Prompt: Como desarrollador de aplicaciones web, genera las instrucciones para agregar Javascript a un documento html. Resalta las buenas prácticas de programación.

Como desarrollador de aplicaciones web, incorporar JavaScript a un documento HTML es una tarea fundamental. Aquí te presento las instrucciones detalladas, resaltando las buenas prácticas de programación para asegurar un código eficiente, mantenible y robusto.

Cómo agregar JavaScript a un documento HTML

Existen tres métodos principales para agregar JavaScript a un documento HTML:

1. **JavaScript Interno (Embedded JavaScript):** Código JavaScript directamente dentro del documento HTML.
2. JavaScript Externo (External JavaScript): Código JavaScript en un archivo separado.
3. **JavaScript en Línea (Inline JavaScript):** Código JavaScript directamente en un atributo de una etiqueta HTML.

A continuación, se detalla cada método con sus respectivas buenas prácticas.

---

1. JavaScript Interno (Embedded JavaScript)

Este método implica colocar el código JavaScript dentro de la etiqueta <script> directamente en el documento HTML.

Instrucciones:

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 JavaScript Interno</title>
</head>
<body>
    <h1>Bienvenido a mi sitio web</h1>
    <p id="parrafoMensaje">Este es un párrafo de ejemplo.</p>

    <script>
        // *** Buenas Prácticas ***
        // 1. Coloca el script al final del <body>:
        //    Esto asegura que el HTML se cargue completamente antes de que el script intente manipularlo,
        //    evitando errores y mejorando la velocidad de carga percibida.
        // 2. Uso de 'use strict';:
        //    Activa el modo estricto de JavaScript, lo que ayuda a escribir código más seguro y a
        //    detectar errores comunes.

        'use strict';

        // 3. Comentarios descriptivos:
        //    Explica la lógica del código para facilitar su comprensión y mantenimiento.
        /**
         * Función que cambia el contenido de un párrafo.
         * @param {string} idElemento - El ID del elemento HTML a modificar.
         * @param {string} nuevoContenido - El nuevo contenido a asignar al elemento.
         */
        function cambiarContenidoParrafo(idElemento, nuevoContenido) {
            const elemento = document.getElementById(idElemento);
            if (elemento) { // 4. Verificación de existencia de elementos:
                            //    Siempre verifica si el elemento existe antes de intentar manipularlo
                            //    para evitar errores de "null".
                elemento.textContent = nuevoContenido;
            } else {
                console.warn(`Advertencia: El elemento con ID "${idElemento}" no fue encontrado.`);
            }
        }

        // Llamada a la función después de que el DOM esté disponible
        cambiarContenidoParrafo('parrafoMensaje', '¡El contenido de este párrafo ha sido cambiado por
JavaScript!');
    </script>
</body>
</html>
```

Buenas Prácticas Resaltadas:

- **Colocación del script al final del <body>:** Por defecto, el navegador detiene el parseo del HTML mientras descarga y ejecuta un script. Colocar los scripts al final

del <body> (justo antes de la etiqueta de cierre </body>) permite que el contenido HTML se renderice completamente antes de que se ejecute el JavaScript, lo que mejora la velocidad de carga percibida y evita errores al intentar manipular elementos que aún no existen en el DOM.

- **Uso de 'use strict';:** Al inicio de un script o función, el modo estricto ('use strict';) ayuda a detectar errores comunes, convierte algunos errores silenciosos en errores lanzados y prohíbe ciertas sintaxis inseguras o anticuadas.
- **Comentarios Descriptivos:** Utiliza comentarios claros y concisos para explicar la intención y la lógica de tu código. Para funciones complejas, considera el uso de JSDoc para documentar parámetros, retornos y una descripción general.
- **Verificación de Existencia de Elementos:** Antes de manipular un elemento del DOM (por ejemplo, con document.getElementById()), siempre verifica que el elemento haya sido encontrado para evitar errores si el ID es incorrecto o el elemento aún no se ha cargado.

---

2. JavaScript Externo (External JavaScript)

Este es el método más recomendado y ampliamente utilizado para agregar JavaScript. El código se coloca en un archivo .js separado y se vincula al documento HTML.

Instrucciones:

Paso 1: Crea un archivo JavaScript (por ejemplo, script.js)

JavaScript

