Tabla de contenidos

1. ¿Qué es Javascript?

Definición

JavaScript es un lenguaje de programación **interpretado, dinámico y de alto nivel** que fue diseñado inicialmente para añadir interactividad a las páginas web. Hoy en día, JavaScript es uno de los lenguajes más populares y versátiles del mundo.

Características Principales

- 🔁 **Interpretado:** No necesita compilación previa.
- 🚀 **Dinámico:** Las variables pueden cambiar de tipo durante la ejecución.
- 🎯 **Orientado a objetos:** Soporta programación orientada a objetos.
- 🔧 **Funcional:** Soporta programación funcional.
- 📱 **Multiplataforma:** Ejecuta en navegadores, servidores, móviles, etc.
- 🌐 **Estándar abierto:** Basado en ECMAScript.

¿Por qué JavaScript?

JavaScript es el lenguaje que da vida a las páginas web. Mientras que HTML proporciona la estructura y CSS el diseño, JavaScript añade **comportamiento e interactividad.**

Imagina una página web como una casa:

- 🏗️ **HTML** es la estructura (paredes, puertas, ventanas).
- 🎨 **CSS** es la decoración (colores, muebles, estilo).
- ⚡ **JavaScript** es la electricidad (luces que se encienden, electrodomésticos que funcionan).

Sin JavaScript, las páginas web serían documentos estáticos como un libro. Con JavaScript, se convierten en **aplicaciones interactivas** que responden a las acciones del usuario.

Ejemplo:

```
<!-- Sin JavaScript: Página estática -->
<button>Hacer clic</button>
<p>Este texto nunca cambia</p>

<!-- Con JavaScript: Página interactiva -->
<button onclick="cambiarTexto()">Hacer clic</button>
<p id="texto">Este texto puede cambiar</p>

<script>
function cambiarTexto() {
    document.getElementById('texto').innerHTML = '¡El texto ha cambiado!';
}
</script>
```

Aplicaciones comunes de JavaScript:

- 🔁 Validación de formularios en tiempo real.
- 📱 Menús dinámicos y navegación interactiva.
- 🛒 Carritos de compra que se actualizan sin recargar la página.
- 🎮 Juegos web y aplicaciones interactivas.
- 📊 Gráficos dinámicos y visualización de datos.

2. Historia y Evolución de JavaScript

Los Primeros Años (1995-2000)

El nacimiento de JavaScript es una historia fascinante. En 1995, **Brendan Eich** creó el lenguaje en solo 10 días en Netscape. Originalmente llamado **"LiveScript"**, se le cambió el nombre por razones de marketing. Es importante recordar: **JavaScript NO es Java.**

- **1995:** Brendan Eich crea JavaScript en Netscape.
- **Propósito inicial:** Añadir interactividad básica a páginas web.
- **Primera versión:** JavaScript 1.0 en Netscape Navigator 2.0.

Guerra de Navegadores (2000-2005)

Una época problemática. Cada navegador implementaba JavaScript de manera diferente (Microsoft creó **JScript**), lo que generaba problemas de compatibilidad y dificultaba el desarrollo.

Estandarización y Era Moderna (2005-Presente)

La llegada de **AJAX** en 2005 y la estandarización con **ECMAScript (ES)** marcaron un antes y un después. **ES5 (2009)** y, sobre todo, **ES6 (2015)** modernizaron el lenguaje, convirtiéndolo en la potente herramienta que es hoy.

- **2009:** ECMAScript 5 (ES5) - Primera gran actualización.
- **2012:** JavaScript en el servidor con Node.js.
- **2015:** ECMAScript 6 (ES6/ES2015) - La revolución moderna.
- **Actualidad:** Actualizaciones anuales y uso extendido (Frontend, Backend, Móvil).

3. JavaScript en el Navegador vs Otros Entornos

JavaScript en el Navegador (Client-Side)

Es su entorno original. Aquí, JavaScript puede:

- ✅ Manipular el DOM (Document Object Model).
- ✅ Responder a eventos del usuario (clicks, teclado).
- ✅ Realizar peticiones HTTP (AJAX/Fetch).
- ✅ Acceder a APIs del navegador (Geolocalización, Cámara).
- ✅ Almacenar datos localmente (localStorage).
- ❌ **No puede** acceder al sistema de archivos por seguridad.

JavaScript en el Servidor (Node.js)

Gracias a **Node.js**, JavaScript puede ejecutarse en el backend. Esto permite:

- ✅ Acceder al sistema de archivos.
- ✅ Conectarse a bases de datos.
- ✅ Crear servidores web y APIs.
- ✅ Ejecutar comandos del sistema.
- ❌ **No puede** manipular el DOM (no existe en el servidor).

Otros Entornos

JavaScript también se utiliza para crear aplicaciones móviles (React Native) y de escritorio (Electron).

