Manual Completo de JavaScript: De Cero a Experto

Portada

Título: Manual Completo de JavaScript: De Cero a Experto Autor: Basdonax AI Serie: Fundamentos del Desarrollo Web

---

Nivel Básico: Los Fundamentos de JavaScript

1. Introducción al Desarrollo Web y JavaScript

Teoría

JavaScript es un **lenguaje de programación de alto nivel** que se utiliza principalmente para hacer que las páginas web sean interactivas. Es uno de los tres pilares fundamentales del desarrollo web moderno, junto con **HTML** (que define la estructura y el contenido) y **CSS** (que define la apariencia y el estilo).

Originalmente, JavaScript se ejecutaba solo en el navegador web del cliente (lo que se conoce como *frontend*), permitiendo manipular el contenido de la página (el DOM), manejar eventos y comunicarse con servidores. Sin embargo, con la llegada de **Node.js**, JavaScript también puede ejecutarse en el servidor (*backend*), lo que lo convierte en un lenguaje *full-stack*.

Nuestro enfoque en este manual será el **JavaScript puro** o **Vanilla JS**, que es la base sólida que todo desarrollador debe dominar antes de pasar a frameworks y librerías más complejas.

Código de Ejemplo Comentado

El primer paso en cualquier lenguaje es mostrar un mensaje. En JavaScript, la forma más común de hacerlo para depuración o para mostrar información simple es usando la consola del navegador.

```
// 1. Esto es un comentario. El código no lo ejecuta.
// 2. console es un objeto global que nos da acceso a la consola del navegador.
// 3. log() es un método de ese objeto que imprime el mensaje que le pasemos.
console.log("¡Hola, Mundo! Este es mi primer código JavaScript.");
```

Explicación:

- `console.log()` es una herramienta esencial para el desarrollador. Permite imprimir mensajes, variables y objetos en la consola del navegador para verificar el estado del programa.
- El texto entre comillas dobles o simples es una **cadena de texto** (String).

Ejercicio Práctico

1. Abre la consola de tu navegador (generalmente presionando `F12` o `Ctrl+Shift+I`).
2. Ve a la pestaña "Console".
3. Escribe el código de ejemplo y presiona `Enter`.
4. **Resultado Esperado:** Ver el mensaje "¡Hola, Mundo! Este es mi primer código JavaScript." impreso en la consola.

---

2. Variables y Tipos de Datos

Teoría

Una **variable** es un espacio de almacenamiento con nombre que se utiliza para guardar datos. En JavaScript moderno (ES6+), usamos principalmente dos palabras clave para declarar variables: `let` y `const`.

Palabra Clave	Propósito	¿Se puede reasignar?	¿Se puede redeclarar?	Alcance (Scope)
`const`	Para valores que **no cambiarán** (constantes).	No	No	De Bloque
`let`	Para valores que **pueden cambiar** (variables).	Sí	No	De Bloque
`var`	(Antigua) Evitar su uso en código moderno.	Sí	Sí	De Función

JavaScript es un lenguaje de **tipado dinámico**, lo que significa que no tienes que especificar el tipo de dato al declarar la variable; el tipo se determina automáticamente en tiempo de ejecución.

Los tipos de datos primitivos más comunes son:

- **String:** Cadenas de texto (ej. `"Hola"`).
- **Number:** Números enteros o decimales (ej. `10`, `3.14`).
- **Boolean:** Valores lógicos, solo `true` o `false`.
- **Null:** Ausencia intencional de cualquier valor.
- **Undefined:** Valor de una variable que ha sido declarada pero aún no se le ha asignado un valor.

Código de Ejemplo Comentado

```
// Declaración de una constante (no cambiará)
const nombreCompleto = "Ana María Pérez"; // Tipo String

// Declaración de una variable (puede cambiar)
let edad = 25; // Tipo Number

// Declaración de un booleano
let esEstudiante = true; // Tipo Boolean

// Una variable sin valor asignado es 'undefined'
let telefono;

// Reasignación de la variable 'edad' (permitido con let)
edad = 26;

// Uso de la interpolación de cadenas (Template Literals) con backticks (``)
console.log(`Hola, soy ${nombreCompleto}, tengo ${edad} años.`);
console.log(`¿Soy estudiante? ${esEstudiante}.`);
console.log(`El valor de 'telefono' es: ${telefono}.`);
```

Explicación:

- La sintaxis `${variable}` dentro de las comillas invertidas (“) permite insertar el valor de una variable directamente en una cadena de texto.
- Intentar reasignar `nombreCompleto` (declarada con `const`) resultaría en un error.

Ejercicio Práctico

Crea un script que guarde tu nombre, edad y profesión en variables adecuadas (`let` o `const`). Luego, muestra la siguiente frase en la consola usando interpolación de cadenas: “Mi nombre es [Nombre], tengo [Edad] años y trabajo como [Profesión].”

Solución (Oculta para el lector):

```
const miNombre = "Carlos";
let miEdad = 30;
const miProfesion = "Desarrollador Web";

console.log(`Mi nombre es ${miNombre}, tengo ${miEdad} años y trabajo como
${miProfesion}.`);
```

---

3. Operadores

Teoría

Los operadores son símbolos que le dicen a JavaScript que realice una operación entre valores y variables.

Tipo de Operador	Símbolo	Descripción	Ejemplo
Aritméticos	`+`, `-`, `*`, `/`	Suma, resta, multiplicación, división.	`5 + 3` resulta `8`
	`%`	Módulo (resto de la división).	`10 % 3` resulta `1`
	`**`	Exponenciación.	`2 ** 3` resulta `8`
Asignación	`=`	Asigna un valor a una variable.	`let x = 10`
	`+=`, `-=`, `*=`	Asigna el resultado de una operación.	`x += 5` es igual a `x = x + 5`
Comparación	`==`	Igualdad (compara solo el valor, ignora el tipo). **Evitar su uso.**	`'5' == 5` es `true`
	`===`	Igualdad estricta (compara valor **y** tipo). **Recomendado.**	`'5' === 5` es `false`
	`!=`, `!==`	Desigualdad y Desigualdad estricta.	`5 !== '5'` es `true`
	`>`, `<`, `>=`, `<=`	Mayor que, menor que, etc.	`10 > 5` es `true`
Lógicos	`&&`	AND lógico (ambas condiciones deben ser `true`).	`(A > 0) && (B < 10)`
	`||`	OR lógico (al menos una condición debe ser `true`).	`(A === 5) || (B === 5)`
	`!`	NOT lógico (invierte el valor booleano).	`!true` es `false`

Código de Ejemplo Comentado

Nos enfocaremos en la diferencia crucial entre la igualdad simple (`==`) y la estricta (`===`).

```
let numero = 10; // Tipo Number
let texto = "10"; // Tipo String
let booleano = true;

// 1. Igualdad simple (NO estricta)
// JavaScript intenta convertir los tipos para compararlos.
console.log(`'10' == 10: ${texto == numero}`); // Resultado: true
(¡Peligro!)

// 2. Igualdad estricta (RECOMENDADA)
// Compara el valor Y el tipo de dato.
console.log(`'10' === 10: ${texto === numero}`); // Resultado: false
(Correcto)

// 3. Operadores lógicos
const esMayorDeEdad = true;
const tienePermiso = false;

// AND (&&): Solo es true si ambos son true
const puedeConducir = esMayorDeEdad && tienePermiso;
console.log(`Puede conducir: ${puedeConducir}`); // Resultado: false

// NOT (!): Invierte el valor
console.log(`No es mayor de edad: ${!esMayorDeEdad}`); // Resultado: false
```

Explicación:

- La igualdad estricta (`===`) es la **mejor práctica** en JavaScript, ya que evita conversiones de tipo inesperadas que pueden llevar a errores difíciles de depurar.
- Los operadores lógicos son fundamentales para construir condiciones complejas.

Ejercicio Práctico

Escribe una expresión que determine si una persona es mayor de edad (variable `edad` mayor o igual a 18) **Y** tiene permiso (variable booleana `tienePermiso` en `true`). Muestra el resultado en la consola.

Solución (Oculta para el lector):

```
const edad = 20;
const tienePermiso = true;

const esApto = (edad >= 18) && tienePermiso;
console.log(`¿Es apto para el proceso? ${esApto}`); // Resultado: true
```

---

4. Estructuras de Control: Condicionales

Teoría

Las estructuras de control condicionales permiten que el programa tome decisiones y ejecute diferentes bloques de código basándose en si una condición es verdadera (`true`) o falsa (`false`).

La estructura más común es el bloque `if...else if...else` :

- El bloque `if` se ejecuta si la condición es verdadera.
- El bloque `else if` se evalúa si la condición anterior fue falsa.
- El bloque `else` se ejecuta si ninguna de las condiciones anteriores fue verdadera.

El operador **ternario** es una forma concisa de escribir un `if...else` simple en una sola línea.

La sentencia `switch` se utiliza para ejecutar diferentes acciones basadas en diferentes condiciones sobre una misma expresión. Es ideal cuando se tienen múltiples valores posibles para una variable.

