Apuntes examen javascript

Índice

Tema 2, Control de flujo

Estructuras de Control Condicionales

Estructura	Descripción	Sintaxis/Ejemplo
`if`	Ejecuta un bloque de código si la condición es verdadera.	`js if (x > 0) { console.log("Positivo"); }`
`if...else`	Ejecuta un bloque de código si la condición es verdadera; otro bloque si es falsa.	`js if (x > 0) { console.log("Positivo"); } else { console.log("No positivo"); }`
`else if`	Agrega condiciones adicionales en una cadena de decisiones.	`js if (x > 0) { console.log("Positivo"); } else if (x < 0) { console.log("Negativo"); } else { console.log("Cero"); }`
`switch`	Evalúa una expresión y ejecuta el bloque correspondiente a su valor.	`js switch (color) { case "rojo": console.log("Stop"); break; default: console.log("Sigue"); }`
Operador Ternario (`? :`)	Simplifica una condición con dos resultados.	`js const mensaje = x > 0 ? "Positivo" : "No positivo"; console.log(mensaje);`

Estructuras de Control de Bucles

Estructura	Descripción	Sintaxis/Ejemplo
`for`	Itera un número determinado de veces, usando un contador.	`js for (let i = 0; i < 5; i++) { console.log(i); }`
`for...of`	Itera sobre elementos de un objeto iterable (como arrays o strings).	`js for (const item of [1, 2, 3]) { console.log(item); }`
`for...in`	Itera sobre las propiedades enumerables de un objeto.	`js for (const key in {a: 1, b: 2}) { console.log(key); }`
`while`	Repite mientras la condición sea verdadera.	`js let i = 0; while (i < 5) { console.log(i); i++; }`
`do...while`	Similar a `while`, pero asegura que el bloque se ejecute al menos una vez.	`js let i = 0; do { console.log(i); i++; } while (i < 5);`

Control de Flujo en Bucles

Estructura	Descripción	Sintaxis/Ejemplo
`break`	Termina un bucle o `switch` antes de que complete todas las iteraciones.	`js for (let i = 0; i < 5; i++) { if (i === 3) break; console.log(i); }`
`continue`	Salta a la siguiente iteración del bucle.	`js for (let i = 0; i < 5; i++) { if (i === 3) continue; console.log(i); }`

Estructuras de Control con Excepciones

Estructura	Descripción	Sintaxis/Ejemplo

`try...catch`	Captura errores que ocurren en el bloque `try`.	`js try { let x = a; } catch (error) { console.log("Error: ", error.message); }`
`try...catch...finally`	Permite ejecutar un bloque de código al final, independientemente de si hubo errores.	`js try { let x = a; } catch (error) { console.log("Error"); } finally { console.log("Terminado"); }`
`throw`	Lanza una excepción personalizada.	`js throw new Error("Algo salió mal");`

Estructuras Asíncronas

Estructura	Descripción	Sintaxis/Ejemplo
`async...await`	Permite escribir código asíncrono que parece sincrónico.	`js async function fetchData() { const data = await fetch(url); return data.json(); }`
`Promise`	Representa una operación que se resolverá en el futuro.	`js new Promise((resolve, reject) => { if (success) resolve("Éxito"); else reject("Error"); });`

Ejemplo Completo con Varias Estructuras de Control

```
// Diccionario con datos de estudiantes
const estudiantes = [
    { nombre: "Ana", nota: 90 },
    { nombre: "Luis", nota: 75 },
    { nombre: "Juan", nota: 80 },
];

// Evaluación de estudiantes
for (const estudiante of estudiantes) {
    const resultado = estudiante.nota >= 80
        ? "Aprobado"
        : "Reprobado";
```

```
    console.log(`${estudiante.nombre} está ${resultado}.`);
}

// Agregar estudiante con entrada manual
const nuevoEstudiante = { nombre: "Carlos", nota: 85 };
estudiantes.push(nuevoEstudiante);

// Filtrar aprobados
const aprobados = estudiantes.filter(est => est.nota >= 80);
console.log("Estudiantes aprobados:", aprobados);

// Buscar un estudiante específico
const buscar = "Ana";
const encontrado = estudiantes.find(est => est.nombre === buscar);
if (encontrado) {
    console.log(`${buscar} fue encontrado con nota
${encontrado.nota}.`);
} else {
    console.log(`${buscar} no está en la lista.`);
}

// Manejo de errores
try {
    const promedio = estudiantes.reduce((acc, est) => acc +
est.nota, 0) / estudiantes.length;
    if (isNaN(promedio)) {
        throw new Error("Error al calcular promedio.");
    }
    console.log("Promedio de notas:", promedio);
} catch (error) {
    console.log("Error:", error.message);
} finally {
    console.log("Evaluación terminada.");
}
```

Métodos Iterativos de Arrays en JavaScript

Lista completa de los métodos más comunes para recorrer o manipular arrays, incluyendo `forEach` y similares, con sus descripciones y ejemplos.

Método	Descripción	Sintaxis/Ejemplo
`forEach`	Ejecuta una función para cada elemento del array.	`js [1, 2, 3].forEach((num) => console.log(num)); // Imprime 1, 2, 3`
`map`	Crea un nuevo array con los resultados de aplicar una función a cada elemento.	`js const dobles = [1, 2, 3].map((num) => num * 2); console.log(dobles); // [2, 4, 6]`
`filter`	Crea un nuevo array con los elementos que cumplen una condición.	`js const pares = [1, 2, 3, 4].filter((num) => num % 2 === 0); console.log(pares); // [2, 4]`
`reduce`	Aplica una función acumulativa a los elementos del array para reducirlo a un solo valor.	`js const suma = [1, 2, 3].reduce((acc, num) => acc + num, 0); console.log(suma); // 6`
`reduceRight`	Similar a `reduce`, pero recorre el array desde el final hacia el principio.	`js const concat = ['a', 'b', 'c'].reduceRight((acc, char) => acc + char, ''); console.log(concat); // "cba"`
`some`	Devuelve `true` si al menos un elemento cumple una condición.	`js const hayNegativos = [1, -2, 3].some((num) => num < 0); console.log(hayNegativos); // true`
`every`	Devuelve `true` si todos los elementos cumplen una condición.	`js const todosPositivos = [1, 2, 3].every((num) => num > 0); console.log(todosPositivos); // true`
`find`	Devuelve el primer elemento que cumple una condición.	`js const primeroPar = [1, 2, 3].find((num) => num % 2 === 0); console.log(primeroPar); // 2`
`findIndex`	Devuelve el índice del primer elemento que cumple una condición.	`js const indicePar = [1, 2, 3].findIndex((num) => num % 2 === 0); console.log(indicePar); // 1`
`flat`	Aplana un array de arrays en uno de una sola dimensión.	`js const plano = [1, [2, 3], [4, [5]]].flat(2); console.log(plano); // [1, 2, 3, 4, 5]`

flatMap	Aplica un `map` y luego aplana el resultado en un único array.	`js const resultado = [1, 2, 3].flatMap((num) => [num, num * 2]); console.log(resultado); // [1, 2, 2, 4, 3, 6]`
includes	Verifica si un array contiene un valor específico.	`js const existe = [1, 2, 3].includes(2); console.log(existe); // true`
indexOf	Devuelve el índice de la primera aparición de un valor.	`js const indice = [1, 2, 3].indexOf(2); console.log(indice); // 1`
lastIndexOf	Devuelve el índice de la última aparición de un valor.	`js const ultimoIndice = [1, 2, 3, 2].lastIndexOf(2); console.log(ultimoIndice); // 3`
sort	Ordena los elementos del array (modifica el array original).	`js const numeros = [3, 1, 2]; numeros.sort((a, b) => a - b); console.log(numeros); // [1, 2, 3]`
reverse	Invierte el orden de los elementos del array (modifica el array original).	`js const invertido = [1, 2, 3].reverse(); console.log(invertido); // [3, 2, 1]`

Ejemplo Completo con Métodos Iterativos

```
const frutas = ['manzana', 'banana', 'cereza', 'banana'];

// forEach
frutas.forEach((fruta, index) => console.log(`${index}: ${fruta}`));

// map
const longitudes = frutas.map((fruta) => fruta.length);
console.log("Longitudes:", longitudes);

// filter
const sinBananas = frutas.filter((fruta) => fruta !== 'banana');
console.log("Frutas sin banana:", sinBananas);

// reduce
```

```
const totalLetras = frutas.reduce((acumulado, fruta) => acumulado +
fruta.length, 0);
console.log("Total de letras:", totalLetras);

// some
const tieneCereza = frutas.some((fruta) => fruta === 'cereza');
console.log("¿Hay cereza?", tieneCereza);

// every
const todasSonLargas = frutas.every((fruta) => fruta.length > 3);
console.log("¿Todas tienen más de 3 letras?", todasSonLargas);

// find
const primeraBanana = frutas.find((fruta) => fruta === 'banana');
console.log("Primera banana encontrada:", primeraBanana);

// sort
const frutasOrdenadas = [...frutas].sort();
console.log("Frutas ordenadas:", frutasOrdenadas);
```

Tema 3, Estructura de datos

Tabla de Conceptos de Estructuras de Datos

Estructura	Descripción	Ejemplo
Array	Colección ordenada de elementos, accesibles por índices numéricos.	`const numeros = [1, 2, 3]; console.log(numeros[0]); // 1`
Objeto (Diccionario)	Colección de pares clave-valor, donde las claves son únicas y los valores pueden ser de cualquier tipo.	`const persona = { nombre: 'Ana', edad: 25 }; console.log(persona.nombre); // Ana`
Set (Conjunto)	Colección de valores únicos, no ordenados.	`const conjunto = new Set([1, 2, 2, 3]); console.log(conjunto); // Set { 1, 2, 3 }`