4. Incorporación de JavaScript en HTML

Método 1: JavaScript Inline (No recomendado)

El código se escribe directamente en los atributos HTML. Útil para pruebas rápidas, pero mezcla responsabilidades.

```
<button onclick="alert('¡Hola Mundo!')">Saludar</button>
```

Método 2: JavaScript Embebido (Internal)

El código se coloca dentro de una etiqueta `<script>` en el mismo archivo HTML. Mejor que el inline, pero no es reutilizable en otras páginas.

```
<script>
    function saludar() {
        alert('¡Hola desde script embebido!');
    }
</script>
```

Método 3: JavaScript Externo (⭐ RECOMENDADO)

El código se guarda en un archivo `.js` separado y se enlaza desde el HTML. Es la mejor práctica por su organización, reutilización y rendimiento.

```
<!-- En el HTML -->
<script src="js/main.js"></script>
```

5. Ubicación de Scripts en HTML

La posición de la etiqueta `<script>` es crucial.

- **En el `<head>`:** El navegador detiene la carga del HTML para descargar y ejecutar el script. Puede ralentizar la visualización de la página.
- **Antes de cerrar `</body>` (⭐ RECOMENDADO):** El HTML se carga primero, por lo que el usuario ve el contenido rápidamente. El script se ejecuta después, ya con todos los elementos HTML disponibles.
- **Con atributos `defer` y `async`:** Métodos avanzados para optimizar la carga de scripts sin bloquear el renderizado de la página.

6. Recursos Adicionales

- **MDN JavaScript Guide:** La mejor documentación para empezar.
- **JavaScript.info:** Un tutorial moderno y completo.
- **CodePen / JSFiddle:** Plataformas para practicar online.

9. Resumen del Capítulo

Conceptos Clave

- ✅ JavaScript añade **interactividad** a las páginas web.
- ✅ Ha evolucionado enormemente gracias al estándar **ECMAScript**.
- ✅ Puede correr en el **navegador** y en el **servidor** (Node.js).
- ✅ La mejor práctica es usar archivos **.js externos**.
- ✅ Los scripts deben colocarse **antes de cerrar el </body>** para mejor rendimiento.

2. Sintaxis Básica de JavaScript

1. Variables en JavaScript

¿Qué es una Variable?

Una variable es como una caja con una etiqueta donde puedes guardar información para usarla más tarde. El nombre de la etiqueta es el identificador de la variable, y el contenido de la caja es el valor.

Analogía práctica:

📦 Caja con etiqueta "edad" → contiene el número 25

📦 Caja con etiqueta "nombre" → contiene el texto "María"

📦 Caja con etiqueta "esEstudiante" → contiene true/false

El Operador de Asignación `=`

Antes de ver los tipos de variables, es fundamental entender el símbolo `=` que usamos constantemente en JavaScript.

❗ **IMPORTANTE: El símbolo `=` NO significa "es igual a"** como en matemáticas. En programación significa **"asignar"** o **"guardar"**.

```
// El = significa "asignar el valor de la derecha a la variable de la izquierda"
let edad = 25;
//  ↑     ↑   ↑
//  |     |   └── Valor que se guarda
//  |     └────── Operador de asignación ("guardar en")
//  └──────────── Variable donde se guarda

// Se lee: "asignar el valor 25 a la variable edad"
// NO se lee: "edad es igual a 25"
```

var - La Forma Clásica (Evitar en código moderno)

`var` **fue la primera forma de declarar variables** en JavaScript, pero tiene comportamientos confusos que pueden causar errores.

```
var edad = 25;
var edad = 30;  // ¡Permitido! Pero puede ser un error
console.log(edad); // 30
```

let - Declaración Moderna de Variables

`let` **es la forma moderna y recomendada** para variables que van a cambiar de valor.

```
let edad = 25;
edad = 26; // ✅ Permitido

let ciudad = "Madrid";
// let ciudad = "Barcelona"; // ❌ Error: Ya ha sido declarada
```

const - Para Valores Constantes

`const` **es para valores que NO van a cambiar** una vez asignados.

```
const PI = 3.14159;
// PI = 3.14; // ❌ Error: No puedes cambiar una constante
```

2. Tipos de Datos Primitivos

¿Qué son los Tipos de Datos?

Los tipos de datos definen qué clase de información puede almacenar una variable. JavaScript tiene 7 tipos primitivos principales.

- 🔢 **Number**: Caja para números
- 📝 **String**: Caja para texto
- ✅ **Boolean**: Caja para verdadero/falso
- ❓ **Undefined**: Caja vacía sin definir
- 🚫 **Null**: Caja intencionalmente vacía

Number - Números

JavaScript maneja todos los números como un solo tipo, sin distinguir entre enteros y decimales.

String - Cadenas de Texto

Los strings son secuencias de caracteres usados para representar texto.

Boolean - Verdadero o Falso

Los booleans representan valores lógicos: verdadero (`true`) o falso (`false`).

Undefined y Null

`undefined` significa que una variable ha sido declarada pero no se le ha asignado ningún valor.

`null` representa la ausencia intencional de valor.

3. Objetos en JavaScript

¿Qué es un Objeto?

Un objeto es como una caja que puede contener múltiples etiquetas y valores relacionados. Mientras que las variables primitivas guardan UN solo valor, los objetos pueden guardar múltiples propiedades relacionadas.

