cargan de manera diferida cuando se requieren, lo cual es útil para optimizar el rendimiento.

```
const ComponenteAsync = () => import('./ComponenteAsync.vue');
Uso de un componente asíncrono:

<template>
  <component :is="componenteAsync"></component>
</template>

<script>
export default {
  data() {
    return {
      componenteAsync: () => import('./ComponenteAsync.vue')
    };
  }
};
</script>
```

5. Enrutamiento con Vue Router
   - 5.1. Instalación y Configuración
   - 5.2. Definición de Rutas
   - 5.3. Enlaces y Navegación
   - 5.4. Rutas Anidadas y Parámetros

6. Gestión del Estado con Vuex
   - 6.1. Instalación y Configuración
   - 6.2. Estado, Mutaciones y Acciones
   - 6.3. Getters y Modulos
   - 6.4. Integración con Componentes

7. Formularios y Validación
   - 7.1. Manejo de Formularios
   - 7.2. Validación de Formularios
   - 7.3. Uso de Librerías de Validación

8. Comunicación entre Componentes
   - 8.1. Props y Eventos Personalizados
   - 8.2. Uso de Bus de Eventos
   - 8.3. Comunicación entre Componentes Hermanos

9. Directivas Personalizadas
   - 9.1. Creación de Directivas Personalizadas
   - 9.2. Uso y Aplicaciones

10. Transiciones y Animaciones
    - 10.1. Transiciones de Estado
    - 10.2. Animaciones con CSS
    - 10.3. Uso de Bibliotecas de Animación

11. Pruebas en Vue.js
    - 11.1. Pruebas Unitarias
    - 11.2. Pruebas de Integración
    - 11.3. Herramientas de Pruebas

12. Despliegue y Optimización
    - 12.1. Compilación para Producción
    - 12.2. Estrategias de Caché
    - 12.3. Optimización de Desempeño

13. Vue.js en la Práctica
    - 13.1. Casos de Uso Comunes
    - 13.2. Mejores Prácticas
    - 13.3. Recursos y Comunidades