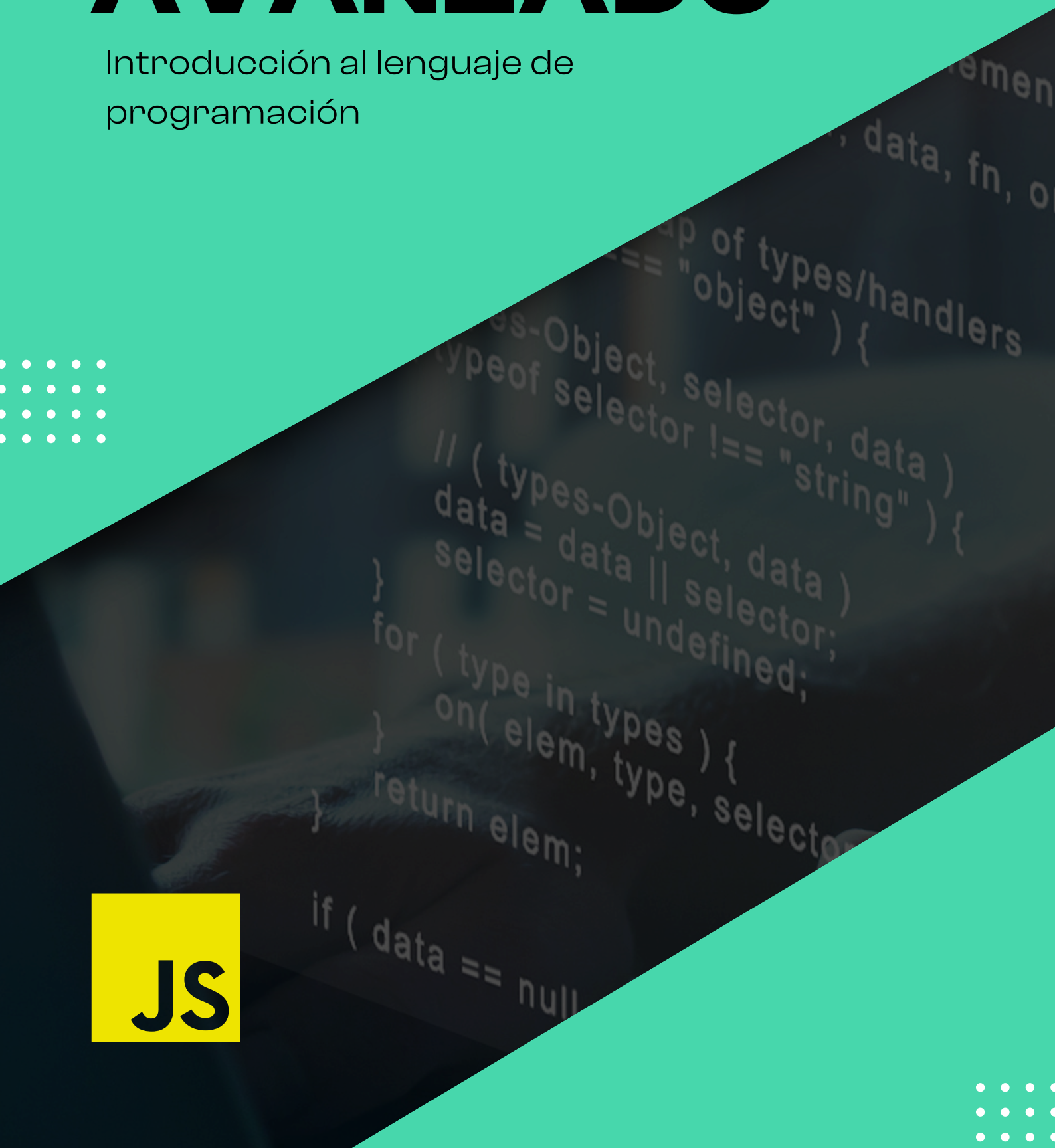
CODE SPACE

JAVASCRIPT AVANZADO

Introducción al lenguaje de programación



JS

JavaScript Avanzado

Funciones y programación funcional

La **programación funcional** es un paradigma de programación basado en el uso de **funciones puras y datos inmutables**. Su objetivo es escribir código más **predecible, reutilizable y fácil de testear.**

En lugar de enfocarse en cómo hacer las cosas paso a paso (como en la programación imperativa), se enfoca en qué se quiere obtener, delegando el cómo a funciones que se combinan y transforman datos.