Analogía práctica:

📦 Objeto: persona = {
  nombre: "María",
  edad: 25,
  esEstudiante: true
}

Crear y Acceder a Propiedades

```
let estudiante = {
    nombre: "Ana",
    edad: 22,
    carrera: "Ingeniería"
};

// 1. Notación de punto (más común)
console.log(estudiante.nombre); // "Ana"

// 2. Notación de corchetes
console.log(estudiante["nombre"]); // "Ana"
```

Modificar, Agregar y Eliminar Propiedades

```
let producto = {
    nombre: "Laptop",
    precio: 800
};

// Modificar
producto.precio = 750;

// Agregar
producto.marca = "Dell";

// Eliminar
delete producto.precio;
```

Métodos en Objetos

Los objetos también pueden contener funciones, que se llaman **métodos**.

```
let calculadora = {
    sumar: function(a, b) {
        return a + b;
    }
};

let resultado = calculadora.sumar(5, 3); // resultado es 8
```

Resumen de la Parte 1

Conceptos Clave

- ✅ **Variables**: `let` para valores que cambian, `const` para constantes.
- ✅ **Tipos Primitivos**: `number`, `string`, `boolean`, `undefined`, `null`.
- ✅ **Objetos**: Para agrupar datos y funcionalidades relacionadas.
- ✅ **Propiedades**: Se acceden con notación de punto (`objeto.propiedad`).
- ✅ **Diferencia Clave**: Los primitivos se copian por valor, los objetos por referencia.

2.1. Operadores en JavaScript

Operadores Aritméticos

Los operadores aritméticos realizan cálculos matemáticos básicos y avanzados.

```
let a = 10;
let b = 3;

console.log(a + b);    // 13 (suma)
console.log(a - b);    // 7  (resta)
console.log(a * b);    // 30 (multiplicación)
console.log(a / b);    // 3.333... (división)
console.log(a % b);    // 1  (módulo/residuo)
console.log(a ** b);   // 1000 (exponenciación: 10³)
```

Operadores de Comparación

Los operadores de comparación comparan valores y devuelven un boolean (true/false).

❗ **MUY IMPORTANTE: Diferencia entre `=`, `==` y `===`**

```
// = (UN SOLO IGUAL) → ASIGNACIÓN (guardar)
let edad = 25;

// == (DOS IGUALES) → COMPARACIÓN DÉBIL (con conversión de tipos)
console.log(25 == "25"); // true

// === (TRES IGUALES) → COMPARACIÓN ESTRICTA (sin conversión)
console.log(25 === "25"); // false
```

Regla de oro: SIEMPRE usa `===` y `!==` para evitar conversiones automáticas y errores inesperados.

Operadores Lógicos

Los operadores lógicos combinan o modifican valores boolean.

```
// AND lógico (&&) - TODOS deben ser verdaderos
let esMayorDeEdad = true;
let tieneLicencia = true;
console.log(esMayorDeEdad && tieneLicencia); // true

// OR lógico (||) - AL MENOS UNO debe ser verdadero
let esFinDeSemana = true;
let esVacaciones = false;
console.log(esFinDeSemana || esVacaciones); // true

// NOT lógico (!) - Invierte el valor
let estaLloviendo = true;
console.log(!estaLloviendo);   // false
```

Operador Ternario (Condicional)

El operador ternario es una forma compacta de escribir if/else simples.

```
// Sintaxis: condición ? valorSiTrue : valorSiFalse
let edad = 18;
let mensaje = edad >= 18 ? "Eres mayor de edad" : "Eres menor de edad";
console.log(mensaje); // "Eres mayor de edad"
```

2. Estructuras de Control Condicionales

if/else - La Base de las Decisiones

Las estructuras if/else permiten que tu programa tome decisiones basadas en condiciones.

```
let puntuacion = 85;

if (puntuacion >= 90) {
    console.log("Excelente - A");
} else if (puntuacion >= 80) {
    console.log("Muy bien - B");
} else {
    console.log("Necesita mejorar");
}
```

switch - Para Múltiples Opciones

`switch` es útil cuando tienes muchas opciones basadas en un valor específico.

```
let diaDeLaSemana = 3;
let nombreDia;

switch (diaDeLaSemana) {
    case 1:
        nombreDia = "Lunes";
        break;
    case 2:
        nombreDia = "Martes";
        break;
    case 3:
        nombreDia = "Miércoles";
        break;
    default:
        nombreDia = "Otro día";
}

console.log(nombreDia); // "Miércoles"
```

3. Ejemplo Integrador: Sistema Simple de Calificaciones

Este ejemplo combina todo lo que hemos aprendido: objetos, operadores y estructuras de control.

```
const NOTA_MINIMA_APROBADO = 60;

let estudiante = {
    nombre: "María González",
    notaFinal: 85,
};

console.log("Estudiante:", estudiante.nombre);
console.log("Nota obtenida:", estudiante.notaFinal);

let mensaje;
if (estudiante.notaFinal >= NOTA_MINIMA_APROBADO) {
    mensaje = "✅ Aprobado";
} else {
    mensaje = "❌ Reprobado";
}
console.log("Estado:", mensaje);
```

4. Errores Comunes y Debugging

Errores Frecuentes

- ❌ **Confundir asignación (`=`) con comparación (`===`)** dentro de un `if`.
- ❌ **Olvidar el `break`** en una estructura `switch`.
- ❌ **Usar comparación débil (`==`)** que puede llevar a resultados inesperados.