Código de Ejemplo Comentado

```
let numero = 0;

// 1. Estructura if/else if/else
if (numero > 0) {
    console.log("El número es positivo.");
} else if (numero < 0) {
    console.log("El número es negativo.");
} else {
    // Se ejecuta si ninguna de las condiciones anteriores es true
    console.log("El número es cero.");
}

// 2. Operador Ternario
const mensaje = (numero === 0) ? "Es cero" : "No es cero";
console.log(mensaje); // Resultado: Es cero

// 3. Sentencia switch
const dia = 3;
let nombreDia;

switch (dia) {
    case 1:
        nombreDia = "Lunes";
        break; // Detiene la ejecución del switch
    case 2:
        nombreDia = "Martes";
        break;
    case 3:
        nombreDia = "Miércoles";
        break;
    default:
        nombreDia = "Día no válido o fin de semana";
}

console.log(`El día es: ${nombreDia}`); // Resultado: Miércoles
```

Explicación:

- El `break` en el `switch` es crucial; sin él, el código continuaría ejecutándose en el siguiente `case` (comportamiento conocido como *fall-through*).

- El operador ternario tiene la forma: `condición ? valor_si_true : valor_si_false`.

Ejercicio Práctico

Crea un `switch` que tome una variable `calificacion` (un número del 1 al 5) y muestre en la consola:

- "Excelente" si es 5.
- "Aprobado" si es 3 o 4.
- "Reprobado" si es 1 o 2.
- "Calificación no válida" si es cualquier otro número.

Solución (Oculta para el lector):

```
const calificacion = 4;
let resultado;

switch (calificacion) {
    case 5:
        resultado = "Excelente";
        break;
    case 4:
    case 3:
        resultado = "Aprobado";
        break;
    case 2:
    case 1:
        resultado = "Reprobado";
        break;
    default:
        resultado = "Calificación no válida";
}

console.log(`El resultado es: ${resultado}`); // Resultado: Aprobado
```

---

5. Estructuras de Control: Bucles (Loops)

Teoría

Los **bucles** o *loops* permiten ejecutar un bloque de código repetidamente. Son esenciales para automatizar tareas y procesar colecciones de datos.

Tipo de Bucle	Uso Principal	Sintaxis
`for`	Cuando se conoce el número exacto de iteraciones.	`for (inicialización; condición; incremento)`
`while`	Cuando la cantidad de iteraciones es desconocida y depende de una condición.	`while (condición)`
`do...while`	Similar a `while`, pero garantiza que el bloque se ejecute **al menos una vez**.	`do { ... } while (condición)`

Las sentencias `break` y `continue` se utilizan dentro de los bucles:

- `break`: Termina el bucle inmediatamente.
- `continue`: Salta la iteración actual y pasa a la siguiente.

Código de Ejemplo Comentado

```
// 1. Bucle for: Iterar 5 veces
console.log("--- Bucle For ---");
for (let i = 0; i < 5; i++) {
    // i++ es una abreviatura de i = i + 1
    console.log(`Iteración número: ${i}`);
}

// 2. Bucle while: Mientras la condición sea verdadera
console.log("--- Bucle While ---");
let contador = 0;
while (contador < 3) {
    console.log(`Contador: ${contador}`);
    contador++; // Es vital incrementar el contador para evitar un bucle
infinito
}

// 3. Uso de break
console.log("--- Bucle con Break ---");
for (let j = 1; j <= 10; j++) {
    if (j === 5) {
        console.log("Se encontró el 5. Terminando el bucle.");
        break; // Sale del bucle
    }
    console.log(`Número: ${j}`);
}
```

Explicación:

- El bucle `for` es el más estructurado: inicializa una variable (`let i = 0`), define la condición de parada (`i < 5`) y el paso de avance (`i++`).
- Un error común con `while` es olvidar la condición de avance, lo que resulta en un **bucle infinito** que congela el programa.

Ejercicio Práctico

Usa un bucle `while` para contar desde 10 hacia atrás hasta 0. Muestra cada número en la consola.

Solución (Oculta para el lector):

```javascript
let cuentaRegresiva = 10;

while (cuentaRegresiva >= 0) {
    console.log(`Cuenta: ${cuentaRegresiva}`);
    cuentaRegresiva--; // Decremento
}
```

---

6. Funciones: Bloques de Código Reutilizables

Teoría

Una **función** es un bloque de código diseñado para realizar una tarea particular. Es la herramienta fundamental para la **reutilización de código** y la organización de programas.

Las funciones pueden:

1. **Recibir datos** a través de **parámetros** (variables definidas en la declaración de la función).
2. **Procesar** esos datos.
3. **Devolver un resultado** usando la palabra clave `return`. Si no se usa `return`, la función devuelve implícitamente `undefined`.

En JavaScript, las funciones son **ciudadanos de primera clase**, lo que significa que pueden ser tratadas como cualquier otro valor: asignadas a variables, pasadas como argumentos a otras funciones y devueltas por otras funciones.

Código de Ejemplo Comentado

```
// 1. Declaración de función tradicional
function calcularAreaRectangulo(base, altura) {
    // base y altura son los parámetros
    const area = base * altura;
    return area; // Devuelve el resultado
}

// 2. Llamada a la función (ejecución)
const area1 = calcularAreaRectangulo(5, 10); // 5 y 10 son los argumentos
console.log(`El área del primer rectángulo es: ${area1}`); // Resultado: 50

// 3. Función asignada a una variable (Expresión de función)
const saludar = function(nombre) {
    return `¡Hola, ${nombre}! Bienvenido al manual.`;
};

// 4. Llamada a la función a través de la variable
const mensajeBienvenida = saludar("Estudiante");
console.log(mensajeBienvenida);
```

Explicación:

- La función `calcularAreaRectangulo` toma dos argumentos, los multiplica y devuelve el resultado.
- La diferencia entre **declaración** (`function nombre()`) y **expresión** (`const nombre = function()`) radica en el *hoisting* (elevación), un concepto que exploraremos más adelante, pero por ahora, la expresión de función asignada a `const` es la más recomendada.

Ejercicio Práctico

Crea una función llamada `esPar` que reciba un número como parámetro. La función debe devolver `true` si el número es par y `false` si es impar. (Pista: usa el operador módulo `%`).

Solución (Oculta para el lector):

```
function esPar(numero) {
    // Si el resto de la división por 2 es 0, es par.
    return numero % 2 === 0;
}

console.log(`¿El 4 es par? ${esPar(4)}`); // Resultado: true
console.log(`¿El 7 es par? ${esPar(7)}`); // Resultado: false
```

Resumen y Ejercicio Integrador del Nivel Básico

Resumen de Conceptos Claves

- **JavaScript** es el lenguaje de la interactividad web, funcionando tanto en el cliente (navegador) como en el servidor (Node.js).
- Las variables se declaran con **`const`** (para valores fijos) y **`let`** (para valores que cambian). Se debe evitar **`var`**.
- Se debe usar la **igualdad estricta (`===`)** para comparar valores y tipos, evitando la igualdad simple (`==`).
- Las estructuras de control (`if/else`, `switch`) permiten tomar **decisiones** en el código.
- Los bucles (`for`, `while`) permiten la **repetición** de tareas.
- Las **funciones** son bloques de código reutilizables que pueden recibir parámetros y devolver resultados con `return`.

Ejercicio Integrador

Crea un script que simule un sistema de descuento simple. El programa debe:

1. Declarar una constante `precioBase` con un valor (ej. 100).
2. Declarar una variable `descuento` (ej. 0.10 para 10%).
3. Usar una función llamada `calcularPrecioFinal` que reciba el precio base y el descuento.
4. Dentro de la función, usar un condicional `if` para verificar si el descuento es mayor a 0. Si lo es, calcula el precio final. Si no lo es, devuelve el precio base.

5. Muestra el precio final en la consola.

Solución (Oculta para el lector):

```
const precioBase = 100;
let descuento = 0.10; // 10%

function calcularPrecioFinal(base, porcentajeDescuento) {
    if (porcentajeDescuento > 0) {
        const montoDescuento = base * porcentajeDescuento;
        return base - montoDescuento;
    } else {
        return base;
    }
}

const precioFinal = calcularPrecioFinal(precioBase, descuento);
console.log(`El precio base es: $${precioBase}`);
console.log(`El descuento aplicado es: ${descuento * 100}%`);
console.log(`El precio final es: $${precioFinal}`);

// Prueba con descuento cero
descuento = 0;
console.log(`Precio final sin descuento: $${calcularPrecioFinal(precioBase,
descuento)}`);
```

Nivel Intermedio: Estructuras de Datos y Funciones Avanzadas

7. Arrays (Arreglos)

Teoría

Un **Array** (o arreglo) es una estructura de datos que permite almacenar una colección ordenada de elementos. Los elementos pueden ser de cualquier tipo de dato (números, cadenas, objetos, incluso otros arrays).

La característica clave de los arrays es que sus elementos están **indexados**, comenzando siempre desde el índice **cero** (`0`).