Características clave

1- **Funciones puras**
   Una función pura:
   - Siempre devuelve el mismo resultado con los mismos argumentos.
   - No tiene efectos secundarios (no modifica variables externar, no escribe en consola, no hace peticiones HTTP, etc.).

```
function suma(a, b) {
  return a + b;
}
```

2- **Evitar efectos secundarios**
   Los efectos secundarios hacen el código menos predecible. En programación funcional, se busca aislarlos o evitarlos.
   Ejemplos de efectos secundarios:
   - Modificar una variable global.
   - Escribir en la consola.
   - Leer/escribir archivos o realizar llamadas a APIs.

3- **Inmutabilidad**

   No se modifican datos, sino que se crean copias con los cambios deseados.

```
const persona = { nombre: "Ana", edad: 25};

const nuevaPersona = {...persona, edad: 26}; // sin mutar el original
```

4- **Funciones de orden superior**
   Son funciones que:
   - Aceptan otras funciones como argumentos.
   - Devuelven funciones como resultado.

```
function aplicar(fn, valor) {
  return fn(valor);
}

function duplicar(x) {
  return x * 2;
}

console.log(aplicar(duplicar, 5)); // 10
```

5- **Composición de funciones**
   Componer funciones significa encadenarlas, de forma que la salida de una sea la entrada de otra.

```
const aMayusculas = x => x.toUpperCase();
const agregarExclamacion = x => x + "!";

const exclamativo = x => agregarExclamacion(aMayusculas(x));

console.log(exclamativo("hola")); // "HOLA!"
```

6- **Transparencia referencial**
   Una expresión tiene transparencia referencial si puede reemplazarse por su valor sin cambiar el comportamiento del programa.

```
const resultado = suma(2, 3); // puede sustituirse por 5 en cualquier lugar
```

Closures

Un closure (o clausura) es una función que **"recuerda" el entorno léxico en el que fue creada**, incluso después de que ese entorno haya terminado su ejecución.

En términos simples:

- Cuando defines una **función dentro de otra función**, la función interna tiene acceso a:
  - Sus propias variables.

- Las variables de la función externa.
- Las variables globales.

```
function crearSaludo(nombre) {
  return function(saludo) {
    console.log(`${saludo}, ${nombre}`);
  };
}

const saludarAna = crearSaludo("Ana");
saludarAna("Hola"); // "Hola, Ana"
saludarAna("Buenos días"); // "Buenos días, Ana"
```

Aquí, *crearSaludo( )* devuelve una función. Esa función recuerda el valor de nombre, aunque *crearSaludo( )* ya haya terminado de ejecutarse.

Estás usando un closure si:

- Tienes una función dentro de otra.
- La función interna usa variables de la externa.
- Esa función interna se sigue ejecutando después de que la externa ya terminó.

JavaScript Avanzado

Clases y Programación Orientada a Objetos (POO)

La POO es un paradigma basado en objetos que encapsulan datos y comportamiento. JavaScript, aunque históricamente fue un lenguaje basado en prototipos desde ES6 incluye una sintaxis basada en clases que facilita la escritura de código más estructurado.

Declaración de Clases

Las clases se definen con la palabra clave *class*. El constructor es un método especial que se ejecuta automáticamente al crear una nueva instancia.

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

const juan = new Persona("Juan", 30);
```

El constructor se llama al hacer *new Persona(...)*. *this* hace referencia a la instancia que se está creando.

Métodos de Instancia y Estáticos

- Métodos de instancia: son funciones disponibles en cada objeto creado con la clase.
- Métodos estáticos: se definen con *static* y se llaman desde la clase, no desde la instancia.

```
class Calculadora {
  suma(a, b) {
    return a + b;
  }

  static version() {
    return "1.0.0";
  }
}

const calc = new Calculadora();
calc.suma(2, 3);
Calculadora.version();
```

this en Clases

Dentro de una clase, *this* hace referencia a la instancia. Pero cuidado: en funciones flecha, *this* no cambia de contexto, mientras que en funciones normales si lo hace.