Técnicas de Debugging

La herramienta más simple y poderosa para empezar a depurar es `console.log()`. Te permite "espiar" el valor de tus variables en diferentes puntos de tu programa para entender qué está pasando.

```
let numero = 5;
console.log("Valor inicial de numero:", numero);

numero = numero * 2;
console.log("Valor después de multiplicar:", numero);

if (numero > 10) {
    console.log("La condición es verdadera, pero no hacemos nada.");
}

console.log("Valor final de numero:", numero);
```

5. Resumen de la Parte 2

Conceptos Clave

- ✅ **Operadores**: Aritméticos, comparación (`===`), lógicos (`&&`, `||`, `!`), y el ternario.
- ✅ **Estructuras de control**: `if/else` para decisiones complejas y `switch` para múltiples opciones específicas.
- ✅ **Debugging**: Usar `console.log` para verificar el flujo y los valores de las variables.

3. Funciones en JavaScript

1. ¿Qué son las Funciones?

Concepto Fundamental

Una función es como una máquina que realiza una tarea específica. Le das algunos ingredientes (parámetros), ella hace su trabajo, y te devuelve un resultado.

¿Por qué usar Funciones?

1. Evitar repetición de código (DRY - Don't Repeat Yourself)

```
// ✅ Con funciones - código reutilizable
function calcularPrecioConDescuento(precio) {
    let descuento = precio * 0.15;
    let precioFinal = precio - descuento;
    console.log("Precio final:", precioFinal);
    return precioFinal;
}

// Usar la función múltiples veces
calcularPrecioConDescuento(100);  // Precio final: 85
calcularPrecioConDescuento(200);  // Precio final: 170
```

2. Organización y legibilidad

```
// ✅ Código organizado con funciones
function validarEdad(edad) {
    return edad >= 18;
}

function mostrarMensajeBienvenida(nombre) {
    console.log("¡Bienvenido/a " + nombre + "!");
}

function procesarRegistro(nombre, edad) {
    if (validarEdad(edad)) {
        mostrarMensajeBienvenida(nombre);
    } else {
        console.log("Debes ser mayor de edad");
    }
}
```

2. Declaración de Funciones

Sintaxis Básica

```
function nombreDeLaFuncion() {
    // Código que se ejecuta
    console.log("¡Hola desde la función!");
}

// Para usar (llamar) la función
nombreDeLaFuncion();
```

3. Parámetros y Argumentos

¿Qué son los Parámetros?

Los parámetros son variables que recibe la función para trabajar con ellos. Son como los "ingredientes" que necesita la función.

```
// "nombre" es un parámetro
function saludarPersona(nombre) {
    console.log("¡Hola " + nombre + "!");
}

// "María" es un argumento (el valor que pasas)
saludarPersona("María");  // ¡Hola María!
```

Parámetros por Defecto

```
function saludarConIdioma(nombre, idioma = "español") {
    if (idioma === "español") {
        console.log("¡Hola " + nombre + "!");
    } else if (idioma === "inglés") {
        console.log("Hello " + nombre + "!");
    }
}

saludarConIdioma("María");              // ¡Hola María!
saludarConIdioma("John", "inglés");     // Hello John!
```

4. Return - Devolver Valores

¿Qué es Return?

`return` **permite que una función devuelva un resultado** que puedes usar en otras partes de tu código.

```
function calcularSuma(a, b) {
    let resultado = a + b;
    return resultado;  // Devuelve el resultado
}

let suma = calcularSuma(5, 3);
console.log("Suma guardada:", suma);  // 8
```

5. Scope (Alcance de Variables)

¿Qué es el Scope?

El scope determina dónde puedes usar una variable en tu código. Es como las "zonas de acceso" de las variables.

Scope Global vs Scope Local

```
// Variable global
let nombreGlobal = "María";

function mostrarInformacion() {
    // Variable local
    let nombreLocal = "Juan";
    console.log("Dentro de función:", nombreGlobal);  // OK
    console.log("Variable local:", nombreLocal);      // OK
}

mostrarInformacion();
console.log("Fuera de función:", nombreGlobal);      // OK
// console.log(nombreLocal); // ❌ Error: no está definida fuera
```

6. Funciones Anónimas

Las funciones anónimas son funciones sin nombre que se asignan a variables o se usan directamente.

```
let saludarAnonima = function() {
    console.log("¡Hola desde función anónima!");
};

saludarAnonima();
```

7. Arrow Functions (Funciones Flecha)

Las arrow functions son una forma más corta de escribir funciones introducida en ES6. Usan la sintaxis `=>` (flecha).

```
// Arrow function
let saludar = () => {
    console.log("¡Hola!");
};

// Una sola línea (return implícito)
let sumaCorta = (a, b) => a + b;

saludar();
console.log(sumaCorta(10, 20)); // 30
```

8. Resumen del Capítulo

Conceptos Clave

- ✅ **Declaración de funciones:** Sintaxis básica y llamadas.
- ✅ **Parámetros y argumentos:** Cómo pasar datos a las funciones.
- ✅ **Return:** Devolver valores para reutilizar resultados.
- ✅ **Scope:** Alcance de variables globales vs locales.
- ✅ **Funciones anónimas y Arrow functions:** Sintaxis moderna y concisa.