Creación de Arrays:

```
const miArray = [elemento1, elemento2, elemento3];
```

Métodos de Manipulación Básica:

Método	Descripción	Mutación	Ejemplo
`push()`	Agrega uno o más elementos al **final** del array.	Sí	`arr.push('nuevo')`
`pop()`	Elimina el **último** elemento y lo devuelve.	Sí	`arr.pop()`
`unshift()`	Agrega uno o más elementos al **inicio** del array.	Sí	`arr.unshift('primero')`
`shift()`	Elimina el **primer** elemento y lo devuelve.	Sí	`arr.shift()`
`length`	Propiedad que devuelve el número de elementos.	No	`arr.length`

Código de Ejemplo Comentado

```
// 1. Creación de un array de nombres
const nombres = ["Ana", "Carlos", "Elena", "David"];

// 2. Acceso a elementos por índice (siempre desde 0)
console.log(`El primer nombre es: ${nombres[0]}`); // Resultado: Ana
console.log(`El último nombre es: ${nombres[nombres.length - 1]}`); //
Resultado: David

// 3. Agregar un elemento al final
nombres.push("Felipe");
console.log(`Array después de push: ${nombres}`); // Resultado:
Ana,Carlos,Elena,David,Felipe

// 4. Eliminar el último elemento
const eliminado = nombres.pop();
console.log(`Elemento eliminado: ${eliminado}`); // Resultado: Felipe
console.log(`Array después de pop: ${nombres}`); // Resultado:
Ana,Carlos,Elena,David
```

Explicación:

- El índice `[0]` siempre apunta al primer elemento.
- La propiedad `length` es útil para conocer el tamaño del array y para acceder al último elemento.

Ejercicio Práctico

Tienes un array de tareas pendientes: `const tareas = ["Comprar pan", "Estudiar JS", "Hacer ejercicio"];`.

1. Agrega “Llamar al médico” al inicio del array.
2. Elimina “Comprar pan” (el primer elemento original).
3. Muestra el array final en la consola.

Solución (Oculta para el lector):

```
const tareas = ["Comprar pan", "Estudiar JS", "Hacer ejercicio"];

tareas.unshift("Llamar al médico");
tareas.shift();

console.log(tareas); // Resultado: ['Estudiar JS', 'Hacer ejercicio',
'Llamar al médico']
```

---

8. Objetos Literales

Teoría

Un **Objeto** es una colección de propiedades, donde cada propiedad tiene un nombre (clave) y un valor. Los objetos son la estructura de datos más fundamental en JavaScript, y se utilizan para representar entidades del mundo real.

Creación de Objetos Literales: Se definen usando llaves `{}`.

```
const miObjeto = {
    clave1: valor1,
    clave2: valor2
};
```

Acceso a Propiedades:

1. **Notación de Punto:** Es la más común y legible. `objeto.clave`
2. **Notación de Corchetes:** Útil cuando el nombre de la clave es dinámico (está en una variable) o contiene espacios/caracteres especiales. `objeto['clave']`

Desestructuración (Destructuring): Es una sintaxis de ES6 que permite extraer valores de arrays u objetos en variables distintas de forma concisa.

Código de Ejemplo Comentado

```
// 1. Creación de un objeto 'libro'
const libro = {
    titulo: "Manual de JavaScript",
    autor: "Basdonax AI",
    paginas: 450,
    disponible: true,
    mostrarInfo: function() { // Un método (función dentro de un objeto)
        console.log(`Título: ${this.titulo}, Páginas: ${this.paginas}`);
    }
};

// 2. Acceso a propiedades
console.log(`El título es: ${libro.titulo}`); // Notación de punto
console.log(`El autor es: ${libro['autor']}`); // Notación de corchetes

// 3. Modificación de una propiedad
libro.paginas = 500;

// 4. Llamada al método
libro.mostrarInfo();

// 5. Desestructuración
const { titulo, autor } = libro;
console.log(`Desestructurado: ${titulo} por ${autor}`);
```

Explicación:

- `this` dentro de un método de objeto se refiere al objeto mismo (`libro` en este caso).
- La desestructuración simplifica la extracción de propiedades, evitando escribir `libro.titulo` y `libro.autor` repetidamente.

Ejercicio Práctico

Crea un objeto `producto` con las propiedades `nombre`, `precio` y `cantidad`. Luego, usa la notación de punto para aumentar el precio en 10 unidades y muestra el nuevo precio en la consola.

Solución (Oculta para el lector):

```
const producto = {
    nombre: "Laptop",
    precio: 800,
    cantidad: 5
};

producto.precio += 10; // producto.precio = 810

console.log(`El nuevo precio es: ${producto.precio}`);
```

---

9. Funciones de Orden Superior (High-Order Functions)

Teoría

Una **Función de Orden Superior (HOF)** es una función que hace al menos una de las siguientes cosas:

1. Toma una o más funciones como argumentos (llamadas *callbacks*).
2. Devuelve una función como resultado.

Las HOF son la base de la **programación funcional** en JavaScript. Los métodos de array más utilizados son HOF, ya que reciben una función *callback* que se aplica a cada elemento.

Métodos de Array Esenciales (HOF):

Método	Propósito	Devuelve
`forEach()`	Ejecuta una función para cada elemento.	`undefined`
`map()`	Crea un **nuevo array** aplicando una función a cada elemento.	Nuevo Array
`filter()`	Crea un **nuevo array** con todos los elementos que cumplen una condición.	Nuevo Array
`reduce()`	Ejecuta una función reductora sobre cada elemento, devolviendo un **único valor**.	Valor Único

Código de Ejemplo Comentado

```
const numeros = [1, 2, 3, 4, 5];

// 1. forEach: Iterar y mostrar cada elemento
console.log("--- forEach ---");
numeros.forEach(function(num) {
    console.log(`Número: ${num}`);
});

// 2. map: Crear un nuevo array con los números duplicados
// Usamos una función flecha (ver siguiente sección) para concisión
const duplicados = numeros.map(num => num * 2);
console.log(`Array duplicado (map): ${duplicados}`); // Resultado: [2, 4, 6,
8, 10]

// 3. filter: Crear un nuevo array solo con los números pares
const pares = numeros.filter(num => num % 2 === 0);
console.log(`Array de pares (filter): ${pares}`); // Resultado: [2, 4]

// 4. reduce: Sumar todos los elementos del array
// 'acc' es el acumulador, 'curr' es el valor actual. El 0 es el valor
inicial del acumulador.
const sumaTotal = numeros.reduce((acc, curr) => acc + curr, 0);
console.log(`Suma total (reduce): ${sumaTotal}`); // Resultado: 15
```

Explicación:

- La gran ventaja de `map` y `filter` es que **no modifican el array original** (son inmutables), lo que facilita la depuración y previene efectos secundarios.
- `reduce` es extremadamente versátil y puede usarse para sumar, promediar, aplanar arrays, y más.

Ejercicio Práctico

Dado el array `const palabras = ["sol", "luna", "estrella", "planeta"];`, usa el método `filter` para crear un nuevo array que contenga solo las palabras con más de 4 letras.

Solución (Oculta para el lector):

```
const palabras = ["sol", "luna", "estrella", "planeta"];

const palabrasLargas = palabras.filter(palabra => palabra.length > 4);

console.log(palabrasLargas); // Resultado: ['estrella', 'planeta']
```

---

10. Funciones Flecha (Arrow Functions) y `this`

Teoría

Las **Funciones Flecha** (`=>`) son una sintaxis más corta y moderna para escribir expresiones de función, introducida en ES6.

Sintaxis Concisa: Si la función solo tiene una expresión y devuelve el resultado de esa expresión, se pueden omitir las llaves `{}` y la palabra clave `return`.

```
// Función tradicional
const sumar = function(a, b) {
    return a + b;
};

// Función Flecha concisa
const sumarFlecha = (a, b) => a + b;
```

El Contexto `this` (La Diferencia Clave): La diferencia más importante es cómo manejan la palabra clave `this`.

- **Funciones Tradicionales:** Definen su propio valor de `this` (que puede cambiar dependiendo de cómo se llama la función).
- **Funciones Flecha: No definen su propio `this`.** En su lugar, capturan el valor de `this` del contexto circundante (contexto léxico). Esto las hace ideales para *callbacks* y métodos donde se necesita mantener el `this` del objeto padre.

Código de Ejemplo Comentado

```
// 1. Ejemplo de sintaxis concisa
const cuadrado = x => x * x;
console.log(`El cuadrado de 5 es: ${cuadrado(5)}`); // Resultado: 25

// 2. Ejemplo de this en un objeto (Función Tradicional vs. Flecha)
const persona = {
    nombre: "Manu",
    edad: 30,

    // Método con función tradicional: 'this' se refiere a 'persona'
    saludarTradicional: function() {
        console.log(`Hola, soy ${this.nombre}`);
    },

    // Método con función flecha: ¡PELIGRO! 'this' no se refiere a 'persona'
    // Se refiere al objeto global (window o undefined en modo estricto)
    saludarFlecha: () => {
        console.log(`Hola, soy ${this.nombre}`); // 'this.nombre' será
undefined
    }
};

persona.saludarTradicional(); // Resultado: Hola, soy Manu
persona.saludarFlecha(); // Resultado: Hola, soy undefined (o similar)
```

Explicación:

- Las funciones flecha son excelentes para operaciones cortas y para *callbacks* (como en `map` o `filter`).
- **Regla de Oro:** Usa funciones tradicionales para métodos de objetos que necesiten acceder a sus propias propiedades con `this`. Usa funciones flecha para todo lo demás.