Map (Mapa)	Colección de pares clave-valor que preserva el orden de inserción y permite claves de cualquier tipo.	`const mapa = new Map([['clave1', 'valor1'], ['clave2', 'valor2']]); console.log(mapa.get('clave1')); // valor1`
WeakMap	Similar a un Map, pero las claves deben ser objetos y las referencias son débiles (no impiden GC).	`const objeto = {}; const weakMap = new WeakMap([[objeto, 'valor']]); console.log(weakMap.get(objeto)); // valor`
WeakSet	Similar a un Set, pero solo admite objetos como valores y utiliza referencias débiles.	`const objeto = {}; const weakSet = new WeakSet([objeto]); console.log(weakSet.has(objeto)); // true`

Operaciones Comunes

Operación	Array	Objeto (Diccionario)	Set	Map
Agregar elementos	`arr.push(valor)`	`objeto['clave'] = valor`	`set.add(valor)`	`map.set(clave, valor)`
Acceder a elementos	`arr[indice]`	`objeto['clave']`	No aplica (iterar)	`map.get(clave)`
Eliminar elementos	`arr.splice(indice, 1)`	`delete objeto['clave']`	`set.delete(valor)`	`map.delete(clave)`
Verificar existencia	`arr.includes(valor)`	`'clave' in objeto`	`set.has(valor)`	`map.has(clave)`
Iterar elementos	`arr.forEach()` o `for...of`	`for...in` o `Object.entries()`	`set.forEach()` o `for...of`	`map.forEach()` o `for...of`

Método	Descripción	Ejemplo
`push(valor)`	Agrega uno o más elementos al final del array.	`arr.push(4);`

`pop()`	Elimina y devuelve el último elemento del array.	`arr.pop();`
`shift()`	Elimina y devuelve el primer elemento del array.	`arr.shift();`
`unshift(valor)`	Agrega uno o más elementos al inicio del array.	`arr.unshift(0);`
`splice(start, count)`	Elimina, reemplaza o inserta elementos en el array.	`arr.splice(1, 1, 'nuevo');`
`slice(start, end)`	Devuelve una copia de una porción del array, sin modificar el original.	`arr.slice(1, 3);`
`concat(array)`	Combina dos o más arrays en uno nuevo.	`arr.concat([4, 5]);`
`forEach(callback)`	Ejecuta una función para cada elemento del array.	`arr.forEach(val => console.log(val));`
`map(callback)`	Crea un nuevo array con los resultados de la función aplicada a cada elemento.	`arr.map(val => val * 2);`
`filter(callback)`	Crea un nuevo array con los elementos que cumplen la condición del callback.	`arr.filter(val => val > 2);`
`find(callback)`	Devuelve el primer elemento que cumple la condición del callback.	`arr.find(val => val > 2);`
`findIndex(callback)`	Devuelve el índice del primer elemento que cumple la condición del callback.	`arr.findIndex(val => val > 2);`
`reduce(callback, acc)`	Reduce el array a un solo valor mediante una función acumulativa.	`arr.reduce((sum, val) => sum + val, 0);`
`every(callback)`	Devuelve `true` si todos los elementos cumplen la condición del callback.	`arr.every(val => val > 0);`
`some(callback)`	Devuelve `true` si al menos un elemento cumple la condición del callback.	`arr.some(val => val > 2);`

`includes(valor)`	Verifica si el array contiene un valor específico.	`arr.includes(3);`
`indexOf(valor)`	Devuelve el índice de la primera aparición de un valor, o `-1` si no está.	`arr.indexOf(2);`
`lastIndexOf(valor)`	Devuelve el índice de la última aparición de un valor, o `-1` si no está.	`arr.lastIndexOf(2);`
`join(separador)`	Combina los elementos en una cadena, separándolos por el separador especificado.	`arr.join(', ');`
`reverse()`	Invierte el orden de los elementos en el array.	`arr.reverse();`
`sort(callback)`	Ordena los elementos del array.	`arr.sort((a, b) => a - b);`
`flat(profundidad)`	Aplana arrays anidados hasta la profundidad especificada.	`arr.flat(2);`
`flatMap(callback)`	Aplica una función a cada elemento y aplana el resultado en un nivel.	`arr.flatMap(val => [val, val * 2]);`
`keys()`	Devuelve un iterador con los índices del array.	`for (let i of arr.keys()) { console.log(i); }`
`values()`	Devuelve un iterador con los valores del array.	`for (let v of arr.values()) { console.log(v); }`
`entries()`	Devuelve un iterador con pares `[índice, valor]` del array.	`for (let [i, v] of arr.entries()) { ... }`

Métodos de Objetos (Diccionarios)

Método/Propiedad	Descripción	Ejemplo
`Object.keys(obj)`	Devuelve un array con las claves del objeto.	`Object.keys(obj); // ['a', 'b']`

`Object.values(obj)`	Devuelve un array con los valores del objeto.	`Object.values(obj); // [1, 2]`
`Object.entries(obj)`	Devuelve un array de pares `[clave, valor]`.	`Object.entries(obj); // [['a', 1], ...]`
`Object.assign(obj1, obj2)`	Copia las propiedades de un objeto a otro.	`Object.assign({}, obj);`
`Object.freeze(obj)`	Congela un objeto, impidiendo modificaciones.	`Object.freeze(obj);`
`Object.seal(obj)`	Sella un objeto, permitiendo solo cambios en propiedades existentes.	`Object.seal(obj);`
`hasOwnProperty(prop)`	Verifica si el objeto tiene una propiedad específica.	`obj.hasOwnProperty('a');`

Métodos de Sets

Método	Descripción	Ejemplo
`add(valor)`	Agrega un valor al conjunto.	`set.add(1);`
`delete(valor)`	Elimina un valor del conjunto.	`set.delete(1);`
`has(valor)`	Verifica si un valor está en el conjunto.	`set.has(1);`
`clear()`	Elimina todos los valores del conjunto.	`set.clear();`
`forEach(callback)`	Itera sobre los valores del conjunto, ejecutando una función para cada valor.	`set.forEach(val => console.log(val));`
`values()`	Devuelve un iterador con los valores del conjunto.	`for (let v of set.values()) { ... }`

Métodos de Maps

Método	Descripción	Ejemplo

`set(clave, valor)`	Agrega un par clave-valor al mapa.	`map.set('a', 1);`
`get(clave)`	Obtiene el valor asociado a una clave.	`map.get('a');`
`delete(clave)`	Elimina un par clave-valor del mapa.	`map.delete('a');`
`has(clave)`	Verifica si una clave está en el mapa.	`map.has('a');`
`clear()`	Elimina todos los pares clave-valor del mapa.	`map.clear();`
`forEach(callback)`	Itera sobre los pares clave-valor, ejecutando una función para cada uno.	`map.forEach((v, k) => console.log(k));`
`keys()`	Devuelve un iterador con todas las claves del mapa.	`map.keys();`
`values()`	Devuelve un iterador con todos los valores del mapa.	`map.values();`
`entries()`	Devuelve un iterador con pares `[clave, valor]`.	`map.entries();`

```
// 1. Usando Arrays para almacenar productos
const productos = [
    { id: 1, nombre: 'Manzana', categoria: 'Frutas', precio: 0.5 },
    { id: 2, nombre: 'Pan', categoria: 'Panadería', precio: 1 },
    { id: 3, nombre: 'Leche', categoria: 'Lácteos', precio: 1.5 }
];

// 2. Usando un Objeto como Diccionario para la categoría
const categorias = {
    Frutas: 'Alimentos frescos',
    Panadería: 'Productos horneados',
    Lácteos: 'Productos derivados de leche'
};

// 3. Usando un Set para verificar duplicados
const idsUnicos = new Set();
productos.forEach(producto => {
    if (idsUnicos.has(producto.id)) {
        console.log(`Duplicado encontrado: ${producto.nombre}`);
    } else {
        idsUnicos.add(producto.id);
    }
});
```

```
// 4. Usando un Map para un inventario con cantidades
const inventario = new Map();
productos.forEach(producto => {
    inventario.set(producto.nombre, { cantidad: 10, precio: producto.precio });
});

// Operaciones sobre el inventario
// Agregar un producto
inventario.set('Queso', { cantidad: 5, precio: 2.5 });

// Verificar existencia
if (inventario.has('Manzana')) {
    console.log('Manzana está en el inventario.');
}

// Actualizar cantidades
if (inventario.has('Pan')) {
    const pan = inventario.get('Pan');
    pan.cantidad += 5;
    inventario.set('Pan', pan);
}

// Calcular el valor total del inventario
let valorTotal = 0;
inventario.forEach(({ cantidad, precio }) => {
    valorTotal += cantidad * precio;
});
console.log(`Valor total del inventario: $${valorTotal.toFixed(2)}`);

// 5. Usando WeakSet para registrar objetos temporales
const objetosTemporales = new WeakSet();
const productoTemporal = { id: 99, nombre: 'Sandía', categoria: 'Frutas' };
objetosTemporales.add(productoTemporal);
console.log(objetosTemporales.has(productoTemporal)); // true

// Limpieza del objeto temporal
// Una vez que el objeto no tiene referencias, WeakSet lo elimina automáticamente.
```