4. Arrays y Métodos en JavaScript

1. ¿Qué son los Arrays?

Concepto Fundamental

Un array es como una lista ordenada donde puedes guardar múltiples valores relacionados. Piensa en él como una **estantería con casillas numeradas**.

Analogía práctica: 📚

```
// ✅ Con arrays - una sola variable con múltiples valores
let estudiantes = ["María", "Juan", "Ana", "Carlos", "Lucía"];

// Fácil de manejar:
console.log("Tenemos", estudiantes.length, "estudiantes");
console.log("El primer estudiante es:", estudiantes[0]);
```

2. Creación y Acceso a Arrays

Formas de Crear Arrays

```
// 1. Forma literal (más común y recomendada)
let frutas = ["manzana", "banana", "naranja"];

// 2. Array vacío (para llenar después)
let listaVacia = [];

// ✅ Recomendación: Usar siempre la forma literal []
```

Acceso a Elementos por Índice

Los arrays en JavaScript empiezan en índice 0.

```
let colores = ["rojo", "verde", "azul", "amarillo"];

// Acceso por índice (empezando en 0)
console.log(colores[0]);  // "rojo"    - primer elemento
console.log(colores[1]);  // "verde"   - segundo elemento
console.log(colores[4]);  // undefined - no existe
```

Modificar Elementos

```
let animales = ["perro", "gato", "pájaro"];

// Cambiar un elemento existente
animales[1] = "conejo";  // Cambia "gato" por "conejo"
console.log("Después:", animales); // ["perro", "conejo", "pájaro"]
```

3. ¿Qué son los Bucles? (Fundamentos para Arrays)

Concepto Fundamental

Un bucle es una forma de **repetir código automáticamente**. Es como dar instrucciones repetitivas: "cuenta del 1 al 10" o "haz esto 5 veces".

Bucle for - "Contar y repetir"

Cuándo usar: Cuando sabes exactamente cuántas veces quieres repetir algo.

```
// Estructura: for (inicialización; condición; incremento)
for (let i = 1; i <= 3; i++) {
    console.log("Número: " + i);
}
// Resultado: Número: 1, Número: 2, Número: 3
```

Bucle while - "Repetir mientras se cumpla condición"

Cuándo usar: Cuando no sabes cuántas veces repetir, pero tienes una condición de parada.

```
// Estructura: while (condición) { ... }
let contador = 0;
while (contador < 3) {
    console.log("Contador: " + contador);
    contador++;
}
// Resultado: Contador: 0, Contador: 1, Contador: 2
```

4. Métodos Fundamentales de Arrays

push() y pop() - Final del Array

`push()` agrega al final. `pop()` elimina del final.

```
let tareas = ["Estudiar"];
tareas.push("Ejercicio"); // ["Estudiar", "Ejercicio"]
let tareaEliminada = tareas.pop(); // "Ejercicio"
```

unshift() y shift() - Inicio del Array

`unshift()` agrega al inicio. `shift()` elimina del inicio.

```
let cola = ["Persona 2"];
cola.unshift("Persona 1"); // ["Persona 1", "Persona 2"]
let personaAtendida = cola.shift(); // "Persona 1"
```

includes() - Verificar Existencia

`includes()` devuelve `true` o `false` si un elemento existe.

```
let tecnologias = ["HTML", "CSS", "JavaScript"];
console.log(tecnologias.includes("JavaScript")); // true
console.log(tecnologias.includes("Python"));   // false
```

5. Introducción a la Iteración sobre Arrays

Iteración es recorrer cada elemento de un array uno por uno, usualmente con un bucle.

```
let frutas = ["manzana", "banana", "naranja"];

console.log("=== INVENTARIO DE FRUTAS ===");
for (let i = 0; i < frutas.length; i++) {
    console.log("Posición " + i + ": " + frutas[i]);
}
```

6. Resumen de la Parte 1

Conceptos Clave

- ✅ **Arrays básicos:** Qué son, cómo crearlos y acceder a elementos.
- ✅ **Bucles fundamentales:** `for` y `while` para repetir código.
- ✅ **Iteración básica:** Recorrer arrays elemento por elemento.
- ✅ **Métodos básicos:** `push`, `pop`, `shift`, `unshift`, `includes`.

4.1. Métodos Avanzados e Iteración

1. Bucle for...of - Iteración Moderna

Después de dominar el bucle `for` tradicional, ahora aprenderemos **`for...of`**, que es **más simple y legible** cuando solo necesitamos los valores.

Comparación: for vs for...of

```
let frutas = ["manzana", "banana", "naranja", "kiwi"];

// ❌ Bucle for tradicional (más verboso)
for (let i = 0; i < frutas.length; i++) {
    console.log("Fruta: " + frutas[i]);
}

// ✅ Bucle for...of (más simple y claro)
for (let fruta of frutas) {
    console.log("Fruta: " + fruta);
}
```

2. Métodos Modernos de Arrays

Los métodos modernos hacen que trabajar con arrays sea **más expresivo y funcional**. Son especialmente útiles para transformar y filtrar datos.

forEach() - Ejecutar Acción para Cada Elemento

`forEach()` es como `for...of` pero en forma de método.

```
let productos = ["Laptop", "Mouse", "Teclado", "Monitor"];

productos.forEach(function(producto) {
    console.log("Producto: " + producto);
});
```

map() - Transformar Array

`map()` crea un **NUEVO** array transformando cada elemento.