```
class Boton {
  constructor(nombre) {
    this.nombre = nombre;
    document.addEventListener("click", () => {
      console.log(this.nombre); // this se mantiene
    })
  }
}
```

Getters y Setters

Permiten definir propiedades personalizadas que se calculan o controlan al acceder o modificar un valor.

```
class Rectangulo {
  constructor(ancho, alto) {
    this.ancho = ancho;
    this.alto = alto;
  }

  get area() {
    return this.ancho * this.alto;
  }

  set ancho(valor) {
    if (valor > 0) this._ancho = valor;
  }

  get ancho() {
    return this._ancho;
  }
}
```

Se usan como propiedades: *rect.area*, no como métodos. Permiten control de acceso o validación.

Herencia (extends, super)

Una clase puede heredar de otra usando *extends*. El constructor de la subclase debe llamar a *super()* para inicializar la clase padre.

```
class Animal {
  constructor(nombre) {
    this.nombre = nombre;
  }

  hablar() {
    console.log(`${this.nombre} hace un ruido.`);
  }
}

class Perro extends Animal {
  hablar() {
    console.log(`${this.nombre} ladra.`);
  }
}

const timmy = new Perro("Timmy");
timmy.hablar(); // "Timmy ladra."
```

Instanceof y Comprobación de Tipos

Instanceof permite comprobar si un objeto es instancia de una clase concreta.

```
timmy instanceof Perro; // true
timmy instanceof Animal; // true
```

También se puede comprobar con *constructor.name*, aunque es más frágil.

```
timmy.constructor.name === "Perro"; // true
```

Encapsulado con #privateFields

Desde ES2022, JavaScript permite definir campos privados con #. No se pueden acceder desde fuera de la clase.

```
class Cuenta {
  #saldo = 0;

  constructor(inicial) {
    this.#saldo = inicial;
  }

  getSaldo() {
    return this.#saldo;
  }
}

const cuenta = new Cuenta(100);
cuenta.getSaldo(); // 100
cuenta.#saldo; //  Error: campo privado
```

Composición vs. Herencia

Herencia implica una jerarquía de clases.

Composición consiste en combinar objetos con funcionalidades específicas.

```
// Composición
const volador = {
  volar() {
    console.log('Estoy volando');
  }
};

const nadador = {
  nadar() {
    console.log('Estoy nadando');
  }
};

function crearPato(nombre) {
  return {
    nombre,
    ...volador,
    ...nadador
  };
}

const pato = crearPato('Donald');
pato.volar(); // Estoy volando
pato.nadar(); // Estoy nadando
```

JavaScript Avanzado

Más Sobre Objetos

Prototipos y herencia prototípica

Todos los objetos en JavaScript tienen una cadena de prototipos (prototype chain). Es la base del sistema de herencia en JS.

```
const animal = {
  hacerSonido() {
    console.log('Sonido genérico');
  }
};

const perro = Object.create(animal);
perro.ladrar = function () {
  console.log('Guau');
};

perro.hacerSonido(); // Sonido genérico
perro.ladrar();      // Guau
```

- *Object.create(proto)* crea un objeto nuevo con el prototipo indicado.
- Si no encuentra una propiedad en el objeto, la busca en su prototipo, y así sucesivamente.

```
console.log(Object.getPrototypeOf(perro) === animal); // true
```

Object.assign()

Copia propiedades de uno o más objetos a otro. Muy útil para combinar objetos o hacer “shallow copies”.

```
const destino = { a: 1 };
const fuente = { b: 2, c: 3 };

Object.assign(destino, fuente);

console.log(destino); // { a: 1, b: 2, c: 3 }
```

Object.seal()

Sella un objeto: no permite añadir ni eliminar propiedades, pero sí modificar las existentes.

```
const usuario = {
  nombre: 'Ana',
  edad: 25
};

Object.seal(usuario);

usuario.nombre = 'Laura'; // ✅ permitido
usuario.nuevoCampo = 'test'; // no se añade
delete usuario.edad; // no se elimina

console.log(usuario); // { nombre: 'Laura', edad: 25 }
console.log(Object.isSealed(usuario)); // true
```

Si quieres un objeto totalmente inmutable, usa *Object.freeze()*.

hasOwnProperty()

Verifica si un objeto tiene una propiedad directamente en sí mismo, no heredada.