Tema 4, Funciones

Tipos de Funciones en JavaScript

Tipo	Descripción	Ejemplo

Declarativas	Funciones definidas con la palabra clave `function`. Pueden ser llamadas antes de su declaración debido al hoisting.	`function saludar() { console.log("Hola"); }`
Expresivas	Funciones asignadas a una variable. No tienen hoisting; deben definirse antes de ser llamadas.	`const sumar = function(a, b) { return a + b; };`
Flecha (Arrow)	Funciones concisas con una sintaxis moderna, introducidas en ES6. No tienen su propio `this`.	`const multiplicar = (a, b) => a * b;`
Anónimas	Funciones sin nombre, generalmente usadas como callbacks o en expresiones.	`(function() { console.log("Soy anónima"); })();`
Autoejecutables (IIFE)	Funciones que se ejecutan automáticamente tras definirse.	`(function() { console.log("Hola mundo"); })();`
Generadoras	Funciones que pueden pausar su ejecución y reanudarla con `yield`. Se declaran con `function*`.	`function* contador() { let i = 0; while (true) yield i++; }`
Asíncronas	Permiten trabajar con operaciones asincrónicas usando `async` y `await`.	`async function obtenerDatos() { const res = await fetch(url); }`
Constructores	Usadas con `new` para crear objetos personalizados.	`function Persona(nombre) { this.nombre = nombre; }`

Métodos y Propiedades Relacionados con Funciones

Método/Propiedad	Descripción	Ejemplo
`call`	Llama una función con un `this` y argumentos específicos.	`saludar.call(obj, arg1, arg2);`

apply	Similar a `call`, pero los argumentos se pasan en un array o `arguments`.	`saludar.apply(obj, [arg1, arg2]);`
bind	Crea una nueva función con `this` vinculado a un objeto específico.	`const nuevaFuncion = saludar.bind(obj);`
arguments	Objeto especial disponible dentro de las funciones que contiene los argumentos pasados a la función.	`function suma() { return arguments[0] + arguments[1]; }`
length	Devuelve el número de argumentos esperados por la función.	`function sumar(a, b) {}; console.log(sumar.length);`
name	Devuelve el nombre de la función.	`console.log(saludar.name);`
toString	Devuelve una cadena con la definición de la función.	`console.log(saludar.toString());`
constructor	Devuelve la función constructora de la función (generalmente `Function`).	`console.log(sumar.constructor);`

Ejemplo Práctico con Diferentes Tipos de Funciones

```
// Declarativa
function saludar(nombre) {
    console.log(`Hola, ${nombre}`);
}
saludar("Carlos");

// Expresiva
const sumar = function(a, b) {
    return a + b;
};
console.log("Suma:", sumar(5, 3));

// Flecha
const multiplicar = (a, b) => a * b;
console.log("Multiplicación:", multiplicar(4, 6));
```

```
// Generadora
function* contador() {
    let i = 0;
    while (true) yield i++;
}
const gen = contador();
console.log("Generador:", gen.next().value, gen.next().value);

// Asíncrona
const fetchSimulado = () => new Promise(resolve => setTimeout(() =>
resolve("Datos recibidos"), 2000));

async function obtenerDatos() {
    const datos = await fetchSimulado();
    console.log(datos);
}
obtenerDatos();

// Autoejecutable (IIFE)
(function() {
    console.log("Hola desde una IIFE");
})();

// Constructor
function Persona(nombre) {
    this.nombre = nombre;
}
const juan = new Persona("Juan");
console.log("Persona:", juan.nombre);

// Uso de call, apply y bind
const persona = {
    nombre: "Luis",
    saludar: function(lugar) {
        console.log(`Hola, soy ${this.nombre} y estoy en ${lugar}`);
    }
};
persona.saludar("Madrid");

const otraPersona = { nombre: "Ana" };
persona.saludar.call(otraPersona, "Barcelona");
```

```
persona.saludar.apply(otraPersona, ["Valencia"]);
const saludarDesdeSevilla = persona.saludar.bind(otraPersona,
"Sevilla");
saludarDesdeSevilla();
```

Tema 5, POO

Tabla de Conceptos de POO en JavaScript

Concepto	Descripción	Ejemplo
Clase	Es una plantilla para crear objetos.	`class Persona { constructor(nombre) { this.nombre = nombre; } }`
Objeto	Es una instancia de una clase, creada con el operador `new`.	`const persona1 = new Persona('Juan');`
Propiedad	Es una variable que pertenece a una clase u objeto.	`this.nombre`
Método	Es una función que pertenece a una clase u objeto.	`hablar() { console.log('Hola'); }`
Encapsulación	Es la ocultación de los detalles internos del objeto y la exposición de solo lo necesario a través de métodos públicos.	`#edad = 25; getEdad() { return this.#edad; }`
Herencia	Permite que una clase (subclase) herede propiedades y métodos de otra (superclase).	`class Empleado extends Persona {}`
Polimorfismo	Es la capacidad de un método en una clase derivada para sobrescribir el comportamiento de un método de la clase base.	`saludar() { console.log('¡Hola, empleado!'); }` (sobreescribe `saludar` de la clase base)
Abstracción	Es la capacidad de definir clases abstractas que no pueden instanciarse,	`class Animal { constructor() { if (new.target === Animal) {`

	utilizadas como base para otras clases.	`throw new Error('No se puede instanciar'); } } }`
Static	Define propiedades o métodos que pertenecen a la clase en lugar de a las instancias.	`static crearPersona() { return new Persona('Predeterminada'); }`
Getters y Setters	Métodos especiales para obtener o establecer valores de propiedades privadas o protegidas.	`get nombre() { return this._nombre; } set nombre(valor) { this._nombre = valor; }`
Prototype	Es un mecanismo por el cual los objetos pueden compartir métodos y propiedades.	`Persona.prototype.saludar = function() { console.log('Hola desde el prototipo'); };`

Sistema básico de gestión de personas y empleados.

```
// Clase base Persona
class Persona {
    #edad; // Encapsulación: propiedad privada

    constructor(nombre, edad) {
        this.nombre = nombre;
        this.#edad = edad;
    }

    // Getter y Setter para edad
    get edad() {
        return this.#edad;
    }

    set edad(nuevaEdad) {
        if (nuevaEdad > 0) {
            this.#edad = nuevaEdad;
        } else {
            console.error('La edad debe ser positiva.');
        }
    }

    // Método de instancia
    saludar() {
```

```
        console.log(`Hola, soy ${this.nombre} y tengo ${this.#edad}
años.`);
    }

    // Método estático
    static crearAnonimo() {
        return new Persona('Anónimo', 30);
    }
}

// Subclase Empleado que hereda de Persona
class Empleado extends Persona {
    constructor(nombre, edad, puesto) {
        super(nombre, edad); // Llama al constructor de la clase base
        this.puesto = puesto;
    }

    // Sobreescritura de método
    saludar() {
        console.log(`Hola, soy ${this.nombre}, trabajo como
${this.puesto}.`);
    }
}

// Uso de las clases
const persona1 = new Persona('Luis', 25);
persona1.saludar(); // Hola, soy Luis y tengo 25 años.
persona1.edad = 26; // Cambia la edad usando el setter
console.log(`Nueva edad: ${persona1.edad}`); // Nueva edad: 26

const empleado1 = new Empleado('Ana', 30, 'Ingeniera');
empleado1.saludar(); // Hola, soy Ana, trabajo como Ingeniera.

const anonimo = Persona.crearAnonimo(); // Método estático
anonimo.saludar(); // Hola, soy Anónimo y tengo 30 años.
```

Expresiones Regulares (RegEx)

Una **expresión regular** es un patrón utilizado para buscar coincidencias dentro de cadenas de texto. Es muy útil para validar, buscar o reemplazar contenido en textos.

---

1. Componentes básicos de una expresión regular

Las expresiones regulares tienen caracteres especiales que ayudan a definir patrones. algunos elementos básicos:

Elemento	Descripción	Ejemplo	Resultado
`.`	Coincide con **cualquier carácter** excepto nueva línea	`/a.c/`	Coincide con "abc", "axc", pero no con "ac".
`^`	Coincide con el **inicio** de una cadena	`/^Hola/`	Coincide con cadenas que empiezan con "Hola".
`$`	Coincide con el **final** de una cadena	`/adiós$/`	Coincide con cadenas que terminan con "adiós".
`*`	Coincide con **cero o más repeticiones**	`/ab*c/`	Coincide con "ac", "abc", "abbc", etc.
`+`	Coincide con **una o más repeticiones**	`/ab+c/`	Coincide con "abc", "abbc", pero no con "ac".
`?`	Hace que el carácter anterior sea **opcional**	`/colou?r/`	Coincide con "color" y "colour".
`[]`	Define un **conjunto de caracteres** permitidos	`/[aeiou]/`	Coincide con cualquier vocal.
`\d`	Coincide con un **dígito** (equivale a `[0-9]`)	`/\d+/`	Coincide con "123", "456", etc.
`\w`	Coincide con un **carácter alfanumérico** (A-Z, 0-9, _)	`/\w+/`	Coincide con "hello_123".
`\s`	Coincide con un **espacio en blanco**	`/\s+/`	Coincide con espacios, tabulaciones, etc.

`|`	`|`	Operador **"o" lógico**	`|/gato`
`( )`	Agrupa patrones	`/(ab)+/`	Coincide con "abab".