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

let duplicados = numeros.map(function(numero) {
    return numero * 2;
});

console.log(duplicados); // [2, 4, 6, 8, 10]
```

filter() - Filtrar Elementos

`filter()` crea un **NUEVO** array con elementos que cumplen una condición.

```
let edades = [15, 22, 17, 35, 19, 28, 16, 45];

let mayoresDeEdad = edades.filter(function(edad) {
    return edad >= 18;
});

console.log(mayoresDeEdad); // [22, 35, 19, 28, 45]
```

3. Combinando Métodos de Arrays

La verdadera potencia viene al **combinar métodos** para operaciones complejas.

```
let empleados = [
    { nombre: "Ana", edad: 28, salario: 3000, departamento: "IT" },
    { nombre: "Bruno", edad: 35, salario: 4500, departamento: "Marketing" },
    { nombre: "Carlos", edad: 24, salario: 2500, departamento: "IT" }
];

let nombresJovenesIT = empleados
    .filter(empleado => empleado.departamento === "IT" && empleado.edad < 30)
    .map(empleado => empleado.nombre.toUpperCase());

console.log(nombresJovenesIT); // ["ANA", "CARLOS"]
```

4. Arrays Bidimensionales (Introducción)

Un **array bidimensional** es un array que contiene otros arrays. Es como una tabla con filas y columnas.

```
let matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

// Acceso: array[fila][columna]
console.log(matriz[1][2]); // 6 (fila 1, columna 2)
```

5. Resumen de la Parte 2

Conceptos Clave

- ✅ **for...of:** Iteración moderna y simplificada.
- ✅ **forEach():** Método para ejecutar acciones en cada elemento.
- ✅ **map():** Transformar arrays sin modificar el original.
- ✅ **filter():** Crear subconjuntos basados en condiciones.
- ✅ **Encadenamiento:** Combinar métodos para operaciones complejas.

4.2. Práctica Integrada

1. Combinando lo que Hemos Aprendido

Ahora que conoces los métodos básicos de arrays, vamos a **combinarlos** para resolver problemas más interesantes.

Ejemplo Práctico: Lista de Tareas

```
let tareas = [];

// Función para agregar tarea
function agregarTarea(descripcion) {
    if (descripcion && descripcion.trim() !== "") {
        tareas.push({
            id: tareas.length + 1,
            descripcion: descripcion.trim(),
            completada: false,
            fecha: new Date().toLocaleDateString()
        });
        console.log("✅ Tarea agregada: " + descripcion);
    } else {
        console.log("❌ La descripción no puede estar vacía");
    }
}

// Función para marcar como completada
function completarTarea(id) {
    let tareaEncontrada = false;
    for (let i = 0; i < tareas.length; i++) {
        if (tareas[i].id === id) {
            tareas[i].completada = true;
            tareaEncontrada = true;
            break;
        }
    }
}

// Función para mostrar todas las tareas
function mostrarTareas() {
    console.log("\n📋 LISTA DE TAREAS:");
    tareas.forEach(function(tarea) {
        let estado = tarea.completada ? "✅" : "⏳";
        console.log(estado + " " + tarea.id + ". " + tarea.descripcion);
    });
}

// Demostración
agregarTarea("Estudiar JavaScript");
agregarTarea("Hacer ejercicio");
completarTarea(1);
mostrarTareas();
```

2. Trabajando con Arrays de Objetos

En la programación real, a menudo trabajamos con **arrays que contienen objetos**. Esto es muy común para representar listas de datos.

Ejemplo: Inventario Simple

```
let inventario = [];

function agregarProducto(nombre, precio, cantidad) {
    if (!nombre || precio <= 0 || cantidad < 0) {
        console.log("❌ Datos inválidos");
        return;
    }
    inventario.push({ nombre, precio, cantidad });
    console.log("✅ Producto agregado: " + nombre);
}

function mostrarInventario() {
    console.log("\n📦 INVENTARIO ACTUAL:");
    inventario.forEach(producto => {
        let total = producto.precio * producto.cantidad;
        console.log(`• ${producto.nombre}: ${producto.cantidad} unidades a $${producto.precio} c/u. Valor: $${total.toFixed(2)}`);
    });
}

function valorTotalInventario() {
    let total = 0;
    inventario.forEach(producto => {
        total += producto.precio * producto.cantidad;
    });
    console.log("💰 Valor total del inventario: $" + total.toFixed(2));
    return total;
}

// Demostración
agregarProducto("Laptop", 800, 5);
agregarProducto("Mouse", 25, 20);
mostrarInventario();
valorTotalInventario();
```

3. Errores Comunes y Cómo Solucionarlos

Error 1: Confundir Índices