```
// script.js

// *** Buenas Prácticas ***
'use strict'; // 1. Modo estricto

// 2. Encapsulamiento en funciones:
//    Agrupa la lógica relacionada en funciones para mejorar la organización y la reusabilidad.
/**
 * Inicializa la funcionalidad de la página cuando el DOM está completamente cargado.
 */
function inicializarPagina() {
    // 3. Event Listener para DOMContentLoaded:
```

```
    //    Asegura que el código se ejecute solo después de que el DOM esté listo,
    //    incluso si el script se carga en el <head> (aunque no es lo ideal sin 'defer').
    console.log("El DOM ha sido completamente cargado.");

    const boton = document.getElementById('miBoton');
    const mensajeParrafo = document.getElementById('parrafoMensajeExterno');

    if (boton && mensajeParrafo) {
        boton.addEventListener('click', function() { // 4. Uso de addEventListener:
                                                     //    Es la forma preferida de adjuntar eventos,
                                                     //    permitiendo múltiples manejadores de eventos y una mejor
separación de preocupaciones.
            mensajeParrafo.textContent = '¡El botón fue clickeado y el contenido ha cambiado!';
            console.log('Botón clickeado.');
        });
    } else {
        console.warn('Advertencia: No se encontraron todos los elementos necesarios (botón o
párrafo).');
    }
}

// 5. Cargar el script de forma asíncrona (si es necesario) o al final del body:
//    Si el script se carga en el <head> y no usa 'defer', usar DOMContentLoaded es crucial.
//    Si se carga al final del body, no es estrictamente necesario, pero es una buena práctica para
asegurar.
document.addEventListener('DOMContentLoaded', inicializarPagina);

// O directamente:
// inicializarPagina(); // Si el script está cargado al final del body
```

Paso 2: Vincula el archivo JavaScript en tu documento HTML

```
    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 JavaScript Externo</title>
</head>
<body>
    <h1>Ejemplo con JavaScript Externo</h1>
    <p id="parrafoMensajeExterno">Haz clic en el botón de abajo.</p>
    <button id="miBoton">Haz Clic Aquí</button>

    <script src="script.js" defer></script>
    </body>
</html>
```

Buenas Prácticas Resaltadas:

- Separación de Preocupaciones: Mantener JavaScript en archivos separados mejora la organización del código, facilita la depuración y promueve la reutilización. El HTML se encarga de la estructura, el CSS del estilo y el JavaScript del comportamiento.
- Caché del Navegador: Los archivos JavaScript externos pueden ser cacheados por el navegador, lo que resulta en tiempos de carga más rápidos para visitas posteriores.
- **Atributos defer y async (para scripts en el head):**
    - defer: El script se descarga en paralelo con el parseo del HTML y se ejecuta solo después de que el documento ha sido completamente parseado. Mantiene el orden relativo de los scripts defer. Es la opción más común y segura si necesitas cargar scripts en el head.
    - async: El script se descarga en paralelo con el parseo del HTML y se ejecuta tan pronto como esté disponible. No garantiza el orden de ejecución con respecto a otros scripts o al parseo del documento. Útil para scripts independientes que no dependen del DOM ni de otros scripts.
- **DOMContentLoaded Event Listener:** Si tu script manipula el DOM y se carga en el head sin defer, es crucial envolver tu lógica en un listener para el evento DOMContentLoaded. Este evento se dispara cuando el documento HTML ha sido completamente cargado y parseado, y el DOM está listo para ser manipulado, sin esperar a que se carguen las hojas de estilo o las imágenes. Si el script se carga al final del <body>, generalmente no es estrictamente necesario, pero sigue siendo una buena práctica para asegurar que el DOM esté listo.
- **Uso de addEventListener:** En lugar de asignaciones directas como onclick, addEventListener es la forma preferida de adjuntar manejadores de eventos. Permite adjuntar múltiples manejadores para el mismo evento en el mismo

elemento y ofrece mayor control.

- **Encapsulamiento en Funciones:** Organiza tu código en funciones para mejorar la modularidad, la legibilidad y la reusabilidad.
- Evitar Variables Globales: Minimiza el uso de variables globales para evitar colisiones de nombres y efectos secundarios no deseados. Utiliza const y let para declarar variables con ámbito de bloque.

---

3. JavaScript en Línea (Inline JavaScript)

Este método consiste en colocar el código JavaScript directamente dentro de un atributo HTML (por ejemplo, onclick, onmouseover). **Este método generalmente se considera una mala práctica** y debe evitarse en la medida de lo posible.

Instrucciones (para fines demostrativos, pero desaconsejado):

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 JavaScript en Línea (Evitar)</title>
</head>
<body>
    <h1>Ejemplo de JavaScript en Línea</h1>
    <p id="parrafoInline">Contenido inicial.</p>

    <button onclick="document.getElementById('parrafoInline').textContent = '¡Contenido cambiado en línea!'; alert('Se cambió el texto!');">
        Cambiar Texto (Mala Práctica)
    </button>
</body>
</html>
```