Ejercicio Práctico

Reescribe la siguiente función tradicional como una función flecha concisa:

```
function multiplicarPorDiez(numero) {
    return numero * 10;
}
```

Solución (Oculta para el lector):

```
const multiplicarPorDiez = numero => numero * 10;

console.log(multiplicarPorDiez(7)); // Resultado: 70
```

---

11. Introducción a la Programación Orientada a Objetos (POO)

Teoría

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que organiza el diseño de software en torno a **objetos**, en lugar de funciones y lógica.

Conceptos Clave:

- **Clase:** Es una plantilla o molde para crear objetos. Define las propiedades (datos) y los métodos (comportamiento) comunes a todos los objetos de ese tipo.
- **Objeto (Instancia):** Es una ocurrencia concreta de una clase.
- **Propiedad:** Las variables que describen el estado del objeto.
- **Método:** Las funciones que definen el comportamiento del objeto.
- **Constructor:** Un método especial que se ejecuta automáticamente al crear un nuevo objeto (`new Clase()`) y se usa para inicializar sus propiedades.

En JavaScript (ES6+), las clases se definen con la palabra clave `class`.

Código de Ejemplo Comentado

```
// 1. Definición de la clase 'Animal'
class Animal {
    // 2. El constructor se llama al crear una nueva instancia
    constructor(nombre, especie) {
        this.nombre = nombre; // Propiedad
        this.especie = especie; // Propiedad
    }

    // 3. Un método (comportamiento)
    saludar() {
        console.log(`Hola, soy ${this.nombre} y soy un ${this.especie}.`);
    }
}

// 4. Creación de objetos (instancias) a partir de la clase
const perro = new Animal("Fido", "Perro");
const gato = new Animal("Miau", "Gato");

// 5. Uso de los métodos del objeto
perro.saludar(); // Resultado: Hola, soy Fido y soy un Perro.
gato.saludar(); // Resultado: Hola, soy Miau y soy un Gato.

// 6. Acceso a propiedades
console.log(`La especie de Fido es: ${perro.especie}`);
```

Explicación:

- La palabra clave `new` se utiliza para crear una nueva instancia de la clase.
- El `constructor` recibe los valores iniciales y los asigna a las propiedades del objeto usando `this`.

Ejercicio Práctico

Crea una clase `Coche` con un constructor que reciba `marca` y `modelo`. Agrega un método llamado `obtenerDescripcion` que devuelva una cadena como: “Este es un [marca] [modelo]”.

Solución (Oculta para el lector):

```
class Coche {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    obtenerDescripcion() {
        return `Este es un ${this.marca} ${this.modelo}.`;
    }
}

const miCoche = new Coche("Toyota", "Corolla");
console.log(miCoche.obtenerDescripcion()); // Resultado: Este es un Toyota Corolla.
```

Resumen y Ejercicio Integrador del Nivel Intermedio

Resumen de Conceptos Claves

- **Arrays** almacenan colecciones ordenadas, indexadas desde cero. Métodos clave: `push`, `pop`, `shift`, `unshift`.
- **Objetos Literales** almacenan datos en pares clave-valor. Se accede con notación de punto o corchetes.
- Las **Funciones de Orden Superior (HOF)** como `map`, `filter` y `reduce` permiten manipular arrays de forma declarativa e inmutable.
- Las **Funciones Flecha** (`=>`) ofrecen una sintaxis concisa y un manejo léxico de `this`, siendo ideales para *callbacks*.
- La **POO** se implementa con la palabra clave `class`, usando el `constructor` para inicializar objetos.

Ejercicio Integrador

Crea un sistema de gestión de empleados.

1. Define una clase `Empleado` con propiedades `nombre`, `edad` y `salario`.
2. Crea un array llamado `nomina` con al menos tres instancias de `Empleado`.

3. Usa el método `filter` para crear un nuevo array llamado `salariosAltos` que contenga solo a los empleados con un salario superior a 50000.
4. Usa el método `map` para crear un array de cadenas que contenga solo los nombres de los empleados con salarios altos.
5. Muestra el array de nombres en la consola.

Solución (Oculta para el lector):

```
class Empleado {
    constructor(nombre, edad, salario) {
        this.nombre = nombre;
        this.edad = edad;
        this.salario = salario;
    }
}

const nomina = [
    new Empleado("Juan", 25, 45000),
    new Empleado("María", 35, 60000),
    new Empleado("Pedro", 40, 55000),
    new Empleado("Laura", 22, 40000)
];

// 3. Filtrar empleados con salario > 50000
const salariosAltos = nomina.filter(empleado => empleado.salario > 50000);

// 4. Mapear para obtener solo los nombres
const nombresSalariosAltos = salariosAltos.map(empleado => empleado.nombre);

console.log("Empleados con salarios altos:");
console.log(nombresSalariosAltos); // Resultado: ['María', 'Pedro']
```

Nivel Avanzado: Interacción con el Navegador y Asincronía

12. El Modelo de Objetos del Documento (DOM)

Teoría

El **Modelo de Objetos del Documento (DOM)** es una interfaz de programación para documentos HTML y XML. Representa la página web como un árbol de objetos, donde cada nodo del árbol es una parte del documento (un elemento, un atributo, un texto, etc.). JavaScript utiliza el DOM para acceder y manipular el contenido, la estructura y el estilo de un documento.

El objeto principal del DOM es el objeto `document`, que representa la página completa.

Métodos de Selección de Elementos Clave:

Método	Descripción	Devuelve
`document.getElementById('id')`	Selecciona un único elemento por su atributo `id`.	Un elemento o `null`
`document.querySelector('selector')`	Selecciona el primer elemento que coincide con el selector CSS especificado.	Un elemento o `null`
`document.querySelectorAll('selector')`	Selecciona todos los elementos que coinciden con el selector CSS especificado.	Una `NodeList` (colección de elementos)

Código de Ejemplo Comentado

Para este ejemplo, asumiremos que existe un archivo HTML simple con el siguiente contenido: `<p id="saludo">Hola, soy un párrafo.</p>`.

```
// 1. Seleccionar el elemento por su ID
const parrafoSaludo = document.getElementById('saludo');

// 2. Seleccionar el mismo elemento usando querySelector
const parrafoQuery = document.querySelector('#saludo');

// 3. Verificar si el elemento fue encontrado
if (parrafoSaludo) {
    // 4. Acceder y modificar el contenido de texto del elemento
    parrafoSaludo.textContent = "¡Hola! El DOM ha cambiado mi texto.";

    // 5. Acceder y modificar un atributo (ej. el atributo 'class')
    parrafoSaludo.className = 'destacado';

    console.log("Contenido del párrafo modificado con éxito.");
} else {
    console.log("El elemento con ID 'saludo' no fue encontrado.");
}
```

Explicación:

- `textContent` permite obtener o establecer el contenido de texto de un nodo.
- `className` permite obtener o establecer el valor del atributo `class` del elemento.

Ejercicio Práctico

Asumiendo que tienes un elemento con la clase CSS `.titulo-principal`, usa `document.querySelector` para seleccionarlo y luego cambia su estilo para que el color del texto sea azul (`style.color = 'blue'`).

Solución (Oculta para el lector):

```
// Asumiendo que existe un elemento <h1 class="titulo-principal">
const titulo = document.querySelector('.titulo-principal');

if (titulo) {
    titulo.style.color = 'blue';
    console.log("El color del título ha sido cambiado a azul.");
}
```

---

13. Manipulación del DOM

Teoría

La manipulación del DOM no se limita a modificar elementos existentes, sino que también incluye la capacidad de **crear, insertar y eliminar** nuevos elementos en la página.

Métodos de Creación e Inserción:

Método	Descripción
`document.createElement('tag')`	Crea un nuevo nodo de elemento HTML (ej. `div`, `p`, `li`).
`elementoPadre.appendChild(elementoHijo)`	Agrega un nodo al final de la lista de hijos del nodo padre.
`elementoPadre.prepend(elementoHijo)`	Agrega un nodo al inicio de la lista de hijos del nodo padre.
`elemento.remove()`	Elimina el elemento del DOM.