---

2. Métodos principales en JavaScript

JavaScript tiene métodos que facilitan el trabajo con expresiones regulares.

test

Comprueba si el patrón existe en una cadena (devuelve `true` o `false`):

```
const regex = /hola/i; // La "i" significa insensible a mayúsculas
console.log(regex.test("Hola mundo")); // true
```

match

Devuelve las coincidencias encontradas en un array (o `null` si no hay coincidencias):

```
const texto = "El número es 12345";
console.log(texto.match(/\d+/)); // ["12345"]
```

replace

Reemplaza el texto que coincide con el patrón:

```
const texto = "Hola mundo";
console.log(texto.replace(/mundo/, "amigos")); // "Hola amigos"
```

split

Divide un string en partes usando una RegEx como separador:

```
const texto = "uno,dos,tres";
console.log(texto.split(/,/)); // ["uno", "dos", "tres"]
```

Validaciones comunes

1. Validar un correo electrónico

Comprueba si una cadena tiene un formato de correo válido.

```
const emailRegex =
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

console.log(emailRegex.test("usuario@dominio.com")); // true
console.log(emailRegex.test("usuario@dominio"));     // false
```

---

2. Validar un número de teléfono (formato internacional)

Formato: +34 600 123 456 o 600123456.

```
const phoneRegex = /^(\+34\s?)?(\d{9}|\d{3}\s\d{3}\s\d{3})$/;

console.log(phoneRegex.test("+34 600 123 456")); // true
console.log(phoneRegex.test("600123456"));      // true
console.log(phoneRegex.test("12345"));          // false
```

---

3. Validar una URL

Comprueba si una cadena tiene el formato de una URL.

```
const urlRegex =
/^(https?:\/\/)?([\w.-]+)\.([a-z]{2,6})(\/[^\s]*)?$/;

console.log(urlRegex.test("https://www.example.com")); // true
console.log(urlRegex.test("http://example.org/page")); // true
console.log(urlRegex.test("www.example"));            // false
```

Búsquedas y coincidencias

4. Buscar todas las palabras en una cadena

Devuelve un array con todas las palabras (separadas por espacios).

```
const text = "Hola, mundo! Aprende JavaScript.";
const wordRegex = /\b\w+\b/g;

console.log(text.match(wordRegex)); // ["Hola", "mundo", "Aprende",
"JavaScript"]
```

---

5. Buscar números en una cadena

Encuentra todas las secuencias de dígitos.

```
const text = "Hay 15 manzanas y 20 naranjas.";
const numberRegex = /\d+/g;

console.log(text.match(numberRegex)); // ["15", "20"]
```

---

6. Extraer etiquetas HTML

Obtén todas las etiquetas <...> de un texto.

```
const html = "<div><p>Hola</p><span>Mundo</span></div>";
const tagRegex = /<\/?[\w\s="'-]+>/g;

console.log(html.match(tagRegex)); // ["<div>", "<p>", "</p>",
"<span>", "</span>", "</div>"]
```

Reemplazos

7. Reemplazar todas las vocales

Convierte todas las vocales en mayúsculas.

```
const text = "Hola, mundo!";
const vowelRegex = /[aeiouáéíóúü]/gi;

console.log(text.replace(vowelRegex, match => match.toUpperCase()));
```

```
// "HOlA, mUndO!"
```

---

8. Eliminar espacios extra

Reduce múltiples espacios consecutivos a uno solo.

```
const text = "Este   texto   tiene   muchos   espacios.";
const spaceRegex = /\s+/g;

console.log(text.replace(spaceRegex, " ")); // "Este texto tiene
muchos espacios."
```

---

9. Censurar palabras inapropiadas

Reemplaza palabras no deseadas por asteriscos.

```
const text = "Esto es un mal ejemplo de lenguaje grosero.";
const badWordRegex = /\b(mal|grosero)\b/gi;

console.log(text.replace(badWordRegex, "****"));
// "Esto es un **** ejemplo de lenguaje ****."
```

Validaciones avanzadas

10. Validar contraseñas fuertes

Requiere al menos 8 caracteres, una mayúscula, una minúscula, un número y un carácter especial.

```
const passwordRegex =
/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}
$/;

console.log(passwordRegex.test("Hola123!"));    // true
console.log(passwordRegex.test("hola123"));     // false
console.log(passwordRegex.test("12345678"));    // false
```

---

11. Validar fechas (formato dd/mm/yyyy o dd-mm-yyyy)

Asegúrate de que la fecha tenga un formato correcto.

```
const dateRegex =
/^(0?[1-9]|[12][0-9]|3[01])[\/-](0?[1-9]|1[0-2])[\/-]\d{4}$/;

console.log(dateRegex.test("25/12/2024")); // true
console.log(dateRegex.test("31-01-1999")); // true
console.log(dateRegex.test("99/99/9999")); // false
```

---

12. Validar códigos postales (España)

Valida un código postal español de 5 dígitos, comenzando con 01-52.

```
const postalCodeRegex = /^(0[1-9]|[1-4]\d|5[0-2])\d{3}$/;

console.log(postalCodeRegex.test("28080")); // true
console.log(postalCodeRegex.test("52999")); // false
```

Métodos para practicar con RegEx en JavaScript

Combinación de búsqueda y validación

Podemos buscar coincidencias y, al mismo tiempo, validarlas.

```
const text = "Mi correo es usuario@example.com y mi web es
https://example.com.";
const emailRegex =
/[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}/g;
const urlRegex = /https?:\/\/[^\s]+/g;

console.log(text.match(emailRegex)); // ["usuario@example.com"]
console.log(text.match(urlRegex));  // ["https://example.com"]
```

Iterar sobre coincidencias

Podemos usar exec en un bucle para capturar múltiples coincidencias.

```
const text = "El precio es $5, luego $10 y después $15.";
const priceRegex = /\$\d+/g;

let match;
while ((match = priceRegex.exec(text)) !== null) {
  console.log(match[0]); // "$5", "$10", "$15"
}
```

Interacción con JSON

El JSON (**JavaScript Object Notation**) es un formato ligero para intercambiar datos. Es fácil de leer y escribir, y está basado en la sintaxis de objetos de JavaScript.

---

1. Conversión de JSON a un objeto JavaScript

Usamos JSON.parse para convertir una cadena JSON en un objeto.

```
const jsonString = '{"nombre": "Luis", "edad": 30}';
const objeto = JSON.parse(jsonString);

console.log(objeto.nombre); // "Luis"
console.log(objeto.edad);   // 30
```

---

2. Conversión de un objeto JavaScript a JSON

Usamos JSON.stringify para convertir un objeto JavaScript a una cadena JSON.

```
const objeto = { nombre: "Ana", edad: 25 };
const jsonString = JSON.stringify(objeto);
```

```
console.log(jsonString); // '{"nombre":"Ana","edad":25}'
```

---

3. Acceso y manipulación de datos JSON

Una vez que el JSON se convierte en un objeto JavaScript, podemos acceder y manipular sus propiedades como cualquier objeto normal.

```
const jsonString = '{"ciudad": "Madrid", "poblacion": 3265000}';
const datos = JSON.parse(jsonString);

// Acceder a datos
console.log(datos.ciudad); // "Madrid"

// Modificar datos
datos.poblacion += 50000;

// Convertir de vuelta a JSON
const nuevoJSON = JSON.stringify(datos);
console.log(nuevoJSON); // '{"ciudad":"Madrid","poblacion":3310000}'
```

---

4. JSON y APIs

Cuando trabajamos con APIs, el formato JSON se utiliza para enviar y recibir datos.

```
fetch("https://api.example.com/data")
  .then(response => response.json()) // Convertir JSON a objeto
  .then(data => console.log(data))   // Trabajar con los datos
  .catch(error => console.error("Error:", error));
```

Resumen

- Las **expresiones regulares** son patrones potentes para validar, buscar o manipular cadenas.

- JSON es un formato clave para el intercambio de datos, y `JSON.parse` y `JSON.stringify` nos permiten convertir entre cadenas JSON y objetos JavaScript.

Tema 6, Modelo de Objetos del Cliente

Atributos Data (`data-*`)

¿Qué son los Data Attributes?

Los **Data Attributes** son atributos personalizados en HTML5 que permiten almacenar datos adicionales en elementos HTML. Se identifican con el prefijo `data-` y son útiles para manejar información dinámica o específica.

html
Copiar código
```
<div data-user-id="12345" data-role="admin">Usuario</div>
```

- **`data-user-id`** y **`data-role`** son atributos personalizados.
- Estos almacenan información (como el ID y el rol del usuario) que puede usarse en JavaScript para manipular o acceder a los datos.

Ventajas:

- **Flexibilidad:** Permite definir atributos personalizados para adaptarse a las necesidades de la aplicación.
- **Compatibilidad:** No interfiere con los atributos estándar de HTML.
- **Acceso fácil desde JavaScript:** Se pueden manipular directamente con el objeto `dataset` o métodos como `getAttribute`.