```
const persona = {
  nombre: 'Luis'
};

console.log(persona.hasOwnProperty('nombre')); // true
console.log(persona.hasOwnProperty('toString')); // false (heredado del prototipo)
```

JavaScript Avanzado

Más Sobre Arrays y Estructuras de Datos

Set y WeakSet

- **Set:** Una colección de valores únicos, sin duplicados.

```
const numeros = new Set([1, 2, 2, 3]);
numeros.add(4);
numeros.delete(2);

console.log(numeros); // Set(3) {1, 3, 4}
console.log(numeros.has(3)); // true
```

→ Iterables.
→ No permite duplicados.
→ Métodos: add, delete, has, clear, size.

- **WeakSet:**

→ Solo almacena objetos.
→ No es iterable.
→ Los objetos pueden ser recolectados por el **garbage collector** si no hay más referencias.

```
const ws = new WeakSet();
const obj = {};

ws.add(obj);
console.log(ws.has(obj)); // true
```

Map y WeakMap

- **Map:** Colección de pares clave-valor. A diferencia de los objetos, las claves pueden ser de cualquier tipo (incluidos objetos o funciones).

```
const mapa = new Map();
mapa.set('clave', 123);
mapa.set({ x: 1 }, 'objeto');

console.log(mapa.get('clave')); // 123
console.log(mapa.size); // 2
```

→ Métodos: set, get, has, delete, clear.

→ Itera en orden de inserción.

```
for (const [k, v] of mapa) {
  console.log(k, v);
}
```

- **WeakMap:**
  → Las claves deben ser objetos.
  → No es iterable.
  → Claves eliminadas si no hay más referencias, ayuda al **garbage collector**

Spread operator (...) y rest operator

Ambos usan **...** , pero según el contexto su significado cambia.

- **Spread:** descompone arrays u objetos.

```
const arr = [1, 2, 3];
const copia = [...arr]; // copia superficial

const obj = { a: 1, b: 2 };
const extendido = { ...obj, c: 3 };
```

- **Rest:** recoge múltiples valores.

```
function sumar(...numeros) {
  return numeros.reduce((acc, n) => acc + n, 0);
}

sumar(1, 2, 3); // 6
```

También se usa en desestructuración:

```
const [primero, ...resto] = [1, 2, 3, 4];
// primero = 1, resto = [2, 3, 4]
```

Flattening arrays: flat( ) y flatMap( )

- **Flat( ):** aplana arrays anidados.

```
const arr = [1, [2, [3, 4]]];
console.log(arr.flat());        // [1, 2, [3, 4]]
console.log(arr.flat(2));       // [1, 2, 3, 4]
```

- **FlatMap( ):** mapea y aplana a un nivel.

```
const palabras = ['hola', 'mundo'];
const resultado = palabras.flatMap(palabra => palabra.split(''));

console.log(resultado); // ['h', 'o', 'l', 'a', 'm', 'u', 'n', 'd', 'o']
```

JavaScript Avanzado

Garbage Collector

El garbage collector (GC) es el mecanismo automático de JavaScript encargado de liberar memoria que ya no se necesita, para que la aplicación no consuma recursos innecesarios.

JavaScript gestiona la memoria automáticamente, no tienes que liberar memoria manualmente como en otros lenguajes.

¿Cómo decide el GC qué borrar?

Principio Clave: accesibilidad.

Si un valor ya no es accesible desde ningún punto del programa, se considera “basura” y puede ser eliminado.

```
let persona = {
  nombre: 'Ada'
};

persona = null; // El objeto original ya no es accesible: se puede recolectar.
```

¿Qué estructuras de memoria hay?

JavaScript guarda los valores en dos lugares:

- Stack (pila): valores primitivos y referencias.
- Heap (montón): objetos, arrays, funciones…lo que ocupa más espacio.

El GC trabaja principalmente en el heap, buscando valores que no tengan referencias vivas.

Algoritmo principal: Mark and Sweep (marcar y barrer)

Este es el algoritmo más utilizado:

1. Marcar todos los valores accesibles directamente (por ejemplo, desde variables globales o el call stack).

2. Desde esos accesibles, seguir referencias y marcar los objetos que se puedan alcanzar.
3. Todo lo que no esté marcado se elimina.