```
// ❌ Error común: Pensar que los arrays empiezan en 1
let colores = ["rojo", "verde", "azul"];
console.log("Primer color:", colores[1]);  // ❌ INCORRECTO

// ✅ Correcto: Los arrays empiezan en 0
console.log("Primer color:", colores[0]);   // ✅ "rojo"
```

Error 2: Confundir map() con forEach()

```
let numeros = [1, 2, 3];

// ❌ Error: Usar forEach() esperando un nuevo array
let duplicados = numeros.forEach(n => n * 2);
console.log(duplicados); // undefined

// ✅ Correcto: Usar map() para crear un nuevo array
let duplicadosCorrecto = numeros.map(n => n * 2);
console.log(duplicadosCorrecto); // [2, 4, 6]
```

4. Ejercicios Prácticos

Ejercicio 1: Lista de Tareas

```
let tareas = ["Estudiar JavaScript", "Hacer ejercicio"];

function agregarTarea(nueva) { /*...*/ }
function completarTarea(indice) { /*...*/ }
function mostrarTareas() { /*...*/ }
```

Ejercicio 2: Notas de Estudiantes

```
let estudiantes = [
    { nombre: "Ana", nota: 85 },
    { nombre: "Luis", nota: 72 }
];

function estudiantesAprobados() { /* Filtrar con nota >= 75 */ }
function nombresEstudiantes() { /* Usar map para obtener solo nombres */ }
```

5. Resumen Final

Lo que has Aprendido

- ✅ **Combinación de métodos**: Resolver problemas complejos.
- ✅ **Arrays de objetos**: Trabajar con datos estructurados.
- ✅ **Debugging**: Identificar y corregir errores comunes.

5. Manipulación del DOM

1. ¿Qué es el DOM?

Definición del DOM

DOM (Document Object Model) es la representación que hace el navegador de una página HTML como un árbol de objetos que JavaScript puede manipular.

```
🌐 Página HTML → 🌳 Árbol DOM → 🎮 JavaScript lo controla
```

El Objeto `document`

`document` es el objeto principal que representa toda la página HTML. Es tu punto de entrada para manipular el DOM.

```
console.log(document.title);        // Título de la página
console.log(document.body);         // Elemento <body>
```

2. Selección de Elementos

getElementById() - Buscar por ID

`getElementById()` encuentra UN elemento que tenga el ID especificado. Los IDs deben ser únicos.

```
let elementoMensaje = document.getElementById("mensaje");
```

querySelector() - Selector CSS Potente

`querySelector()` usa selectores CSS para encontrar el primer elemento que coincida.

```
let titulo = document.querySelector(".titulo");
let contenedor = document.querySelector("#contenedor");
```

querySelectorAll() - Seleccionar Múltiples Elementos

`querySelectorAll()` devuelve TODOS los elementos que coincidan con el selector.

```
let todosLosParrafos = document.querySelectorAll("p");
todosLosParrafos.forEach(parrafo => {
    console.log(parrafo.textContent);
});
```

3. Modificación de Contenido

innerHTML - Contenido HTML

`innerHTML` permite leer y modificar el contenido HTML dentro de un elemento.

```
let contenido = document.getElementById("contenido");
contenido.innerHTML = "<h2>¡Nuevo contenido!</h2>";
```

textContent - Solo Texto

`textContent` trabaja solo con texto, ignorando etiquetas HTML. Es más seguro para datos de usuario.

```
let mensaje = document.getElementById("mensaje");
mensaje.textContent = "Este es texto seguro";
```

4. Modificación de Atributos y Clases

setAttribute() y getAttribute()

```
let imagen = document.getElementById("imagen");
imagen.setAttribute("src", "nueva-foto.jpg");
```

classList - API Moderna para Clases

`classList` proporciona métodos para trabajar con clases CSS de manera más fácil.

```
let elemento = document.getElementById("caja");
elemento.classList.add("activo");
elemento.classList.remove("viejo");
elemento.classList.toggle("visible");
```

5. Creación y Manipulación de Elementos

createElement() y appendChild()

`createElement()` crea nuevos elementos que luego puedes insertar en el DOM con **`appendChild()`**.

```
let contenedor = document.getElementById("contenedor");
let nuevoParrafo = document.createElement("p");
nuevoParrafo.textContent = "Párrafo agregado dinámicamente";
contenedor.appendChild(nuevoParrafo);
```

6. Resumen del Capítulo

Conceptos Vistos

- ✅ **DOM**: Comprensión del árbol de objetos que representa HTML.
- ✅ **Selección**: getElementById, querySelector, querySelectorAll.
- ✅ **Contenido**: innerHTML vs textContent.
- ✅ **Atributos y Clases**: setAttribute, classList.
- ✅ **Creación**: createElement, appendChild para elementos dinámicos.

6. Eventos

1. ¿Qué son los Eventos?

Definición de Eventos

Un evento es una acción que ocurre en una página web que JavaScript puede detectar y responder. Los eventos son la base de la interactividad en las aplicaciones web.