Acceso desde JavaScript

1. **Usando `getAttribute`**:
```
const element = document.querySelector('div');
console.log(element.getAttribute('data-user-id')); // "12345"
```

2. **Usando `dataset`**:
```
const element = document.querySelector('div');
console.log(element.dataset.userId); // "12345"
element.dataset.role = 'editor'; // Cambia el valor de data-role
```

Propiedades y Métodos del DOM

Propiedades principales de los elementos

Propiedad	Descripción	Ejemplo
`id`	Obtiene o establece el atributo `id` del elemento.	`element.id = 'nuevoId';`
`className`	Obtiene o establece las clases como texto.	`element.className = 'miClase';`
`classList`	Facilita la manipulación de clases como un objeto.	`element.classList.add('miClase');`
`innerHTML`	Obtiene o establece el contenido HTML interno.	`element.innerHTML = '<p>Hola</p>';`
`outerHTML`	Igual que `innerHTML`, pero incluye el propio elemento.	`element.outerHTML = '<div>Nuevo</div>';`
`textContent`	Obtiene o establece el texto, excluyendo etiquetas HTML.	`element.textContent = 'Hola';`
`style`	Manipula estilos CSS en línea.	`element.style.color = 'red';`
`children`	Devuelve una colección de hijos elementos.	`const hijos = element.children;`
`parentElement`	Devuelve el elemento padre.	`const padre = element.parentElement;`
`nextElementSibling`	Devuelve el siguiente hermano elemento.	`const siguiente = element.nextElementSibling;`
`previousElementSibling`	Devuelve el hermano anterior elemento.	`const anterior = element.previousElementSibling;`

`dataset`	Accede a los `data-*` como objeto.	`element.dataset.userId = '123';`

Métodos principales del DOM

Método	Descripción	Ejemplo
`getAttribute(name)`	Obtiene el valor de un atributo.	`element.getAttribute('src');`
`setAttribute(name, value)`	Establece o actualiza un atributo.	`element.setAttribute('alt', 'imagen');`
`removeAttribute(name)`	Elimina un atributo.	`element.removeAttribute('id');`
`hasAttribute(name)`	Verifica si un atributo existe (`true` o `false`).	`element.hasAttribute('href');`
`appendChild(child)`	Agrega un nodo hijo.	`element.appendChild(nuevoElemento);`
`removeChild(child)`	Elimina un nodo hijo.	`element.removeChild(hijo);`
`replaceChild(newChild, oldChild)`	Reemplaza un nodo hijo con otro.	`element.replaceChild(nuevo, viejo);`
`cloneNode(deep)`	Crea una copia del nodo.	`const copia = element.cloneNode(true);`
`insertAdjacentHTML(pos, html)`	Inserta HTML en una posición relativa.	`element.insertAdjacentHTML('beforeend', '<p>Hola</p>');`
`focus()`	Lleva el foco al elemento.	`element.focus();`

`blur()`	Quita el foco del elemento.	`element.blur();`

Manipulación de clases con `classList`

Método	Descripción	Ejemplo
`add(className)`	Agrega una clase al elemento.	`element.classList.add('miClase');`
`remove(className)`	Elimina una clase del elemento.	`element.classList.remove('miClase');`
`toggle(className)`	Alterna entre agregar y eliminar una clase.	`element.classList.toggle('miClase');`
`contains(className)`	Verifica si el elemento tiene una clase (`true/false`).	`element.classList.contains('miClase');`
`replace(old, new)`	Reemplaza una clase por otra.	`element.classList.replace('vieja', 'nueva');`

Selección de Elementos en el DOM

Método	Descripción	Devuelve	Ejemplo
`getElementById(id)`	Selecciona un elemento por su `id`.	Elemento	`document.getElementById('miId');`
`getElementsByClassName(class)`	Selecciona elementos por clase.	HTMLCollection	`document.getElementsByClassName('miClase');`
`getElementsByTagName(tag)`	Selecciona elementos por etiqueta HTML.	HTMLCollection	`document.getElementsByTagName('div');`

`querySelector(selector)`	Selecciona el primer elemento que coincida con un selector CSS.	Elemento	`document.querySelector('.miClase');`
`querySelectorAll(selector)`	Selecciona todos los elementos que coincidan con un selector CSS.	NodeList	`document.querySelectorAll('div');`

Diferencias entre HTMLCollection y NodeList:

- **HTMLCollection:** Se actualiza dinámicamente si cambia el DOM.
- **NodeList:** Es estática y permite métodos como `.forEach()`.

Ejemplo completo:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Manipulación DOM Completa</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 20px;
      line-height: 1.5;
    }
    .dynamic-container {
      border: 1px solid #ddd;
      padding: 10px;
      margin-top: 10px;
    }
    .highlight {
      background-color: #ffeb3b;
```

```
      color: #000;
      font-weight: bold;
    }
    button {
      padding: 10px 15px;
      margin: 5px;
      border: none;
      background-color: #007bff;
      color: white;
      cursor: pointer;
      border-radius: 5px;
    }
    button:hover {
      background-color: #0056b3;
    }
  </style>
</head>
<body>
  <h1>Ejemplo Completo de Manipulación del DOM</h1>
  <p>Presiona los botones para interactuar dinámicamente con el
contenido.</p>

  <button id="add-element">Agregar elemento dinámico</button>
  <button id="toggle-highlight">Alternar clase "highlight"</button>
  <button id="update-data">Actualizar atributos data-*</button>
  <button id="show-data">Mostrar todos los atributos data-*</button>

  <div id="container" class="dynamic-container"
data-name="contenedor" data-count="0">
    <p>Este es un contenedor dinámico. Su contenido cambiará al
interactuar con los botones.</p>
  </div>

  <script>
    // Seleccionar el contenedor principal
    const container = document.getElementById('container');

    // Función para agregar un elemento dinámico
    document.getElementById('add-element').addEventListener('click',
() => {
      const newElement = document.createElement('div');
```

```
      const count = parseInt(container.dataset.count) + 1;
      newElement.textContent = `Elemento dinámico #${count}`;
      newElement.setAttribute('data-item', count);
      newElement.className = 'dynamic-item';
      container.appendChild(newElement);

      // Actualizar el contador en data-count
      container.dataset.count = count;
    });

    // Función para alternar la clase "highlight" en el contenedor

document.getElementById('toggle-highlight').addEventListener('click'
, () => {
      container.classList.toggle('highlight');
    });

    // Función para actualizar atributos data-*
    document.getElementById('update-data').addEventListener('click',
() => {
      const currentName = container.dataset.name;
      const newName = currentName === 'contenedor' ?
'nuevo-contenedor' : 'contenedor';
      container.dataset.name = newName;

      alert(`Atributo "data-name" actualizado a: ${newName}`);
    });

    // Función para mostrar todos los atributos data-*
    document.getElementById('show-data').addEventListener('click',
() => {
      const dataAttributes = Object.entries(container.dataset);
      const messages = dataAttributes.map(([key, value]) =>
`data-${key}: ${value}`);
      alert('Atributos data-* del contenedor:\n' +
messages.join('\n'));
    });

    // Evento dinámico en elementos hijos (delegación de eventos)
    container.addEventListener('click', (event) => {
      if (event.target.classList.contains('dynamic-item')) {
```

```
        const itemData = event.target.getAttribute('data-item');
        alert(`Hiciste clic en el elemento dinámico con
data-item="${itemData}"`);
      }
    });
  </script>
</body>
</html>
```

Tema 7, Eventos

1. Eventos de ratón

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Ratón</title>
</head>
<body>
    <button id="mouseButton">Haz algo con el ratón</button>

    <script>
        const button = document.getElementById('mouseButton');

        button.addEventListener('click', () => alert('¡Hiciste
clic!'));
        button.addEventListener('dblclick', () => alert('¡Doble
clic!'));
        button.addEventListener('mouseover', () =>
button.style.backgroundColor = 'lightblue');
        button.addEventListener('mouseout', () =>
button.style.backgroundColor = '');
        button.addEventListener('mousedown', () =>
button.textContent = '¡Presionaste el botón!');
        button.addEventListener('mouseup', () => button.textContent
= '¡Soltaste el botón!');
    </script>
```

```
</body>
</html>
```

---

2. Eventos de teclado

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Teclado</title>
</head>
<body>
    <input type="text" id="keyboardInput" placeholder="Escribe
algo">

    <script>
        const input = document.getElementById('keyboardInput');

        input.addEventListener('keydown', () => console.log('Tecla
presionada'));
        input.addEventListener('keyup', () => console.log('Tecla
soltada'));
    </script>
</body>
</html>
```

---

3. Eventos de formulario

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de Formulario</title>
</head>
```

```
<body>
    <form id="myForm">
        <input type="text" id="formInput" placeholder="Escribe
aquí">
        <select id="formSelect">
            <option value="opcion1">Opción 1</option>
            <option value="opcion2">Opción 2</option>
        </select>
        <button type="submit">Enviar</button>
    </form>

    <script>
        const form = document.getElementById('myForm');
        const input = document.getElementById('formInput');
        const select = document.getElementById('formSelect');

        form.addEventListener('submit', (event) => {
            event.preventDefault(); // Evita que se recargue la
página
            alert('Formulario enviado');
        });

        select.addEventListener('change', () =>
alert(`Seleccionaste: ${select.value}`));
        input.addEventListener('input', () => console.log(`Texto:
${input.value}`));
        input.addEventListener('focus', () =>
input.style.backgroundColor = 'lightyellow');
        input.addEventListener('blur', () =>
input.style.backgroundColor = '');
    </script>
</body>
</html>
```

---

4. Eventos de la ventana o documento

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos de la Ventana</title>
</head>
<body>
    <h1>Redimensiona o desplaza la ventana</h1>

    <script>
        window.addEventListener('load', () => console.log('Página
cargada'));
        window.addEventListener('resize', () => console.log('Tamaño
de ventana cambiado'));
        window.addEventListener('scroll', () => console.log('Has
desplazado la página'));
    </script>
</body>
</html>
```

---

5. Eventos táctiles

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>Eventos Táctiles</title>
</head>
<body>
    <div id="touchArea" style="width: 100%; height: 200px;
background-color: lightgray;">
        Toca aquí
    </div>

    <script>
        const touchArea = document.getElementById('touchArea');

        touchArea.addEventListener('touchstart', () =>
console.log('Inició el toque'));
        touchArea.addEventListener('touchmove', () =>
console.log('Arrastrando...'));
```

```
        touchArea.addEventListener('touchend', () =>
console.log('Toque finalizado'));
    </script>
</body>
</html>
```

Ejemplos con tablas:

1. Eventos de Mouse

Evento	Descripción	Ejemplo
`click`	Ocurre cuando el usuario hace clic en un elemento.	`element.addEventListener('click', () => {...});`
`dblclick`	Ocurre cuando el usuario hace doble clic en un elemento.	`element.addEventListener('dblclick', () => {...});`
`mousemove`	Se activa cuando el puntero se mueve dentro del área del elemento.	`element.addEventListener('mousemove', () => {...});`
`mouseover`	Ocurre cuando el puntero entra en el área de un elemento.	`element.addEventListener('mouseover', () => {...});`
`mouseout`	Ocurre cuando el puntero sale del área de un elemento.	`element.addEventListener('mouseout', () => {...});`
`mousedown`	Ocurre cuando el botón del ratón se presiona sobre un elemento.	`element.addEventListener('mousedown', () => {...});`
`mouseup`	Ocurre cuando se suelta el botón del ratón sobre un elemento.	`element.addEventListener('mouseup', () => {...});`
`contextmenu`	Ocurre al abrir el menú contextual (botón derecho del ratón).	`element.addEventListener('context menu', () => {...});`

---

2. Eventos de Teclado

Evento	Descripción	Ejemplo

`keydown`	Ocurre cuando una tecla se presiona.	`document.addEventListener('keydown', () => {...});`
`keypress`	**[Deprecado]** Igual que `keydown`, pero no incluye teclas no imprimibles.	`document.addEventListener('keypress', () => {...});`
`keyup`	Ocurre cuando se suelta una tecla.	`document.addEventListener('keyup', () => {...});`

---

3. Eventos de Formulario

Evento	Descripción	Ejemplo
`submit`	Ocurre al enviar un formulario.	`form.addEventListener('submit', (e) => {...});`
`change`	Ocurre cuando un elemento `<input>`, `<select>` o `<textarea>` cambia de valor.	`input.addEventListener('change', () => {...});`
`input`	Ocurre cada vez que se modifica el valor de un elemento de entrada.	`input.addEventListener('input', () => {...});`
`focus`	Ocurre cuando un elemento recibe el foco.	`element.addEventListener('focus', () => {...});`
`blur`	Ocurre cuando un elemento pierde el foco.	`element.addEventListener('blur', () => {...});`
`reset`	Ocurre cuando se reinicia un formulario.	`form.addEventListener('reset', () => {...});`

---

4. Eventos de Documento y Ventana

Evento	Descripción	Ejemplo
`DOMContentLoaded`	Ocurre cuando el DOM se ha cargado completamente.	`document.addEventListener('DOMContentLoaded', () => {});`

`load`	Ocurre cuando una página o recurso específico (imágenes, scripts) se carga.	`window.addEventListener('load', () => {...});`
`resize`	Ocurre cuando se redimensiona la ventana.	`window.addEventListener('resize', () => {...});`
`scroll`	Ocurre cuando el usuario desplaza la barra de scroll.	`window.addEventListener('scroll', () => {...});`
`unload`	Ocurre cuando la página se está descargando.	`window.addEventListener('unload', () => {...});`

---

5. Eventos de Drag & Drop

Evento	Descripción	Ejemplo
`drag`	Ocurre cuando un elemento se arrastra.	`element.addEventListener('drag', () => {...});`
`dragstart`	Se dispara al iniciar el arrastre de un elemento.	`element.addEventListener('dragstart', () => {...});`
`dragend`	Se dispara al terminar el arrastre de un elemento.	`element.addEventListener('dragend', () => {...});`
`dragover`	Ocurre cuando un elemento arrastrado pasa sobre un área válida.	`area.addEventListener('dragover', (e) => { e.preventDefault(); });`
`drop`	Se dispara al soltar un elemento en un área válida.	`area.addEventListener('drop', (e) => {...});`

---

6. Eventos de Multimedia

Evento	Descripción	Ejemplo

`play`	Ocurre cuando se inicia la reproducción de un elemento multimedia.	`video.addEventListener('play', () => {...});`
`pause`	Ocurre cuando la reproducción es pausada.	`audio.addEventListener('pause', () => {...});`
`ended`	Ocurre cuando la reproducción termina.	`video.addEventListener('ended', () => {...});`
`timeupdate`	Se activa cuando el tiempo actual de reproducción cambia.	`video.addEventListener('timeupdate', () => {...});`

---

7. Eventos de Otros Tipos

Evento	Descripción	Ejemplo
`error`	Se activa cuando ocurre un error en la carga de recursos.	`img.addEventListener('error', () => {...});`
`copy`	Ocurre cuando se copia contenido.	`document.addEventListener('copy', () => {...});`
`paste`	Ocurre cuando se pega contenido.	`document.addEventListener('paste', () => {...});`

Diferencia entre `this` y `event.target` en eventos de JavaScript

Cuando trabajamos con manejadores de eventos en JavaScript, a menudo necesitamos distinguir entre dos conceptos clave: **`this`** y **`event.target`**. Aunque ambos se utilizan en el contexto de eventos, representan diferentes cosas.

---

1. `this` en los manejadores de eventos

Qué representa:
Hace referencia al **elemento al que está vinculado el manejador de eventos**. En otras palabras, `this` apunta al **elemento que recibió el evento**.

Cómo funciona:

- El valor de `this` se determina por el contexto de ejecución de la función.

- En un manejador de eventos registrado con `addEventListener`, `this` generalmente apunta al elemento donde se vinculó el manejador (es decir, el elemento que disparó el evento).
- Si usas **funciones flecha** como manejadores de eventos, el valor de `this` no se enlazará al elemento objetivo, sino que conservará el valor del **contexto externo**.

Ejemplo con `this`:

```
<button id="myButton">Click Me</button>

<script>
const button = document.getElementById('myButton');

// Agregamos un evento al botón
button.addEventListener('click', function () {
    console.log(this); // "this" se refiere al botón
});
</script>
```

En este caso, `this` apunta al botón que recibió el clic.

2. `event.target` en los manejadores de eventos

Qué representa:
`event.target` hace referencia al **elemento específico que disparó el evento**. Si un elemento hijo dentro de un contenedor dispara el evento, `event.target` se refiere al hijo, no al contenedor que tiene el manejador.

Cómo funciona:

- **`event.target`** siempre apunta al **elemento donde ocurrió físicamente el evento**, independientemente de dónde se haya registrado el manejador.
- Se usa especialmente cuando estamos manejando eventos en un contenedor o cuando trabajamos con **eventos delegados**.

Ejemplo con `event.target`:

```
<div id="container">
    <button>Button 1</button>
    <button>Button 2</button>
</div>

<script>
const container = document.getElementById('container');

container.addEventListener('click', function (event) {
```

```
    console.log(event.target); // "event.target" se refiere al botón
que fue clicado
    console.log(this);  // "this" se refiere al contenedor <div>
});
</script>
```

Si haces clic en uno de los botones dentro del `div`, `event.target` se referirá al botón específico, mientras que `this` seguirá apuntando al `div` contenedor.

3. ¿Se pueden usar indistintamente?

No siempre. Aunque en algunos casos ambos pueden coincidir (cuando haces clic directamente en el elemento que tiene el manejador), hay situaciones donde no son intercambiables:

- **Usa `this`** cuando necesites trabajar con el **elemento que tiene el manejador de eventos**.
- **Usa `event.target`** cuando necesites identificar **el elemento exacto que originó el evento**, especialmente en eventos delegados.

4. Diferencia clave en eventos delegados

En eventos delegados, el manejador de eventos se registra en un **elemento contenedor** y usamos **`event.target`** para identificar el **elemento hijo** que originó el evento.

Evento Delegado

¿Qué es un evento delegado?

La **delegación de eventos** es una técnica en JavaScript que permite asignar un único manejador de eventos a un **elemento padre** (contenedor) para gestionar los eventos de sus **elementos hijos**. Incluso si los elementos hijos se agregan dinámicamente al DOM, el manejador en el contenedor seguirá funcionando.

Cómo funciona la delegación de eventos

Cuando un evento ocurre en un elemento, **este evento burbujea** (bubble) hacia arriba en el árbol DOM, propagándose desde el elemento que disparó el evento hasta sus ancestros. En los eventos delegados, aprovechamos esta propagación colocando un único manejador en un **elemento ancestro** y utilizando **`event.target`** para identificar el **elemento hijo** que disparó el evento.

Ventajas de la delegación de eventos

1. **Menor consumo de recursos**:
   No necesitamos agregar un manejador a cada elemento hijo, solo uno en el contenedor.
2. **Soporte para elementos dinámicos**:
   Si los elementos hijos son creados o eliminados dinámicamente, el manejador en el contenedor sigue funcionando sin necesidad de reconfigurarlos.
3. **Mantenimiento más sencillo**:
   Con menos manejadores, el código es más fácil de mantener y depurar.

Ejemplo práctico sin delegación

En este ejemplo, necesitamos agregar un manejador de eventos a cada `li` de una lista. Esto puede ser ineficiente si la lista tiene muchos elementos o si se agregan dinámicamente.

```
<ul>
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>

<script>
const items = document.querySelectorAll('li');
items.forEach(item => {
    item.addEventListener('click', () => {
        console.log('Elemento clicado:', item.textContent);
    });
});
</script>
```

Ejemplo práctico con delegación

Aquí, usamos **delegación de eventos**: colocamos un único manejador de eventos en el `ul` (el contenedor de los `li`). Usamos **`event.target`** para identificar qué `li` fue clicado.

```
<ul id="list">
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>
```

```
<script>
const list = document.getElementById('list');
list.addEventListener('click', function (event) {
    if (event.target.tagName === 'LI') {
        console.log('Elemento clicado:', event.target.textContent);
    }
});
</script>
```

En este caso:

- Solo hay un manejador en el `ul`.
- Si agregamos nuevos elementos `li` dinámicamente, el manejador en el `ul` seguirá funcionando.