Código de Ejemplo Comentado

Asumiremos que existe un elemento contenedor con el ID `lista-dinamica`: `<ul id="lista-dinamica"></ul>`.

```
// 1. Seleccionar el contenedor (el <ul>)
const lista = document.getElementById('lista-dinamica');

// 2. Crear un nuevo elemento <li>
const nuevoItem = document.createElement('li');

// 3. Asignarle contenido de texto
nuevoItem.textContent = "Nuevo elemento agregado con JavaScript";

// 4. Agregar el nuevo elemento al final de la lista
if (lista) {
    lista.appendChild(nuevoItem);
    console.log("Elemento <li> agregado al final de la lista.");
}

// 5. Crear otro elemento y agregarlo al inicio
const primerItem = document.createElement('li');
primerItem.textContent = "¡Soy el primero!";
if (lista) {
    lista.prepend(primerItem);
    console.log("Elemento <li> agregado al inicio de la lista.");
}

// 6. Eliminar un elemento (ej. el primerItem después de 2 segundos)
setTimeout(() => {
    if (primerItem) {
        primerItem.remove();
        console.log("El primer elemento ha sido eliminado.");
    }
}, 2000);
```

Explicación:

- La creación de elementos se hace en memoria con `createElement()`. El elemento no aparece en la página hasta que se inserta con un método como `appendChild()` o `prepend()`.
- `setTimeout()` se usa aquí para demostrar la eliminación, pero es un concepto de asincronía que veremos en detalle más adelante.

Ejercicio Práctico

Crea un botón y un párrafo en tu HTML. Usa JavaScript para que, al cargar la página, el párrafo esté oculto (`style.display = 'none'`). Luego, crea un script que, al hacer clic en el botón, haga que el párrafo sea visible (`style.display = 'block'`).

Solución (Oculta para el lector):

```
// Asumiendo <button id="btn-mostrar">Mostrar</button> y <p id="texto-
oculto">Texto</p>
const boton = document.getElementById('btn-mostrar');
const texto = document.getElementById('texto-oculto');

// Ocultar al inicio
if (texto) {
    texto.style.display = 'none';
}

// Función para mostrar
if (boton && texto) {
    boton.onclick = function() {
        texto.style.display = 'block';
    };
}
```

---

14. Eventos y Manejo de Interacciones

Teoría

Los **eventos** son acciones que ocurren en el sistema que el navegador notifica a JavaScript. Ejemplos comunes incluyen un clic de ratón, una pulsación de tecla, el envío de un formulario o la carga de la página.

Para responder a estos eventos, se utiliza el método `addEventListener()` .

Sintaxis de `addEventListener` :

```
elemento.addEventListener('tipoDeEvento', funcionManejadora);
```

Tipo de Evento Común	Descripción
`click`	El usuario hace clic en el elemento.
`mouseover`	El puntero del ratón se mueve sobre el elemento.
`submit`	Se envía un formulario.
`keydown`	Una tecla es presionada.
`load`	La página o un recurso ha terminado de cargarse.

La función manejadora recibe un objeto `event` como argumento, que contiene información útil sobre el evento, como la posición del ratón o el elemento que lo disparó (`event.target`).

Código de Ejemplo Comentado

Asumiremos un botón con ID `contador-btn` y un párrafo con ID `contador-texto`.

```
const botonContador = document.getElementById('contador-btn');
const textoContador = document.getElementById('contador-texto');
let contador = 0;

// 1. Función que se ejecutará cuando ocurra el evento
function manejarClick(evento) {
    contador++;
    textoContador.textContent = `Clicks: ${contador}`;

    // 2. El objeto 'evento' permite acceder a detalles
    console.log(`Tipo de evento: ${evento.type}`);
}

// 3. Registrar el 'listener' (escuchador)
if (botonContador && textoContador) {
    botonContador.addEventListener('click', manejarClick);
    console.log("Escuchador de eventos 'click' registrado.");
}

// 4. Ejemplo de prevención de comportamiento por defecto
const formulario = document.querySelector('form');
if (formulario) {
    formulario.addEventListener('submit', function(e) {
        // Previene que el formulario se envíe y recargue la página
        e.preventDefault();
        console.log("Formulario interceptado. Envío prevenido.");
    });
}
```

Explicación:

- `addEventListener` es la forma moderna y recomendada de manejar eventos, ya que permite registrar múltiples manejadores para el mismo evento en el mismo elemento.
- `e.preventDefault()` es crucial para controlar el flujo de formularios y enlaces.

Ejercicio Práctico

Crea un campo de texto (`<input type="text">`). Usa el evento `keyup` para que, cada vez que el usuario presione una tecla, el contenido del campo se muestre en tiempo real en un párrafo debajo.

Solución (Oculta para el lector):

```
// Asumiendo <input id="entrada"> y <p id="salida"></p>
const entrada = document.getElementById('entrada');
const salida = document.getElementById('salida');

if (entrada && salida) {
    entrada.addEventListener('keyup', function(e) {
        salida.textContent = `Escribiste: ${e.target.value}`;
    });
}
```

---

15. Programación Asíncrona: Callbacks y Event Loop

Teoría

JavaScript es un lenguaje de **un solo hilo** (*single-threaded*), lo que significa que solo puede ejecutar una tarea a la vez. Si una tarea tarda mucho (como una petición de red o una operación de disco), el programa se "bloquea" y la interfaz de usuario deja de responder.

La **Programación Asíncrona** es la solución. Permite que las operaciones largas se ejecuten en segundo plano sin bloquear el hilo principal.

El **Event Loop** es el mecanismo que permite a JavaScript manejar la asincronía. Funciona monitoreando la **Pila de Llamadas** (*Call Stack*) y la **Cola de Tareas** (*Task Queue*). Cuando la Pila de Llamadas está vacía (es decir, el hilo principal está libre), el Event Loop toma la primera tarea de la Cola y la empuja a la Pila para su ejecución.

Funciones de Retraso:

- `setTimeout(callback, delay)`: Ejecuta la función `callback` una sola vez después de un `delay` (en milisegundos).
- `setInterval(callback, delay)`: Ejecuta la función `callback` repetidamente cada `delay` milisegundos.

Código de Ejemplo Comentado

```
console.log("1. Inicio del script");

// 2. Simulación de una operación asíncrona
setTimeout(function() {
    // Esta función (callback) se va a la Cola de Tareas
    console.log("3. Tarea asíncrona completada (después de 2 segundos)");
}, 2000); // 2000 milisegundos = 2 segundos

console.log("2. Fin del script (el hilo principal no espera)");

// Resultado en consola:
// 1. Inicio del script
// 2. Fin del script (el hilo principal no espera)
// (2 segundos después)
// 3. Tarea asíncrona completada (después de 2 segundos)
```

Explicación:

- El `setTimeout` no detiene el programa. Simplemente programa la función para que se ejecute más tarde.
- El mensaje "Fin del script" se imprime antes que el mensaje del `setTimeout`, demostrando que el hilo principal continuó su ejecución sin esperar.

Ejercicio Práctico

Crea un contador que se inicie en 10 y se actualice cada segundo, mostrando el número actual en la consola. El contador debe detenerse cuando llegue a 0. (Pista: usa `setInterval` y `clearInterval`).

Solución (Oculta para el lector):

```
let cuenta = 10;

const intervaloID = setInterval(() => {
    console.log(`Tiempo restante: ${cuenta}`);
    cuenta--;

    if (cuenta < 0) {
        clearInterval(intervaloID); // Detiene el intervalo
        console.log("¡Tiempo terminado!");
    }
}, 1000); // 1000 milisegundos = 1 segundo
```

---

16. Programación Asíncrona: Promesas

Teoría

Las **Promesas** (`Promise`) son un objeto que representa la eventual finalización (o fracaso) de una operación asíncrona y su valor resultante. Fueron introducidas para resolver el problema del *Callback Hell* (anidamiento excesivo de callbacks).

Una Promesa tiene tres estados posibles:

1. **Pending (Pendiente):** El estado inicial, ni cumplida ni rechazada.
2. **Fulfilled (Cumplida):** La operación se completó con éxito.
3. **Rejected (Rechazada):** La operación falló.

Se utilizan los métodos `.then()` para manejar el éxito y `.catch()` para manejar el error.

Código de Ejemplo Comentado

```
// 1. Función que devuelve una Promesa
function simularCarga(exito) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (exito) {
                // Si la operación fue exitosa, llamamos a resolve()
                resolve("Datos del servidor cargados correctamente.");
            } else {
                // Si la operación falló, llamamos a reject()
                reject("Error: Falló la conexión con el servidor.");
            }
        }, 1500);
    });
}

// 2. Consumo de la Promesa (caso de éxito)
simularCarga(true)
    .then(resultado => {
        console.log(`Éxito: ${resultado}`); // Se ejecuta si resolve() es
llamado
    })
    .catch(error => {
        console.error(`Error: ${error}`); // Se ejecuta si reject() es
llamado
    });

// 3. Consumo de la Promesa (caso de error)
simularCarga(false)
    .then(resultado => {
        // No se ejecuta
    })
    .catch(error => {
        console.error(`Fallo: ${error}`); // Se ejecuta
    });
```

Explicación:

- El constructor `new Promise()` recibe una función con dos argumentos: `resolve` y `reject`.
- El encadenamiento de `.then()` permite ejecutar código secuencialmente, evitando el anidamiento.

Ejercicio Práctico

Crea una función que devuelva una promesa que se resuelva después de 3 segundos con el mensaje "Promesa cumplida". Usa `.then()` para mostrar el mensaje en la consola.

