Learning.Ebooks

MUESTRA GRATUITA

JAVASCRIPT MODERNO

ES2024 · async/await · Módulos · Proyectos reales

ES2024 | Async/Await | Módulos | Closures | DOM | APIs

- ✓ 10 capítulos progresivos desde cero
- ✓ +50 ejemplos de código comentados
- ✓ 4 proyectos reales listos para tu portafolio
- ✓ Patrones y buenas prácticas de la industria

De variables a arquitectura profesional

Incluye proyectos: API Fetcher · Todo App · Clima · Gestor de Tareas

JAVASCRIPT MODERNO

Muestra gratuita — Capítulos 00, 01 y 02

ES2024 · Async/Await · Módulos · DOM · APIs · Proyectos Reales

Publicado por **Learning.Ebooks**

Esta muestra gratuita incluye los primeros 3 capítulos. El libro completo está disponible en **leanpub.com/javascript-moderno**

Versión 1.0 — 2026

CONTENIDO DE ESTA MUESTRA

EN EL LIBRO COMPLETO

JAVASCRIPT MODERNO

ES2024 · Async/Await · Módulos · DOM · APIs · Proyectos Reales

Publicado por **Learning.Ebooks**

Versión 1.0 — 2026

TABLA DE CONTENIDOS

CAPÍTULO 0

Introducción

¿Por qué aprender JavaScript moderno en 2026?

00

¿Por qué JavaScript moderno?

JavaScript es el único lenguaje que corre de forma nativa en todos los navegadores del mundo. Hoy también domina el backend con Node.js, las apps mobile con React Native, las apps de escritorio con Electron y hasta la inteligencia artificial con TensorFlow.js. Aprenderlo bien no es una opción — es una ventaja competitiva enorme.

CONSEJO
Según el Stack Overflow Developer Survey 2024, JavaScript es el lenguaje más usado por 12 años consecutivos. Dominarlo abre puertas en frontend, backend y mobile al mismo tiempo.

Qué cubre este libro

Este ebook te lleva de los fundamentos hasta patrones profesionales:

- El sistema de tipos de JS y cómo evitar sus trampas
- Funciones avanzadas, closures y el scope
- Programación asincrónica con Promises y async/await
- El sistema de módulos ES6+ para proyectos escalables
- Manipulación del DOM y manejo de eventos de forma eficiente
- Consumo de APIs REST con Fetch API
- Las últimas novedades de ES2020 a ES2024
- 4 proyectos reales para tu portafolio

NOTA
Cada capítulo incluye ejemplos de código reales, ejercicios prácticos y un proyecto que podés usar en producción.

Herramientas que vas a necesitar

setup.sh

```
// Opción 1: Ejecutar en el navegador
// Abrí las DevTools con F12 → Consola

// Opción 2: Node.js (recomendado)
// Descargalo en nodejs.org
node --version   // Verificar instalación
node script.js   // Ejecutar archivo

// Opción 3: Online
// codesandbox.io, stackblitz.com, replit.com
```

RESUMEN DEL CAPÍTULO

- ✓ JavaScript es el lenguaje más utilizado del mundo (12 años consecutivos)
- ✓ ES2024 trajo mejoras importantes al lenguaje que usaremos en este libro
- ✓ Necesitás un editor (VSCode) y Node.js — ambos son gratuitos
- ✓ Cada capítulo tiene ejemplos, ejercicios y un proyecto práctico

CAPÍTULO 1

Variables, Tipos y Operadores

let, const, var, tipos primitivos y coerción de tipos

01

Variables: let, const y var

Antes de ES6, la única forma de declarar variables en JavaScript era con **var**. ES6 introdujo **let** y **const** que solucionan los problemas históricos de var. Entender la diferencia es fundamental.

variables.js

```
// VAR — solo usarlo en código legado
var nombre = "Ana";
var nombre = "Luis";  // ✓ No da error (reasignación de var)
console.log(nombre);  // "Luis"

// LET — para variables que cambian
let edad = 25;
edad = 26;            // ✓ OK — se puede reasignar
// let edad = 27;     // ✗ Error — no se puede redeclarar

// CONST — para valores que no cambian
const PI = 3.14159;
// PI = 3;            // ✗ TypeError: Assignment to constant variable

// CONST con objetos — el objeto puede mutar
const user = { nombre: "Ana", edad: 25 };
user.edad = 26;       // ✓ OK — mutamos la propiedad
// user = {};         // ✗ Error — no podemos reasignar la variable
```

■■ ATENCIÓN

var tiene function scope y sufre de hoisting. Esto causó miles de bugs históricos. Usar siempre let/const en código moderno.