```
🖱️ Usuario hace clic → 🔔 Se dispara evento → ⚡ JavaScript responde
```

Tipos Comunes de Eventos

```
// Eventos de mouse: click, dblclick, mouseenter, mouseleave
// Eventos de teclado: keydown, keyup, keypress
// Eventos de formulario: submit, change, input, focus, blur
// Eventos de página: load, resize, scroll
```

2. Event Handlers - La Forma Básica

Event Handlers Inline (en HTML)

Los event handlers inline se escriben directamente en el HTML. Es la forma más simple pero no la más recomendada.

```
<button onclick="alert('¡Hola!')">Saludar</button>
```

Event Handlers desde JavaScript

Puedes asignar event handlers desde JavaScript usando propiedades del elemento.

```
let boton = document.getElementById("mi-boton");
boton.onclick = function() {
    console.log("¡Botón clickeado!");
};
```

3. Event Listeners - La Forma Moderna

addEventListener() - Método Recomendado

`addEventListener()` es la forma moderna y más flexible de manejar eventos. Permite múltiples listeners para el mismo evento.

```
let boton = document.getElementById("mi-boton");

boton.addEventListener("click", function() {
    console.log("¡Listener 1 ejecutado!");
});

boton.addEventListener("click", () => {
    console.log("¡Listener 2 ejecutado!");
});
```

4. El Objeto Event

El objeto `event` contiene información detallada sobre el evento que acaba de ocurrir. Se pasa automáticamente como parámetro a la función del event listener.

```
let boton = document.getElementById("mi-boton");
boton.addEventListener("click", function(event) {
    console.log("Tipo de evento:", event.type);
    console.log("Elemento objetivo:", event.target);
});
```

event.target vs event.currentTarget

`event.target` es el elemento que originó el evento, mientras que `event.currentTarget` es el elemento que tiene el listener.

5. Eventos de Formulario

Los formularios tienen eventos específicos como **`submit`** para el envío o **`input`** para cambios en tiempo real.

```
let formulario = document.getElementById("mi-formulario");
formulario.addEventListener("submit", function(event) {
    event.preventDefault(); // Evita que la página se recargue
    console.log("Formulario enviado");
});
```

6. Remover Event Listeners

Puedes quitar event listeners cuando ya no los necesites para optimizar el rendimiento, usando **`removeEventListener()`**.

```
function miFuncion() {
    console.log("Función ejecutada");
}
let boton = document.getElementById("mi-boton");
boton.addEventListener("click", miFuncion);

// Para remover, necesitas la referencia exacta a la función
boton.removeEventListener("click", miFuncion);
```

7. Resumen del Capítulo

Conceptos Vistos

- ✅ **Eventos:** Comprensión de qué son y cómo funcionan.
- ✅ **Event Handlers:** Inline y desde JavaScript.
- ✅ **Event Listeners:** `addEventListener()` como método moderno.
- ✅ **Objeto Event:** Acceso a información detallada del evento.
- ✅ **Eventos de Formulario:** Validación y manejo de inputs.

7. Depurando el código

1. ¿Qué es el Debugging?

Definición de Debugging

Debugging es el proceso de encontrar y corregir errores (bugs) en tu código. Es una habilidad fundamental para cualquier programador.

```
🐛 Bug en código → 🔍 Investigar → 🔧 Corregir → ✅ Código funcionando
```

Tipos de Errores Comunes

```
// 1. ERRORES DE SINTAXIS - El código no puede ejecutarse
// let nombre = "Juan;  // ❌ Falta comilla de cierre

// 2. ERRORES DE REFERENCIA - Variable no existe
// console.log(variableInexistente);  // ❌ ReferenceError

// 3. ERRORES DE TIPO - Usar algo de forma incorrecta
// let numero = 5;
// numero.toUpperCase();  // ❌ TypeError: números no tienen toUpperCase

// 4. ERRORES LÓGICOS - El código funciona, pero hace algo incorrecto
// function sumar(a, b) {
//     return a - b;  // ❌ Debería ser a + b
// }
```

2. Console.log - Tu Mejor Herramienta

`console.log()` es la herramienta de debugging más simple y efectiva para empezar.

Técnicas Avanzadas de console.log

```
// console.warn - Para advertencias
console.warn("⚠️ Esta función está obsoleta");

// console.error - Para errores
console.error("❌ Error crítico encontrado");

// console.table para objetos y arrays
let estudiantes = [
    { nombre: "Ana", nota: 85 },
    { nombre: "Luis", nota: 92 },
];
console.table(estudiantes);
```

3. Herramientas de Desarrollador

Las herramientas de desarrollador están integradas en todos los navegadores modernos. Se abren generalmente con la tecla **F12**.

4. Breakpoints Básicos

Un breakpoint es un punto de parada donde el navegador pausará la ejecución del código para que puedas inspeccionarlo paso a paso.

```
function miFuncion() {
    let variable1 = "valor inicial";
    debugger; // ⏸️ Pausa aquí automáticamente
    let variable2 = variable1.toUpperCase();
    return variable2;
}
```

5. Resumen del Capítulo

Conceptos Vistos

- ✅ **Console.log:** Herramienta básica y técnicas avanzadas.
- ✅ **Dev Tools:** Navegación y uso del panel de consola.
- ✅ **Breakpoints:** Establecimiento y uso efectivo.
- ✅ **Debugging Sistemático:** Metodología para encontrar y corregir errores.