Solución (Oculta para el lector):

```
function promesaDeEspera() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Promesa cumplida");
        }, 3000);
    });
}

promesaDeEspera().then(mensaje => {
    console.log(mensaje);
});
```

---

17. Programación Asíncrona: `async/await`

Teoría

`async/await` es una sintaxis introducida en ES8 que permite escribir código asíncrono que parece y se comporta como código síncrono. Es la forma más legible y moderna de trabajar con Promesas.

- La palabra clave **`async`** se coloca antes de una función para indicar que la función siempre devolverá una Promesa. Si la función devuelve un valor que no es una Promesa, JavaScript lo envuelve automáticamente en una Promesa resuelta.
- La palabra clave **`await`** solo puede usarse dentro de una función `async`. Hace que la ejecución de la función `async` se pause hasta que la Promesa a la que se aplica se resuelva, y luego devuelve el valor resultante.

El manejo de errores se realiza con el bloque `try...catch`, al igual que en el código síncrono.

Código de Ejemplo Comentado

Reescribiremos el ejemplo de la sección anterior usando `async/await`.

```
// La función que devuelve la Promesa (no necesita cambios)
function obtenerDatos() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Datos cargados con async/await."), 2000);
    });
}

// 1. Declarar la función como asíncrona
async function mostrarDatos() {
    console.log("Iniciando carga...");

    try {
        // 2. Usar await para esperar la resolución de la Promesa
        const mensaje = await obtenerDatos();

        // 3. Este código solo se ejecuta después de que la Promesa se
resuelve
        console.log(`Resultado: ${mensaje}`);
    } catch (error) {
        // 4. Captura cualquier error (reject) de la Promesa
        console.error(`Ocurrió un error: ${error}`);
    }

    console.log("Función mostrarDatos finalizada.");
}

mostrarDatos();
console.log("El hilo principal sigue ejecutándose (no espera a
mostrarDatos).");
```

Explicación:

- Aunque `await` pausa la función `mostrarDatos`, **no bloquea el hilo principal** de JavaScript. El código fuera de la función `async` (como el último `console.log`) continúa ejecutándose inmediatamente.
- El bloque `try...catch` reemplaza al `.catch()` de las Promesas, haciendo el manejo de errores más familiar.

Ejercicio Práctico

Crea una función `async` llamada `obtenerUsuario` que simule la obtención de un usuario después de 1 segundo. Dentro de `obtenerUsuario`, usa `await` para esperar la Promesa y luego muestra el nombre del usuario en la consola.

Solución (Oculta para el lector):

```
function simularAPI() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve({ id: 1, nombre: "Juan Pérez" });
        }, 1000);
    });
}

async function obtenerUsuario() {
    try {
        const usuario = await simularAPI();
        console.log(`Usuario obtenido: ${usuario.nombre}`);
    } catch (error) {
        console.error("Fallo al obtener usuario.");
    }
}

obtenerUsuario();
```

Resumen y Ejercicio Integrador del Nivel Avanzado

Resumen de Conceptos Claves

- El **DOM** es la representación en árbol de la página web, accesible a través del objeto `document`.
- Se seleccionan elementos con `getElementById`, `querySelector` y `querySelectorAll`.
- Se manipulan elementos con `textContent`, `className`, `createElement`, `appendChild`, y `remove`.

- Los **Eventos** se manejan con `addEventListener`, y `e.preventDefault()` detiene la acción por defecto.
- La **Asincronía** se maneja con el **Event Loop** para evitar el bloqueo del hilo principal.
- Las **Promesas** (`Promise`) representan el resultado futuro de una operación asíncrona, con manejo de éxito (`.then()`) y error (`.catch()`).
- **`async/await`** es la sintaxis moderna para consumir Promesas de forma más legible, usando `try...catch` para el manejo de errores.

Ejercicio Integrador

Crea un script que simule la carga de un recurso y lo muestre en un elemento del DOM.

1. Crea una función que devuelva una Promesa que se resuelva después de 2 segundos con el texto "¡Contenido cargado!".
2. Crea una función `async` llamada `cargarContenido`.
3. Dentro de `cargarContenido`, usa `await` para esperar la Promesa.
4. Selecciona un elemento del DOM (ej. un `<div id="resultado">`) y usa el resultado de la Promesa para actualizar su `textContent`.
5. Mientras espera, el elemento debe mostrar "Cargando...".

Solución (Oculta para el lector):

```
// Asumiendo <div id="resultado"></div>
const resultadoDiv = document.getElementById('resultado');

function simularCarga() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve("¡Contenido cargado!");
        }, 2000);
    });
}

async function cargarContenido() {
    if (resultadoDiv) {
        resultadoDiv.textContent = "Cargando..."; // Estado inicial
    }

    try {
        const contenido = await simularCarga();

        if (resultadoDiv) {
            resultadoDiv.textContent = contenido; // Resultado final
        }
    } catch (error) {
        if (resultadoDiv) {
            resultadoDiv.textContent = "Error al cargar el contenido.";
        }
    }
}

cargarContenido();
```

Nivel Experto: APIs del Navegador y Modularización

18. Consumo de APIs con `fetch`

Teoría

La **Fetch API** es la interfaz moderna y basada en Promesas de JavaScript para realizar peticiones de red (como peticiones HTTP) a servidores. Reemplaza al antiguo

`XMLHttpRequest` y es la herramienta estándar para consumir servicios web (APIs).

Peticiones Comunes:

- **GET:** Se utiliza para solicitar datos de un recurso especificado. Es la petición por defecto.
- **POST:** Se utiliza para enviar datos a un servidor para crear o actualizar un recurso.

La función `fetch()` devuelve una **Promesa** que se resuelve con el objeto `Response`. Este objeto `Response` no contiene directamente los datos JSON, sino que proporciona métodos para acceder al cuerpo de la respuesta, como `response.json()` o `response.text()`, los cuales también devuelven Promesas.

Código de Ejemplo Comentado

Usaremos una API pública de prueba (`JSONPlaceholder`) para obtener una lista de usuarios.

```
const API_URL = 'https://jsonplaceholder.typicode.com/users';

async function obtenerUsuarios() {
    console.log("Iniciando petición a la API...");

    try {
        // 1. Petición GET (por defecto)
        const respuesta = await fetch(API_URL);

        // 2. Verificar si la respuesta fue exitosa (código 200-299)
        if (!respuesta.ok) {
            // Lanza un error si el estado HTTP no es OK (ej. 404, 500)
            throw new Error(`Error HTTP: ${respuesta.status}`);
        }

        // 3. Convertir la respuesta a JSON (esto también es asíncrono)
        const datosUsuarios = await respuesta.json();

        // 4. Mostrar los datos obtenidos
        console.log("Datos de usuarios obtenidos:");
        datosUsuarios.forEach(usuario => {
            console.log(`- ${usuario.name} (${usuario.email})`);
        });

    } catch (error) {
        // 5. Capturar errores de red o los lanzados en el paso 2
        console.error("Fallo al obtener los datos:", error.message);
    }
}

obtenerUsuarios();
```

Explicación:

- El uso de `async/await` simplifica enormemente el manejo de las dos Promesas que devuelve `fetch` (la respuesta y la conversión a JSON).
- Es fundamental verificar la propiedad `response.ok` para manejar errores del servidor.

Ejercicio Práctico

Obtén un solo post de la API (`https://jsonplaceholder.typicode.com/posts/1`) y muestra su título y cuerpo en la consola.

Solución (Oculta para el lector):

```
async function obtenerPost() {
    try {
        const respuesta = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
        if (!respuesta.ok) throw new Error('Fallo la petición');

        const post = await respuesta.json();

        console.log(`Título del Post: ${post.title}`);
        console.log(`Cuerpo del Post: ${post.body.substring(0, 50)}...`);
    } catch (error) {
        console.error(error.message);
    }
}

obtenerPost();
```

---

19. Almacenamiento Local: `localStorage` y `sessionStorage`

Teoría

Las APIs de **Web Storage** (`localStorage` y `sessionStorage`) permiten a las aplicaciones web almacenar datos de forma persistente en el navegador del usuario. Esto es útil para guardar preferencias, estados de la aplicación o datos que no necesitan ser enviados al servidor constantemente.

Almacenamiento	Persistencia	Capacidad	Uso Común
`localStorage`	**Persiste** incluso después de cerrar el navegador.	5-10 MB	Preferencias de usuario (tema oscuro), datos de sesión.
`sessionStorage`	Se borra cuando el usuario **cierra la pestaña** del navegador.	5-10 MB	Datos temporales de una sesión (carrito de compras).

Métodos Comunes:

- `setItem(clave, valor)`: Almacena un par clave-valor.
- `getItem(clave)`: Recupera el valor asociado a la clave.
- `removeItem(clave)`: Elimina la clave y su valor.
- `clear()`: Elimina todas las entradas.