Tipos primitivos

Tipo	Ejemplo	typeof	Descripción
`string`	"Hola"	string	Texto
`number`	42 / 3.14	number	Números (enteros y decimales)
`boolean`	true / false	boolean	Verdadero o falso
`undefined`	undefined	undefined	Variable sin valor asignado
`null`	null	object	Ausencia intencional de valor
`bigint`	9007199254740993n	bigint	Números enteros grandes
`symbol`	Symbol('id')	symbol	Identificador único

Coerción de tipos

JavaScript convierte tipos automáticamente en algunas operaciones. Esto se llama **coerción implícita** y es una fuente común de bugs:

coercion.js

```
// Coerción implícita — comportamientos sorprendentes
console.log("5" + 3);      // "53" (string concatenation)
console.log("5" - 3);      // 2    (numeric operation)
console.log(true + 1);     // 2    (true → 1)
console.log(false + 1);    // 1    (false → 0)
console.log(null + 1);     // 1    (null → 0)
console.log(undefined+1); // NaN (no es un número)

// Coerción explícita — siempre preferible
Number("42");      // 42
String(42);        // "42"
Boolean(0);        // false
Boolean("hola");   // true
parseInt("42px");  // 42
parseFloat("3.14abc"); // 3.14
```

Igualdad: == vs ===

equality.js

```
// == (igualdad débil) — hace coerción de tipos
console.log(0 == false);    // true  ← peligroso
console.log("" == false);   // true  ← peligroso
console.log(null == undefined); // true

// === (igualdad estricta) — siempre usar esto
console.log(0 === false);   // false ← correcto
console.log("" === false);  // false ← correcto
console.log(1 === "1");     // false ← correcto
```

✓ **BUENA PRÁCTICA**

Usar siempre === (triple igual). Solo usar == cuando explícitamente necesitás comparar null con undefined.

Operadores modernos

operators.js

```
// Nullish coalescing (??) — ES2020
const port = process.env.PORT ?? 3000;  // 3000 si PORT es null/undefined
const config = null ?? { debug: false }; // { debug: false }

// Logical assignment — ES2021
let a = null;
a ??= "default";   // a = "default" (solo si es null/undefined)
let b = 0;
b ||= 10;          // b = 10 (si es falsy)
let c = 1;
c &&= c * 2;       // c = 2 (si es truthy)

// Optional chaining (?.) — ES2020
const user = { profile: { avatar: "img.png" } };
console.log(user?.profile?.avatar);   // "img.png"
console.log(user?.address?.city);     // undefined (sin error)
console.log(user?.getAge?.());        // undefined si no existe
```

RESUMEN DEL CAPÍTULO

- ✓ Usar const por defecto, let cuando el valor cambia, nunca var
- ✓ JavaScript tiene 7 tipos primitivos — entender undefined vs null es clave
- ✓ La coerción implícita es fuente de bugs: siempre usar === y conversiones explícitas
- ✓ ?? y ?. (ES2020) simplifican el manejo de valores nulos enormemente

✏ **EJERCICIO**

EJERCICIO: Creá una función que reciba un objeto usuario con campos opcionales (nombre, edad, ciudad) y retorne un string formateado. Usá optional chaining y nullish coalescing para manejar los campos faltantes.

CAPÍTULO 2

Funciones y Scope

Arrow functions, closures, IIFE, hoisting y buenas prácticas

02

Declaración de funciones

JavaScript tiene múltiples formas de definir funciones, cada una con comportamiento diferente. Elegir la forma correcta impacta el scope, el hoisting y el valor de **this**.

funciones.js

```
// 1. Function declaration — tiene hoisting
saluda("Ana");  // ✓ Funciona antes de la declaración
function saluda(nombre) {
    return `Hola, ${nombre}!`;
}

// 2. Function expression — NO tiene hoisting
// saluda2("Ana");  // ✗ ReferenceError
const saluda2 = function(nombre) {
    return `Hola, ${nombre}!`;
};

// 3. Arrow function — ES6, no tiene su propio 'this'
const saluda3 = (nombre) => `Hola, ${nombre}!`;
const doble = n => n * 2;              // parámetro único sin paréntesis
const sumar = (a, b) => a + b;         // retorno implícito
const getUser = () => ({ id: 1 });     // retornar objeto con paréntesis
```