Cuándo usar eventos delegados

1. **Grandes cantidades de elementos**:
   Es útil cuando tenemos muchos elementos similares y queremos reducir la cantidad de manejadores.
2. **Elementos dinámicos**:
   Ideal para listas, tablas o galerías donde los elementos pueden ser añadidos o eliminados después de cargar la página.
3. **Mejorar el rendimiento**:
   Permite optimizar la memoria y el rendimiento al reducir la cantidad de manejadores y aprovechar la burbujeo de eventos.

Resumen

- **`this`** hace referencia al **elemento al que está vinculado el manejador de eventos**.
- **`event.target`** hace referencia al **elemento que realmente disparó el evento**.
- La **delegación de eventos** permite manejar eventos en un contenedor para todos los elementos dentro de él, utilizando `event.target` para identificar el origen del evento.

Espero que esto te ayude a entender mejor la diferencia entre **`this`** y **`event.target`**, así como los beneficios de usar **eventos delegados** en tus aplicaciones JavaScript.

Tema 8, Control de datos y API

1. Errores en JavaScript

Tipos de Errores en JavaScript

Tipo	Descripción	Ejemplo
Errores en desarrollo	Se producen al escribir código incorrecto. El editor los detecta antes de ejecutar el programa.	`console.lo("Error"); // SyntaxError`
Errores en ejecución	Ocurren mientras la aplicación se ejecuta, como variables no definidas o errores en cálculos.	`console.log(variableNoDefinida); // ReferenceError`

Tipos de Errores según su impacto

Tipo	Descripción	Ejemplo
Error	Detiene la ejecución del programa completamente.	`throw new Error("Error grave");`
Excepción	Se prevé y gestiona mediante `try...catch`, permitiendo que la ejecución continúe.	`try { JSON.parse("texto"); } catch(e) { console.log("Error en JSON"); }`
Warning	No detiene la ejecución, pero indica un problema potencial. (No existen oficialmente en JS).	`console.warn("Cuidado con esta acción");`

Manejo de Errores con `try...catch...finally`

```
try {
    let x = 10;
    x.toUpperCase(); // Esto causará un error
} catch (error) {
    console.error("Se ha producido un error:", error.message);
} finally {
    console.log("Este bloque siempre se ejecuta.");
}
```

---

2. Módulos en JavaScript

Los **módulos** permiten dividir el código en archivos reutilizables. Se usan con `import` y `export`.

Exportar un Módulo

Archivo **`matematica.js`**:

```
export const PI = 3.1416;

export function sumar(a, b) {
    return a + b;
}

export function multiplicar(a, b) {
    return a * b;
}
```

Importar un Módulo

Archivo **`app.js`**:

```
import { PI, sumar } from './matematica.js';

console.log("El valor de PI es:", PI);
console.log("Suma:", sumar(5, 3));
```

Importación con Alias

```
import { sumar as sumaNumeros } from './matematica.js';

console.log(sumaNumeros(10, 5)); // 15
```

Importar Todo un Módulo con un Alias

```
import * as math from './matematica.js';

console.log(math.PI);
console.log(math.sumar(2, 2));
```

---

3. API en JavaScript

¿Qué es una API?

Una **API** (Application Programming Interface) permite la comunicación entre aplicaciones o servicios.

Tipos de API en JavaScript

API	Descripción
DOM API	Manipula el documento HTML y sus elementos (`document.querySelector`, `document.createElement`).
Fetch API	Permite hacer solicitudes HTTP asíncronas (`fetch("https://api.example.com")`).
Web Storage API	Almacena datos en el navegador (`localStorage`, `sessionStorage`).
Geolocation API	Obtiene la ubicación del usuario (`navigator.geolocation.getCurrentPosition`).
Notification API	Envia notificaciones al usuario (`new Notification("Mensaje")`).

Ejemplo de Fetch API

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("Error al obtener datos:",
error));
```

---

4. LocalStorage y Cookies

Diferencia entre LocalStorage y Cookies

Característica	LocalStorage	Cookies
Tamaño máximo	Hasta varios MB.	4 KB.

Persistencia	Hasta que el usuario lo borre.	Se pueden definir fechas de expiración.
Accesible desde	Solo JavaScript.	JavaScript y el servidor.
Uso típico	Guardar configuraciones, datos de usuario temporales.	Identificación de usuario, datos de sesión.

Uso de LocalStorage

```
// Guardar dato
localStorage.setItem("usuario", "Juan");

// Leer dato
console.log(localStorage.getItem("usuario")); // "Juan"

// Eliminar dato
localStorage.removeItem("usuario");

// Limpiar todo el almacenamiento
localStorage.clear();
```

Guardar Objetos en LocalStorage

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

// Convertir a JSON y guardar
localStorage.setItem("perfil", JSON.stringify(usuario));

// Recuperar y convertir de JSON a objeto
const perfilRecuperado = JSON.parse(localStorage.getItem("perfil"));
console.log(perfilRecuperado.nombre); // "Ana"
```

Uso de Cookies

```
// Crear una cookie (expira en 7 días)
document.cookie = "usuario=Juan; expires=" + new Date(2025, 0,
1).toUTCString();

// Leer cookies
console.log(document.cookie);
```

```
// Eliminar cookie (se pone una fecha expirada)
document.cookie = "usuario=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Ejemplo Completo: API, LocalStorage y Notificaciones

```
// Guardar un dato en localStorage
localStorage.setItem("visitas",
parseInt(localStorage.getItem("visitas") || 0) + 1);

// Mostrar una notificación si el usuario ha visitado más de 5 veces
if (localStorage.getItem("visitas") > 5) {
    Notification.requestPermission().then(permission => {
        if (permission === "granted") {
            new Notification("¡Gracias por visitarnos más de 5
veces!");
        }
    });
}

// Obtener ubicación del usuario
navigator.geolocation.getCurrentPosition(position => {
    console.log(`Latitud: ${position.coords.latitude}, Longitud:
${position.coords.longitude}`);
});
```

Conclusión

- **Errores**: Debemos manejarlos con `try...catch` para evitar que el programa falle.
- **Módulos**: Permiten dividir el código en archivos reutilizables con `import` y `export`.
- **API**: JavaScript ofrece muchas API útiles como `Fetch`, `Geolocation` y `Notification`.
- **LocalStorage y Cookies**: Se usan para almacenar datos en el navegador.

1. Introducción a Web Storage y Cookies

Web Storage

Característica	localStorage	sessionStorage
Persistencia	Los datos se mantienen hasta que se borran manualmente o mediante código.	Los datos se mantienen solo durante la sesión; se eliminan al cerrar la pestaña.
Capacidad	Permite almacenar varios megabytes de datos.	Similar a localStorage (generalmente, la misma capacidad).
Accesible Desde	Solo desde JavaScript en el navegador.	Solo desde JavaScript en el navegador.
Uso Principal	Guardar configuraciones, preferencias del usuario, caché local, etc.	Almacenar datos temporales que solo se necesitan mientras la pestaña está abierta.

Cookies

Característica	Cookies
Persistencia	Se definen mediante fechas de expiración; pueden ser de sesión o persistentes.
Capacidad	Generalmente, hasta 4 KB de datos.
Accesible Desde	Tanto desde JavaScript como desde el servidor en cada petición HTTP.
Uso Principal	Autenticación, seguimiento de sesión, personalización basada en el servidor, etc.

> **Nota:** Tanto en Web Storage como en Cookies, **los datos se almacenan como cadenas de texto**. Por ello, para guardar datos complejos (objetos, arrays), es necesario **convertirlos a una cadena JSON** mediante `JSON.stringify` y, al recuperarlos, **convertirlos de vuelta a su forma original** con `JSON.parse`.

---

2. Transformación y Destransformación de Datos JSON

Cuando necesitas almacenar información compleja en localStorage, sessionStorage o incluso en cookies, debes:

- **Transformar (Serializar)**: Convertir el objeto o array a una cadena JSON usando `JSON.stringify`.
- **Destransformar (Parsear)**: Convertir la cadena JSON de vuelta a su objeto o array original usando `JSON.parse`.

Proceso de Transformación

1. **Almacenar datos complejos:**
   - Convierte el objeto a una cadena JSON.
   - Guarda la cadena en Web Storage o en una cookie.
2. **Recuperar datos complejos:**
   - Lee la cadena desde el almacenamiento.
   - Convierte la cadena JSON a objeto o array.

Ejemplo Básico en localStorage

```
// Objeto a almacenar

const usuario = {

  nombre: "María",

  edad: 28,

  intereses: ["música", "programación", "deportes"]

};


// Transformar el objeto a JSON y guardarlo en localStorage

localStorage.setItem("perfilUsuario", JSON.stringify(usuario));
```

```
// Recuperar y destransformar (parsear) el objeto almacenado

const perfilRecuperado =
JSON.parse(localStorage.getItem("perfilUsuario"));


console.log("Nombre:", perfilRecuperado.nombre);  // "María"

console.log("Intereses:", perfilRecuperado.intereses);
```

> **Importante:**
> Si se intenta usar `JSON.parse` sobre una cadena que no es un JSON válido, se lanzará un error. Por ello, es recomendable envolver la operación en un bloque `try...catch`.

Ejemplo con Manejo de Errores en JSON.parse

```
// Supongamos que recuperamos una cadena que podría no ser JSON
válido:

const datos = localStorage.getItem("datosInesperados");


try {

  const objeto = JSON.parse(datos);

  console.log("Datos parseados:", objeto);

} catch (error) {

  console.error("Error al parsear JSON:", error.message);