Nota Importante: Web Storage solo almacena **cadenas de texto (Strings)**. Para guardar objetos o arrays, es necesario convertirlos a String usando `JSON.stringify()` antes de guardarlos, y convertirlos de vuelta usando `JSON.parse()` al recuperarlos.

Código de Ejemplo Comentado

```
const claveTema = 'preferenciaTema';
const claveUsuario = 'datosUsuario';

// 1. Guardar una cadena simple (String)
localStorage.setItem(claveTema, 'oscuro');
console.log(`Tema guardado: ${localStorage.getItem(claveTema)}`);

// 2. Guardar un Objeto (necesita serialización)
const usuario = {
    nombre: "Basdonax",
    id: 101,
    ultimaConexion: new Date().toISOString()
};

// Convertir el objeto a String JSON antes de guardar
localStorage.setItem(claveUsuario, JSON.stringify(usuario));

// 3. Recuperar el Objeto (necesita deserialización)
const usuarioString = localStorage.getItem(claveUsuario);
if (usuarioString) {
    const usuarioRecuperado = JSON.parse(usuarioString);
    console.log(`Usuario recuperado: ${usuarioRecuperado.nombre}`);
    console.log(`Tipo de dato recuperado: ${typeof usuarioRecuperado}`); //
Resultado: object
}

// 4. Eliminar un elemento
// localStorage.removeItem(claveTema);
```

Explicación:

- `JSON.stringify()` convierte un objeto JavaScript en una cadena JSON.
- `JSON.parse()` convierte una cadena JSON de vuelta a un objeto JavaScript.

Ejercicio Práctico

Guarda un array de números `[10, 20, 30]` en `sessionStorage`. Luego, recupéralo, agrégale el número 40 y guárdalo de nuevo.

Solución (Oculta para el lector):

```
const claveNumeros = 'misNumeros';
const arrayOriginal = [10, 20, 30];

// 1. Guardar el array (serializado)
sessionStorage.setItem(claveNumeros, JSON.stringify(arrayOriginal));

// 2. Recuperar y deserializar
const arrayString = sessionStorage.getItem(claveNumeros);
let arrayRecuperado = JSON.parse(arrayString);

// 3. Modificar y guardar de nuevo
arrayRecuperado.push(40);
sessionStorage.setItem(claveNumeros, JSON.stringify(arrayRecuperado));

console.log(`Array final en sessionStorage:
${sessionStorage.getItem(claveNumeros)}`);
```

---

20. Modularización con Módulos ES6

Teoría

Antes de ES6 (ECMAScript 2015), JavaScript no tenía un sistema de módulos nativo, lo que llevaba a problemas de colisión de nombres en el alcance global. Los **Módulos ES6** resuelven esto, permitiendo dividir el código en archivos separados (módulos) con su propio alcance (scope) privado.

Ventajas de la Modularización:

- **Organización:** Código más limpio y fácil de mantener.
- **Reutilización:** Las funciones pueden ser exportadas e importadas en diferentes partes del proyecto.
- **Aislamiento:** Las variables y funciones dentro de un módulo son privadas por defecto, evitando conflictos globales.

Sintaxis Clave:

- `export`: Se utiliza para exponer funciones, variables o clases de un módulo.
- `import`: Se utiliza para acceder a las funciones, variables o clases exportadas por otro módulo.

Tipo de Exportación	Sintaxis	Uso
Nombrada	`export const nombre = ...`	Para exportar múltiples elementos.
Por Defecto	`export default funcion/clase`	Solo puede haber una por módulo.

Nota: Para que el navegador reconozca los módulos, el script principal debe ser cargado con el atributo `type="module"` en el HTML: `<script type="module" src="main.js"></script>`.

Código de Ejemplo Comentado

Archivo: `utilidades.js`

```
// Exportación nombrada
export const PI = 3.14159;

// Exportación nombrada de una función
export function sumar(a, b) {
    return a + b;
}

// Exportación por defecto (solo una por archivo)
export default class Calculadora {
    multiplicar(a, b) {
        return a * b;
    }
}
```

Archivo: `main.js`

```
// 1. Importación nombrada (se usa el nombre exacto)
import { PI, sumar } from './utilidades.js';

// 2. Importación por defecto (se puede usar cualquier nombre)
import MiCalculadora from './utilidades.js';

console.log(`El valor de PI es: ${PI}`);
console.log(`Suma de 5 + 3: ${sumar(5, 3)}`);

const calc = new MiCalculadora();
console.log(`Multiplicación de 4 * 2: ${calc.multiplicar(4, 2)}`);
```

Explicación:

- Las rutas de importación (`./utilidades.js`) deben ser relativas o absolutas y deben incluir la extensión `.js` .
- La importación por defecto no usa llaves `{}` y se le puede dar cualquier nombre local.

Ejercicio Práctico

Crea un archivo `constantes.js` que exporte una constante `IVA = 0.21` . Luego, impórtala en tu archivo principal y úsala para calcular el precio final de un producto de $100.

Solución (Oculta para el lector): Archivo: `constantes.js`

```
export const IVA = 0.21;
```

Archivo: `main.js`

```
import { IVA } from './constantes.js';

const precio = 100;
const precioFinal = precio * (1 + IVA);

console.log(`Precio final con IVA: ${precioFinal}`); // Resultado: 121
```

---

21. Manejo de Errores Avanzado

Teoría

Un manejo de errores robusto es crucial para cualquier aplicación de nivel experto. JavaScript utiliza el mecanismo de **excepciones** para manejar errores.

Bloque `try...catch...finally`:

- **`try`:** Contiene el código que se quiere ejecutar y que podría generar un error.
- **`catch (error)`:** Contiene el código que se ejecuta si ocurre un error en el bloque `try`. El objeto `error` contiene información sobre la excepción.
- **`finally`:** Contiene el código que se ejecuta siempre, haya o no un error.

Lanzamiento de Errores: La palabra clave **`throw`** se utiliza para crear y lanzar una excepción personalizada. Es una buena práctica lanzar objetos `Error` para proporcionar información estandarizada.

Manejo de Errores en Asincronía: En las funciones `async`, el bloque `try...catch` captura automáticamente los errores (`reject`) de las Promesas a las que se aplica `await`.

Código de Ejemplo Comentado

```
function dividir(a, b) {
    if (b === 0) {
        // 1. Lanzar un error personalizado
        throw new Error("Error de División: No se puede dividir por cero.");
    }
    return a / b;
}

function ejecutarDivision(num1, num2) {
    try {
        // 2. Código que puede fallar
        const resultado = dividir(num1, num2);
        console.log(`Resultado: ${resultado}`);
    } catch (error) {
        // 3. Captura el error lanzado
        console.error(`¡Ocurrió un error! Mensaje: ${error.message}`);
    } finally {
        // 4. Se ejecuta siempre
        console.log("Proceso de división finalizado.");
    }
}

ejecutarDivision(10, 2); // Ejecución exitosa
ejecutarDivision(10, 0); // Ejecución con error
```

Explicación:

- El objeto `Error` tiene una propiedad `message` que contiene la descripción del error.
- El uso de `try...catch` previene que un error detenga la ejecución de todo el programa.

Ejercicio Práctico

Implementa un bloque `try...catch` en una función `async` que use `fetch` para manejar un error de red. Simula el error cambiando la URL a una que no existe.

Solución (Oculta para el lector):

```
async function obtenerDatosSeguro() {
    const URL_INVALIDA = 'https://una-url-que-no-existe.com/datos';

    try {
        const respuesta = await fetch(URL_INVALIDA);
        if (!respuesta.ok) {
            throw new Error(`Error HTTP: ${respuesta.status}`);
        }
        const datos = await respuesta.json();
        console.log("Datos obtenidos:", datos);
    } catch (error) {
        // Captura errores de red (ej. DNS no resuelto) o errores HTTP
        console.error("Fallo catastrófico en la petición:", error.message);
    }
}

obtenerDatosSeguro();
```

---

22. Buenas Prácticas y Estándares Modernos

Teoría

Adoptar buenas prácticas y estándares modernos (ES6+) es lo que diferencia a un desarrollador experto. Un código limpio, predecible y bien estructurado es más fácil de mantener y escalar.

Práctica	Descripción	Ejemplo
Uso de `const` y `let`	Usar `const` por defecto. Solo usar `let` si la variable necesita ser reasignada. **Nunca usar `var`**.	`const nombre = "Juan";`
Inmutabilidad	Evitar modificar objetos y arrays directamente. Usar métodos que devuelvan nuevas copias (`map`, `filter`, *spread operator* `...`).	`const nuevoArray = [...arrayOriginal, nuevoElemento];`
Nombres Descriptivos	Usar nombres de variables, funciones y clases que describan claramente su propósito.	`calcularPrecioFinal` en lugar de `calcP`
Comentarios y Documentación	Comentar el *por qué* del código complejo, no el *qué*. Usar JSDoc para documentar funciones.	`// Necesario para compatibilidad con IE11`
Funciones Puras	Funciones que, dado el mismo input, siempre devuelven el mismo output y no tienen efectos secundarios (no modifican variables externas).	`const sumar = (a, b) => a + b;`

Código de Ejemplo Comentado

```
// Mal: Mutación directa del array (efecto secundario)
const listaTareas = ["Comprar", "Pagar"];
listaTareas.push("Estudiar");

// Bien: Inmutabilidad con Spread Operator (...)
const listaTareasOriginal = ["Comprar", "Pagar"];
// Crea un nuevo array con los elementos del original más el nuevo
const nuevaListaTareas = [...listaTareasOriginal, "Estudiar"];

console.log("Lista Original (mutada):", listaTareas); // Muestra el cambio
console.log("Lista Original (inmutable):", listaTareasOriginal); // No
muestra el cambio
console.log("Nueva Lista:", nuevaListaTareas);
```

Explicación:

- El **Spread Operator** (`...`) es una herramienta poderosa de ES6 que permite expandir un iterable (como un array) en sus elementos individuales. Es clave para la inmutabilidad.