Parámetros avanzados

params.js

```
// Valores por defecto — ES6
function crearUser(nombre, rol = "user", activo = true) {
    return { nombre, rol, activo };
}
crearUser("Ana");                  // { nombre: "Ana", rol: "user", activo: true }
crearUser("Carlos", "admin");      // { nombre: "Carlos", rol: "admin", activo: true }

// Rest parameters — agrupa argumentos sobrantes
function sumar(primero, ...resto) {
    return primero + resto.reduce((acc, n) => acc + n, 0);
}
sumar(1, 2, 3, 4);   // 10

// Destructuring en parámetros
function mostrarUser({ nombre, edad, ciudad = "No especificada" }) {
    return `${nombre}, ${edad} años — ${ciudad}`;
}
mostrarUser({ nombre: "Ana", edad: 25 }); // "Ana, 25 años — No especificada"
```

Closures — uno de los conceptos más importantes de JS

Un closure ocurre cuando una función 'recuerda' el scope donde fue creada, incluso después de que ese scope haya terminado de ejecutarse. Los closures son la base de módulos, memoización y muchos patrones avanzados.

closures.js

```
// Ejemplo clásico de closure
function crearContador(inicio = 0) {
    let cuenta = inicio;  // variable privada al closure
    return {
        incrementar: () => ++cuenta,
        decrementar: () => --cuenta,
        reset: () => { cuenta = inicio; },
        valor: () => cuenta,
    };
}
const contador = crearContador(10);
contador.incrementar();  // 11
contador.incrementar();  // 12
contador.decrementar();  // 11
contador.valor();        // 11
// No podemos acceder a 'cuenta' directamente — está encapsulada

// Closure para memoización
function memoize(fn) {
    const cache = new Map();
    return function(...args) {
        const key = JSON.stringify(args);
        if (cache.has(key)) return cache.get(key);
        const result = fn.apply(this, args);
        cache.set(key, result);
        return result;
    };
}
const fibMemo = memoize(function fib(n) {
    return n <= 1 ? n : fibMemo(n-1) + fibMemo(n-2);
});
```

■ CONSEJO

Los closures son la base del patrón módulo, de los hooks de React y de muchos patrones funcionales. Invertir tiempo en entenderlos bien vale la pena.

IIFE y funciones auto-invocadas

iife.js

```
// IIFE — Immediately Invoked Function Expression
(function() {
    const secreto = "Solo visible aquí";
    console.log("IIFE ejecutado");
})();

// Arrow IIFE
const resultado = (() => {
    const datos = [1, 2, 3, 4, 5];
    return datos.reduce((a, b) => a + b, 0);
})();
console.log(resultado);  // 15

// Uso moderno: inicialización asíncrona
(async () => {
    const datos = await fetch("/api/datos").then(r => r.json());
    console.log(datos);
})();
```

Higher-Order Functions

hof.js

```
// Una función que recibe o retorna funciones
function aplicar(valor, ...funciones) {
    return funciones.reduce((v, fn) => fn(v), valor);
}

const doblar    = x => x * 2;
const sumarDiez = x => x + 10;
const alCuadrado = x => x ** 2;

aplicar(3, doblar, sumarDiez, alCuadrado);  // ((3*2)+10)^2 = 256

// Currying — funciones que retornan funciones
const multiplicar = (a) => (b) => a * b;
const triplicar = multiplicar(3);
const cuadruplicar = multiplicar(4);

triplicar(5);     // 15
cuadruplicar(7);  // 28
```

RESUMEN DEL CAPÍTULO

- ✓ Arrow functions no tienen this propio — ideales para callbacks y métodos cortos
- ✓ Los closures son el mecanismo de encapsulación nativo de JavaScript
- ✓ Los parámetros por defecto y rest parameters hacen el código más robusto
- ✓ Las Higher-Order Functions son la base de map, filter, reduce y de React

JS

¿Te gustó la muestra?

El libro completo incluye 13 capítulos más:

- ✓ Clases, herencia y mixins (Cap. 04)
- ✓ Async/Await y Promises en profundidad (Cap. 05)
- ✓ Módulos ES6+ y patrón barrel (Cap. 06)
- ✓ DOM, eventos y delegación (Cap. 07)
- ✓ Fetch API y cliente HTTP propio (Cap. 08)
- ✓ ES2020–ES2024 — todas las novedades (Cap. 09)
- ✓ Patrones de diseño: Observer, Factory, Strategy (Cap. 11)
- ✓ Performance, debugging y testing con Jest (Cap. 12)
- ✓ Iteradores, generadores y Symbols (Cap. 13)
- ✓ Web APIs modernas: Workers, IntersectionObserver (Cap. 15)
- ✓ PROYECTO FINAL: Gestor de Tareas completo (Cap. 10)
- ✓ PROYECTO BONUS: App del Clima con geolocalización

Conseguilo por solo $9

Precio de lanzamiento — sube a $25 pronto

leanpub.com/javascript-moderno