```
function crearUsuario() {
  let user = {
    nombre: 'Cristina',
    datos: {
      edad: 30
    }
  };
  return user;
}

let u = crearUsuario();
// Aquí, el objeto está referenciado → NO se borra

u = null;
// Ahora ya no hay referencia a ese objeto → será recolectado
```

Retención de memoria (Memory leaks)

A veces se mantienen referencias innecesarias, y el GC no puede limpiar. Esto se llama fuga de memoria.

Casos comunes:

- Variables globales sin limpiar.
- Closures mal gestionados.
- Event listeners no eliminados.
- Timers (setInterval, setTimeOut) activos innecesariamente.
- Referencias en estructuras como Map, Set que no se borran.

```
let lista = [];

function guardarReferencia() {
  let obj = { nombre: 'fantasma' };
  lista.push(obj); // obj nunca se libera si lista vive para siempre
}
```

Herramientas del navegador para devs

Los navegadores modernos tienen herramientas para inspeccionar la memoria:

- Chrome DevTools - pestaña memory.

- Puedes hacer snapshots de memoria y ver qué objetos están vivos.
- Puedes buscar retained objects y referencias no liberadas.

Buenas prácticas para evitar problemas

- Evita variables globales innecesarias.
- Elimina event listeners (removeEventListeners) cuando ya no los uses.
- Limpia setInterval y setTimeout con clearInterval/clearTimeout.
- Usa WeakMap/WeakSet cuando guardes datos temporales ligados a objetos.
- Ten cuidado con arrays u objetos que acumulen datos (como cachés).

JavaScript Avanzado

Manejo de Errores

Los errores pueden gestionarse de forma controlada mediante bloques *try/catch*, lanzando errores personalizados con *throw* y creando clases propias de errores.

Try/catch

Permite capturar y manejar errores que ocurren durante la ejecución de un bloque de código.

```
try {
  // Código que puede lanzar un error
  let resultado = 10 / 0;
  console.log(resultado);
} catch (error) {
  console.error('Ocurrió un error:', error.message);
}
```

El código dentro del *catch* solo se ejecuta si hay un error.

- *Try/catch* solo atrapa errores en tiempo de ejecución síncronos.
- No captura errores en funciones *async* si no usas *await* o *catch*.

```
try {
  setTimeout(() => {
    throw new Error('Error en setTimeout');
  }, 1000);
} catch (e) {
  console.log('¿Lo veremos?'); // ¡No lo captura!
}
```

Throw

Lanza errores manualmente. Puedes lanzar un error cuando detectes una situación anómala en tu código.

```
function dividir(a, b) {
  if (b === 0) {
    throw new Error('No se puede dividir entre cero');
  }
  return a / b;
}

try {
  dividir(10, 0);
} catch (e) {
  console.error(e.message); // No se puede dividir entre cero
}
```

Errores personalizados

Puedes crear tus propias clases de error para representar situaciones específicas.

```
class UsuarioInvalidoError extends Error {
  constructor(mensaje) {
    super(mensaje);
    this.name = 'UsuarioInvalidoError';
  }
}

function crearUsuario(nombre) {
  if (!nombre) {
    throw new UsuarioInvalidoError('El nombre es obligatorio');
  }
  return { nombre };
}

try {
  crearUsuario('');
} catch (e) {
  if (e instanceof UsuarioInvalidoError) {
    console.error(`Error del usuario: ${e.message}`);
  } else {
    console.error('Otro error:', e.message);
  }
}
```

Buenas prácticas

- Crea errores personalizados con clases cuando sea útil diferenciarlos.
- No abuses del *try/catch*, úsalo donde tenga sentido controlar un error.
- No ocultes errores sin hacer nada en el *catch*.
- Usa *finally* si necesitas ejecutar algo sí o sí (como cerrar conexiones, limpiar datos, etc.)

JavaScript Avanzado

Programación asíncrona

JavaScript es un lenguaje **monohilo y no bloqueante**. La programación asíncrona permite realizar operaciones (como peticiones a una API o temporizadores) sin bloquear el flujo principal de ejecución.

Conceptos clave

- Asincronía: permite ejecutar código "en segundo plano" mientras el resto del programa sigue corriendo.