Por qué es una Mala Práctica:

- Mezcla de Contenido, Estilo y Comportamiento: Rompe el principio de separación de preocupaciones, dificultando el mantenimiento y la lectura del código.

- **Dificultad de Mantenimiento:** Si necesitas cambiar la funcionalidad, debes modificar el HTML directamente, lo que es propenso a errores y engorroso en sitios grandes.
- **Falta de Reusabilidad:** El código JavaScript en línea no es fácilmente reutilizable en otras partes del documento o en otras páginas.
- **Problemas de Seguridad:** Puede ser vulnerable a ataques de inyección de código (XSS) si el contenido del atributo se genera dinámicamente sin una sanitización adecuada.
- **No Obtrusivo:** El JavaScript no intrusivo es una buena práctica donde el JavaScript se añade al HTML utilizando métodos como addEventListener en lugar de atributos en línea.

Resumen de Buenas Prácticas Generales para JavaScript

- **Separación de Preocupaciones:** Mantén JavaScript, CSS y HTML en archivos separados siempre que sea posible.
- **Minimiza las Variables Globales:** Utiliza const y let para declarar variables con ámbito de bloque y evita la contaminación del espacio de nombres global.
- **Usa const y let en lugar de var:** const para valores que no cambian, let para valores que pueden reasignarse. var tiene un ámbito de función y puede llevar a comportamientos inesperados (hoisting).
- **Evita el Bloqueo del Renderizado:** Coloca los scripts al final del <body> o utiliza los atributos defer o async para scripts en el <head>.
- **Manejo de Errores:** Implementa bloques try...catch para manejar errores de forma elegante y evita que pequeños errores rompan toda la aplicación.
- Validación de Datos: Valida siempre los datos de entrada, especialmente aquellos provenientes del usuario o de APIs externas.
- Optimización del DOM: Las manipulaciones del DOM son costosas. Minimiza el número de veces que accedes y modificas el DOM. Agrupa cambios o utiliza fragmentos de documentos.
- **Comentarios y Documentación:** Documenta tu código de forma clara y concisa. Utiliza JSDoc para funciones y módulos complejos.
- **Consistencia en el Estilo del Código:** Utiliza un linter (como ESLint) y un formateador (como Prettier) para mantener un estilo de código consistente en todo el proyecto.
- **Pruebas Unitarias e Integración:** Escribe pruebas para tu código JavaScript para asegurar su funcionalidad y detectar regresiones.
- Rendimiento: Optimiza tu código JavaScript para un rendimiento óptimo, prestando atención a bucles, algoritmos y uso de memoria.

- **Accesibilidad:** Asegúrate de que tus interacciones JavaScript sean accesibles para todos los usuarios, incluyendo aquellos que utilizan tecnologías de asistencia.
- **Seguridad:** Ten en cuenta las vulnerabilidades de seguridad como XSS y CORS al escribir código JavaScript, especialmente si interactúa con datos de usuario o APIs externas.

Siguiendo estas instrucciones y buenas prácticas, podrás integrar JavaScript en tus aplicaciones web de manera efectiva, produciendo código limpio, eficiente y fácil de mantener.

Sources

1. https://github.com/Davidrdbj05/ASIR