Ejercicio Práctico

Refactoriza el siguiente código para usar `const` y `let` correctamente, y usa una función flecha concisa.

```
var x = 5;
var y = 10;
function multiplicar(a, b) {
    return a * b;
}
var resultado = multiplicar(x, y);
```

Solución (Oculta para el lector):

```
const x = 5;
const y = 10;

// Función flecha concisa y constante
const multiplicar = (a, b) => a * b;

const resultado = multiplicar(x, y);
console.log(resultado); // Resultado: 50
```

Resumen y Proyecto Integrador Final

Resumen de Conceptos Claves del Nivel Experto

- **`fetch` API** es el estándar para peticiones HTTP, usando `async/await` para un manejo limpio de las Promesas.
- **Web Storage** (`localStorage` y `sessionStorage`) permite la persistencia de datos en el navegador, requiriendo `JSON.stringify` y `JSON.parse` para objetos.

- **Módulos ES6** (`import` / `export`) son esenciales para la organización y el aislamiento del código.
- El manejo de errores se centraliza en el bloque **`try...catch`** y el lanzamiento de excepciones con **`throw new Error()`** .
- Las **Buenas Prácticas** se centran en la inmutabilidad, el uso de `const` y `let` , y la claridad del código.

Proyecto Integrador: To-Do List Interactiva con Almacenamiento Local

El objetivo de este proyecto es aplicar todos los conocimientos adquiridos (DOM, Eventos, Objetos, Arrays, `localStorage`) para construir una aplicación funcional.

Requisitos del Proyecto:

1. **Estructura HTML Mínima:** Un campo de texto (`<input>`), un botón para agregar y una lista (`<ul>`) para mostrar las tareas.
2. **Modelo de Datos:** Cada tarea debe ser un objeto JavaScript (ej. `{ id: 1, texto: "Tarea", completada: false }`).
3. **Persistencia:** Las tareas deben guardarse en `localStorage` y cargarse al iniciar la aplicación.
4. **Funcionalidad:**
    - Permitir agregar nuevas tareas al hacer clic en el botón.
    - Mostrar las tareas en la lista (`<ul>`).
    - Permitir marcar una tarea como completada (ej. al hacer clic en el texto de la tarea).
    - Permitir eliminar una tarea.

Estructura del Código (Guía):

```
// Archivo: app.js

// 1. Constantes y Variables Globales
const inputTarea = document.getElementById('input-tarea');
const botonAgregar = document.getElementById('btn-agregar');
const listaTareas = document.getElementById('lista-tareas');
const CLAVE_STORAGE = 'tareas-js';

let tareas = []; // Array principal de objetos tarea

// 2. Funciones de Almacenamiento
function guardarTareas() {
    localStorage.setItem(CLAVE_STORAGE, JSON.stringify(tareas));
}

function cargarTareas() {
    const tareasString = localStorage.getItem(CLAVE_STORAGE);
    if (tareasString) {
        tareas = JSON.parse(tareasString);
    }
    renderizarTareas(); // Llamar a la función de renderizado
}

// 3. Función de Renderizado (DOM)
function renderizarTareas() {
    listaTareas.innerHTML = ''; // Limpiar la lista antes de renderizar

    tareas.forEach(tarea => {
        const li = document.createElement('li');
        li.textContent = tarea.texto;
        li.dataset.id = tarea.id; // Guardar el ID en el elemento DOM

        // Estilo para tareas completadas
        if (tarea.completada) {
            li.style.textDecoration = 'line-through';
            li.style.color = '#888';
        }

        // Agregar manejador de evento para marcar como completada
        li.addEventListener('click', toggleCompletada);

        // Botón de eliminar
        const botonEliminar = document.createElement('button');
        botonEliminar.textContent = 'X';
        botonEliminar.style.marginLeft = '10px';
```

```
        botonEliminar.addEventListener('click', eliminarTarea);

        li.appendChild(botonEliminar);
        listaTareas.appendChild(li);
    });
}

// 4. Funciones de Lógica
function agregarTarea() {
    const texto = inputTarea.value.trim();
    if (texto === '') return; // No agregar tareas vacías

    const nuevaTarea = {
        id: Date.now(), // ID único basado en el tiempo
        texto: texto,
        completada: false
    };

    tareas = [...tareas, nuevaTarea]; // Inmutabilidad: crear un nuevo array
    guardarTareas();
    renderizarTareas();
    inputTarea.value = ''; // Limpiar input
}

function toggleCompletada(e) {
    // Obtener el ID del elemento <li> clickeado
    const id = Number(e.currentTarget.dataset.id);

    // Usar map para crear un nuevo array inmutablemente
    tareas = tareas.map(tarea => {
        if (tarea.id === id) {
            return { ...tarea, completada: !tarea.completada }; // Invertir
estado
        }
        return tarea;
    });

    guardarTareas();
    renderizarTareas();
}

function eliminarTarea(e) {
    // Detener la propagación para que no se active toggleCompletada del
<li>
    e.stopPropagation();
```

```
    // El ID está en el <li>, que es el padre del botón
    const id = Number(e.currentTarget.parentNode.dataset.id);

    // Usar filter para crear un nuevo array sin la tarea eliminada
    tareas = tareas.filter(tarea => tarea.id !== id);

    guardarTareas();
    renderizarTareas();
}

// 5. Inicialización
botonAgregar.addEventListener('click', agregarTarea);
window.addEventListener('load', cargarTareas); // Cargar al iniciar la
página
```

Apéndice

Soluciones a los Ejercicios Propuestos

- **Nivel Básico - Ejercicio Integrador:**

```
const precioBase = 100;
let descuento = 0.10;


function calcularPrecioFinal(base, porcentajeDescuento) {
    if (porcentajeDescuento > 0) {
        const montoDescuento = base * porcentajeDescuento;
        return base - montoDescuento;
    } else {
        return base;
    }
}


const precioFinal = calcularPrecioFinal(precioBase, descuento);
console.log(`El precio final es: $${precioFinal}`);
```

- **Nivel Intermedio - Ejercicio Integrador:**

```
class Empleado {
    constructor(nombre, edad, salario) {
        this.nombre = nombre;
        this.edad = edad;
        this.salario = salario;
    }
}


const nomina = [
    new Empleado("Juan", 25, 45000),
    new Empleado("María", 35, 60000),
    new Empleado("Pedro", 40, 55000),
    new Empleado("Laura", 22, 40000)
];


const salariosAltos = nomina.filter(empleado => empleado.salario > 50000);
const nombresSalariosAltos = salariosAltos.map(empleado => empleado.nombre);


console.log("Empleados con salarios altos:");
console.log(nombresSalariosAltos);
```

- **Nivel Avanzado - Ejercicio Integrador:**

```
// Asumiendo <div id="resultado"></div>
const resultadoDiv = document.getElementById('resultado');


function simularCarga() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve("¡Contenido cargado!");
        }, 2000);
    });
}


async function cargarContenido() {
    if (resultadoDiv) {
        resultadoDiv.textContent = "Cargando...";
    }


    try {
        const contenido = await simularCarga();


        if (resultadoDiv) {
            resultadoDiv.textContent = contenido;
        }
    } catch (error) {
        if (resultadoDiv) {
            resultadoDiv.textContent = "Error al cargar el
contenido.";
        }
    }
}


cargarContenido();
```

Glosario de Términos

Término	Definición
Vanilla JS	Término para referirse al JavaScript puro, sin librerías ni frameworks.
DOM	Modelo de Objetos del Documento. Interfaz de programación para documentos HTML.
Asincronía	Ejecución de tareas sin bloquear el hilo principal, permitiendo que el programa continúe.
Promesa	Objeto que representa el resultado eventual de una operación asíncrona.
`async/await`	Sintaxis moderna para manejar Promesas de forma más legible.
`fetch` **API**	Interfaz para realizar peticiones de red (HTTP) basadas en Promesas.
`localStorage`	Almacenamiento persistente de datos en el navegador (no expira).
HOF	Función de Orden Superior. Función que recibe o devuelve otra función.
Spread Operator	Operador (`...`) que expande un iterable en sus elementos. Clave para la inmutabilidad.
`try...catch`	Bloque de código para manejar excepciones y errores de forma controlada.