```
console.log('1');
setTimeout(() => console.log('2'), 0);
console.log('3');
// Salida: 1 → 3 → 2
```

- Call Stack (pila de llamadas): Es donde JavaScript guarda las funciones que están siendo ejecutadas. Se comporta como una pila (LIFO – last in, first out).

```
function saludar() {
  console.log('Hola');
}

saludar(); // → La función entra y sale del stack
```

- Event Loop: Es el mecanismo que permite a JavaScript gestionar tareas asíncronas.
    - Call Stack: ejecuta el código principal.
    - Task queue (macrotask): setTimeout, setInterval, eventos del DOM.
    - Microtasks: promises, queueMicrotask

  El event loop va comprobando si el stack está vacío y en ese momento inyecta tareas desde las colas.

Macrotask vs Microtask

```
console.log('Inicio');

setTimeout(() => {
  console.log('Macrotask');
}, 0);

Promise.resolve().then(() => {
  console.log('Microtask');
});

console.log('Fin');

// Salida:
// Inicio
// Fin
// Microtask
// Macrotask
```

Las microtareas tienen prioridad sobre las macrotareas. Siempre se ejecutan antes si ambas están en cola.

Técnicas asíncronas

- Callbacks: una función pasada como argumento, que se ejecuta cuando se completa una tarea.

```
function cargarDatos(callback) {
  setTimeout(() => {
    callback('Datos cargados');
  }, 1000);
}

cargarDatos((resultado) => {
  console.log(resultado);
});
```

Evita esto:

```
hacerAlgo(() => {
  hacerOtraCosa(() => {
    hacerMas(() => {
      // callback hell
    });
  });
});
```

- Promesas (promise): objeto que representa una operación asíncrona con tres posibles estados (pending, fulfilled o rejected).

```
const promesa = new Promise((resolve, reject) => {
  const exito = true;
  if (exito) {
    resolve('Todo bien');
  } else {
    reject('Algo fue mal');
  }
});

promesa
  .then((resultado) => console.log(resultado))
  .catch((error) => console.error(error));
```

- Async/await: syntactic sugar sobre promesas. Permite escribir código asíncrono como si fuera síncrono.

```
function espera(ms) {
  return new Promise((res) => setTimeout(res, ms));
}

async function ejemplo() {
  console.log('Empieza');
  await espera(1000);
  console.log('Después de 1s');
}

ejemplo();
```

Await solo se puede usar dentro de funciones *async*.

Buenas prácticas

- Prefiere *async/await* para código legible.
- Usa *try/catch* para capturar errores en *await*.
- No mezcles callbacks y promesas si puedes evitarlo.
- Usa *Promise.All( )* o *Promise.race( )* si tienes varias tareas concurrentes.

JavaScript Avanzado

JavaScript y Seguridad

JavaScript es una herramienta poderosa para crear aplicaciones web dinámicas, pero su uso incorrecto o poco cuidadoso puede abrir la puerta a vulnerabilidades graves. Algunas de las más comunes son los ataques de XSS, inyección de código, CSRF y problemas relacionados con CORS.

Prevención de XSS (Cross-Site Scripting) e Inyecciones de Código

XSS ocurre cuando un atacante logra inyectar código JavaScript malicioso en una página web vista por otros usuarios. Esto puede permitirles robar cookies, secuestras sesiones, redirigir usuarios o mostrar contenido fraudulento.

Tipos de XSS:

- Reflejado (Reflected): el script malicioso viene como parte de una URL y se ejecuta cuando el servidor lo refleja en la respuesta.
- Almacenado (Stored): el código malicioso se guarda en la base de datos o en un almacenamiento persistente y se ejecuta cuando otro usuario accede.
- DOM-based: el ataque ocurre directamente en el navegador al manipular el DOM con datos no sanitizados.

Ejemplo de código vulnerable:

```
const search = location.search;
document.getElementById("result").innerHTML = "Buscaste: " + search;
```

Si alguien accede a ?q=<script>alert('XSS')</script>, el script se ejecutará.

Prevención:

- No usar innerHTML con datos no confiables. Usa textContent o librerías como DOMPurify.
- Sanitizar la entrada del usuario: elimina o escapa caracteres como >, <, “, ‘, etc.
- Usar frameworks que escapen HTML por defecto: React, Vue, Angular.

Inyecciones de código (Command/Code Injection)

Aunque más comunes en el backend (como inyecciones SQL o de comandos), también pueden darse en frontend cuando evalúas código dinámico (con eval, Function, setTimeout, etc.).

```
eval(userInput); // ¡Nunca hagas esto!
```

Prevención:

- Evita funciones como eval(), function(), setTimeout(string) con datos externos.
- Usa estructuras de control normales y sanitización estricta.
- Valida siempre los datos del usuario, incluso en el cliente.

CSRF (Cross-Site Request Forgery)

CSRF es un ataque donde el navegador del usuario realiza una petición no autorizada (como cambiar la contraseña, hacer una compra, etc.) a un sitio en el que el usuario está autenticado.

Por ejemplo:

1. El usuario está logueado en banco.com.
2. Visita malicioso
3. .com, que carga un formulario oculto que hace un POST a banco.com/transferencia.
4. El navegador incluye automáticamente las cookies de sesión, la acción se realiza sin que el usuario lo sepa.

Prevención:

- Tokens CSRF: el servidor genera un token único por sesión o petición, y el frontend lo incluye en cada formulario o petición POST. Si el token no coincide, se rechaza la petición.
- SameSite Cookies: configurar cookies con SameSite=Strict o SameSite=Lax ayuda a prevenir que se envíen desde sitios externos.
- Cabeceras personalizadas: muchas APIS exigen una cabecera como X-Requested-With, que no puede ser enviada por formularios externos simples.

CORS (Cross-Origin Resource Sharing)

Por seguridad, los navegadores restingen las peticiones HTTP entre orígenes distintos. Por ejemplo, desde https://midominio.com no se puede hacer una petición fetch directamente a https://otrodominio.com/api si ese servidor no lo permite.

¿Qué permite CORS?

CORS es un sistema de permisos que define qué orígenes pueden acceder a recursos en otro servidor. Si el servidor no incluye una cabecera Access-Control-Allow-Origin: https://midominio.com, el navegador bloqueará la respuesta.

No confundas CORS con seguridad del lado del cliente:

- CORS no protege tu servidor de ataques. Solo regula qué frontend puede acceder al backend.
- Las restricciones CORS se aplican en el navegador, no en el servidor.
- No confíes en CORS como única medida de seguridad. Siempre debes validar y autenticar cada petición en el servidor.

JavaScript Avanzado

Automatización y Optimización en JavaScript

Cuando una aplicación JavaScript crece, es fundamental automatizar tareas y optimizar el código para que sea más rápido, ligero y fácil de mantener. En este contexto, entran en juego conceptos como **tree shaking**, **code splitting**, **minificación**, así como herramientas como **Webpack** y **Babel**.

Tree Shaking

Tree Shaking es una técnica para eliminar el código que no se usa (código "muerto") durante el proceso de construcción (build). Esto permite generar bundles más pequeños.

¿Cómo funciona?

Funciona gracias a los módulos ES6 **(import/export)**. Herramientas como Webpack o Rollup pueden detectar qué funciones, clases o variables no se usan y eliminarlas del archivo final.

```
// utils.js
export function sumar(a, b) { return a + b }
export function restar(a, b) { return a - b }

// main.js
import { sumar } from './utils.js';
sumar(2, 3);
```

Si solo usamos **sumar**, restas se elimina automáticamente en producción gracias a tree shaking.

Code Splitting (División de Código)

Code Splitting divide tu aplicación en varios archivos JavaScript más pequeños **(chunks)**, en lugar de un solo archivo gigante. Así, el navegador solo carga lo que necesita cuando lo necesita **(lazy loading)**.

```
button.addEventListener('click', () => {
  import('./moduloPesado.js').then((modulo) => {
    modulo.ejecutar();
  });
});
```

Este **import()** solo carga moduloPesado.js cuando se pulsa el botón, no antes.

Minificación y Uglificación

Es el proceso de eliminar espacios, saltos de línea, comentarios y nombres innecesarios del código para reducir su tamaño sin cambiar su comportamiento.

Además de minificar, la uglificación renombra variables y funciones con nombres muy cortos (como a, b, c) para ahorrar más espacio.

```
function saludo(nombre) {
  console.log("Hola " + nombre);
}

function a(b){console.log("Hola "+b);}
```

Estas técnicas son muy útiles para la versión en producción, pero hacen que el código sea ilegible, por eso se aplica solo al final del proceso de desarrollo.

Webpack, Babel y herramientas modernas

Webpack:

Es un empaquetador de módulos (module bundler). Permite:

- Unir muchos archivos JS, CSS, imágenes, etc. en un solo bundle optimizado.
- Aplicar loaders y plugins para transformar, optimizar y servir tu código.
- Hacer tree shaking, code splitting, minificación y mucho más.

```
// webpack.config.js
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: __dirname + '/dist',
  },
  mode: 'production',
};
```

Babel:

Es un transpilador de JavaScript. Convierte código moderno (ES6+) en código compatible con navegadores antiguos.

```
// Código moderno
const saludar = () => console.log("Hola");

// Babel lo convierte a:
var saludar = function () {
  return console.log("Hola");
};
```

Se integra con Webpack mediante un babel-loader.

Herramientas modernas adicionales

- Vite → Alternativa a Webpack, mucho más rápida.
- Esbuild → Muy rápido, ideal para desarrollo y bundling.
- Rollup → Excelente para librerías, muy bueno con tree shaking.
- Parcel → Cero configuraciones, ideal para proyectos rápidos.

JavaScript Avanzado

Herramientas útiles en JavaScript

JSON.stringify y JSON.parse

- **JSON.stringify(objeto)** → convierte un objeto JavaScript en una cadena de texto JSON.
- **JSON.parse(string)** → convierte una cadena JSON válida en un objeto JavaScript.

¿Para qué sirven?

- Enviar datos al servidor (API REST).
- Guardar objetos en localstorage.
- Clonar objetos simples.
- Depurar estructuras complejas.

```
const usuario = {
  nombre: "Cristina",
  edad: 30,
};

// Convertir a JSON (enviar al servidor, guardar en localStorage…)
const texto = JSON.stringify(usuario);
// → '{"nombre":"Cristina","edad":30}'

console.log(typeof texto); // string

// Convertir de nuevo a objeto JS
const obj = JSON.parse(texto);
console.log(obj.nombre); // "Cristina"
```

Cosas importantes:

- Solo serializa datos válidos en JSON: no funciones, ni fechas, ni undefined.
- Si el objeto tiene referencias circulares (obj.a = obj), JSON.stringify fallará.

Operador ?? (Nullish Coalescing)

Este operador devuelve el primer valor que NO sea **null** ni **undefined**.

```
const nombre = usuario.nombre ?? "Invitado";
```

Es muy útil para poner valores por defecto sin que falsy como 0, false 0 “” lo activen por error.

```
let edad = 0;

let resultado = edad ?? 18;
console.log(resultado); // 0 ✅ (no se activa el 18 porque 0 no es null ni undefined)

let nombre = null;
console.log(nombre ?? "Anónimo"); // "Anónimo"
```

Comparado con **||**:

```
console.log(0 || 18);     // 18 ❌ (porque 0 es falsy)
console.log(0 ?? 18);     // 0 ✅
```

Operador ?. (Optional Chaining)

Permite acceder a propiedades de forma segura, sin lanzar errores si alguna parte del camino es **null** o **undefined**.

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

console.log(persona.direccion?.ciudad); // "Madrid"
console.log(persona.trabajo?.empresa);  // undefined (no lanza error)
```

También funciona con llamada a funciones:

```
persona.saludar?.(); // Solo se ejecuta si `saludar` existe
```

O acceso a arrays:

```
const lista = null;
const primero = lista?.[0]; // undefined, sin error
```