  // Aquí podrías definir una acción por defecto, como borrar la
entrada o notificar al usuario

}
```

---

3. Uso de Cookies con Datos JSON

Como las cookies solo almacenan cadenas, el proceso es muy similar al de Web Storage.

Guardar un Objeto en una Cookie

```
// Función para establecer una cookie con una clave, valor y fecha de expiración (en días)

function setCookie(nombre, valor, dias) {

  let fecha = new Date();

  fecha.setTime(fecha.getTime() + (dias * 24 * 60 * 60 * 1000));

  let expiracion = "expires=" + fecha.toUTCString();

  document.cookie = `${nombre}=${valor}; ${expiracion}; path=/`;

}


// Objeto a almacenar

const carrito = {

  productos: ["camisa", "pantalón", "zapatos"],

  total: 150.75

};


// Convertir el objeto a JSON y guardarlo en una cookie

setCookie("carritoCompras", JSON.stringify(carrito), 7);
```

Leer y Parsear Datos de una Cookie

```
// Función para obtener el valor de una cookie por su nombre

function getCookie(nombre) {

  const nombreEQ = nombre + "=";

  const cookies = document.cookie.split(';');
```

```
  for (let cookie of cookies) {
    cookie = cookie.trim();
    if (cookie.indexOf(nombreEQ) === 0) {
      return cookie.substring(nombreEQ.length, cookie.length);
    }
  }
  return null;
}

const carritoCookie = getCookie("carritoCompras");

try {
  const carritoObj = JSON.parse(carritoCookie);
  console.log("Carrito:", carritoObj);
} catch (error) {
  console.error("Error al parsear la cookie JSON:", error.message);
}
```

---

4. Buenas Prácticas y Consideraciones

- **Validar el JSON antes de parsear:**
  Siempre que recuperes datos que se supone están en formato JSON, usa un bloque `try...catch` para manejar posibles errores en caso de que la cadena no sea válida.
- **Verificar la existencia de datos:**
  Comprueba si la clave existe en localStorage, sessionStorage o en las cookies antes de intentar parsearla.

- **Manejo de datos corruptos:**
  Si el parseo falla, considera borrar la entrada o restaurar un valor por defecto para evitar que la aplicación se bloquee.
- **Uso consistente del formato:**
  Asegúrate de que, al guardar datos complejos, siempre se utilice `JSON.stringify` y al recuperar, `JSON.parse`.
- **Cookies y Seguridad:**
  Recuerda que las cookies se envían al servidor en cada petición HTTP, por lo que no es aconsejable almacenar información sensible en ellas sin aplicar métodos de seguridad (como cifrado).

---

5. Resumen

- **Web Storage (localStorage y sessionStorage)** y **Cookies** almacenan **cadenas de texto**; por ello, al guardar datos complejos, es esencial **transformarlos a JSON** usando `JSON.stringify`.
- Al recuperar dichos datos, se debe **destransformar** (parsear) la cadena JSON usando `JSON.parse`, envolviendo esta operación en un bloque `try...catch` para evitar errores.
- El manejo adecuado de la serialización y el parseo es fundamental para evitar errores inesperados y garantizar que la información se almacene y recupere correctamente.

1. Web Storage (localStorage y sessionStorage)

Estos objetos permiten almacenar datos en el navegador de forma sencilla. Los datos se guardan como **cadenas de texto**, por lo que al trabajar con objetos o arrays es necesario convertirlos a JSON.

Método/Propiedad	Descripción	Ejemplo

`setItem(key, value)`	Almacena un valor asociado a una clave. El valor debe ser una cadena (por eso, para objetos, se usa JSON.stringify).	`localStorage.setItem("usuario", "Juan");`
		
		`localStorage.setItem("perfil", JSON.stringify({ nombre: "Ana", edad: 25 }));`

`getItem(key)`	Recupera el valor asociado a una clave. Retorna `null` si la clave no existe.	`let usuario = localStorage.getItem("usuario");`
`removeItem(key)`	Elimina el elemento almacenado con la clave especificada.	`localStorage.removeItem("usuario");`
`clear()`	Elimina **todos** los datos almacenados en el objeto Storage.	`localStorage.clear();`
`key(index)`	Retorna la clave ubicada en la posición `index` del objeto Storage.	`let primeraClave = localStorage.key(0);`
`length`	Propiedad que indica la cantidad de elementos almacenados en el Storage.	`console.log(localStorage.length);`

Nota: La misma sintaxis y métodos se aplican a **sessionStorage**; la diferencia principal es la persistencia de los datos:

- **localStorage:** Los datos se mantienen hasta que se borran manualmente o mediante código.
- **sessionStorage:** Los datos se mantienen solo durante la sesión del navegador (se borran al cerrar la pestaña).

---

2. Operaciones Comunes con Cookies

Las cookies se gestionan a través de la propiedad `document.cookie`. No disponen de métodos nativos como Web Storage, por lo que se realizan operaciones mediante asignación y lectura de cadenas.

Operación	Descripción	Ejemplo
Crear/Actualizar	Se asigna una cookie especificando nombre, valor y opciones (como fecha de expiración, path, etc.).	`document.cookie = "usuario=Juan; expires=Fri, 31 Dec 9999 23:59:59 GMT; path=/";`
Leer Cookies	`document.cookie` retorna una cadena con todas las cookies en formato `clave=valor; clave2=valor2; ...`.	`console.log(document.cookie);`
Eliminar Cookie	Para borrar una cookie, se asigna la misma cookie con una fecha de expiración en el pasado.	`document.cookie = "usuario=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";`

Importante:

- Dado que las cookies se envían al servidor en cada petición, es recomendable **no almacenar información sensible** sin aplicar técnicas de seguridad (como el cifrado).
- La capacidad de las cookies es limitada (generalmente unos 4 KB).

---

3. Transformación y Destransformación de Datos JSON

Almacenar datos complejos (objetos, arrays) requiere **convertirlos a una cadena JSON** para su almacenamiento y luego **parsearlos** al recuperarlos. Esto es fundamental para evitar errores en el proceso.

Método	Descripción	Ejemplo

`JSON.stringify(valor)`	Convierte un objeto o array en una cadena JSON.	`let cadena = JSON.stringify({ nombre: "Ana", edad: 25 });`
`JSON.parse(cadena)`	Convierte una cadena JSON en el objeto o array original.	`let objeto = JSON.parse(cadena);`

Ejemplo Completo: Almacenando y Recuperando un Objeto

```
// Objeto a almacenar
const usuario = {
  nombre: "María",
  edad: 28,
  intereses: ["música", "programación", "deportes"]
};

// Transformar el objeto a JSON y guardarlo en localStorage
localStorage.setItem("perfilUsuario", JSON.stringify(usuario));

// Recuperar el dato y destransformarlo (parsear)
try {
  const perfilRecuperado = JSON.parse(localStorage.getItem("perfilUsuario"));
  console.log("Nombre:", perfilRecuperado.nombre);
  console.log("Intereses:", perfilRecuperado.intereses);
} catch (error) {
```

```
  console.error("Error al parsear JSON:", error.message);
}
```

---

4. Buenas Prácticas para Web Storage y Cookies

Buena Práctica	Descripción	Consecuencia de No Aplicarla
Validar la existencia de datos	Verificar que la clave existe antes de intentar parsear el valor.	Posible error de `JSON.parse` en caso de obtener `null` o datos no válidos.
Usar try...catch al parsear JSON	Manejar errores de parseo para evitar que la aplicación se rompa si la cadena no es un JSON válido.	La aplicación puede detenerse si se lanza un error inesperado.
Eliminar datos obsoletos o corruptos	Borrar o reemplazar datos que hayan fallado al parsear, para evitar comportamientos anómalos.	Datos corruptos pueden persistir y causar errores en la aplicación.
Mantener consistencia en el formato	Siempre usar `JSON.stringify` al guardar y `JSON.parse` al recuperar datos complejos.	Inconsistencias en los datos pueden llevar a errores difíciles de depurar.

Tema 9 programación asíncrona

1. Diferencia entre Programación Síncrona y Asíncrona

Característica	Síncrona	Asíncrona
Ejecución	Línea por línea, esperando a que cada instrucción termine antes de continuar.	Se ejecutan tareas en paralelo sin bloquear la ejecución principal.
Ejemplo	`console.log("Inicio"); let resultado = operacionLenta(); console.log("Fin");`	`console.log("Inicio"); setTimeout(() => console.log("Fin"), 2000);`
Bloqueo	Bloquea el hilo principal hasta que la tarea finaliza.	No bloquea el hilo principal; otras tareas pueden seguir ejecutándose.

Ejemplo en Código

```
// Código síncrono (bloqueante)
console.log("Inicio");
for (let i = 0; i < 1e9; i++) {} // Simula una tarea pesada
console.log("Fin");

// Código asíncrono (no bloqueante)
console.log("Inicio");
setTimeout(() => console.log("Fin"), 2000);
console.log("Esto se ejecuta antes que 'Fin'");
```

---

2. Callbacks y Callback Hell

Los **callbacks** son funciones pasadas como argumento a otras funciones para ser ejecutadas más tarde. Sin embargo, anidar múltiples callbacks genera un código difícil de leer, conocido como **callback hell**.

Ejemplo de Callback Hell

```
setTimeout(() => {
    console.log("Paso 1");
    setTimeout(() => {
        console.log("Paso 2");
        setTimeout(() => {
            console.log("Paso 3");
        }, 500);
    }, 1000);
```

```
}, 1500);
```

Problema: El código se vuelve difícil de mantener y leer.
Solución: Usar **Promesas** y **async/await**.

---

3. Promesas en JavaScript

Las **promesas** manejan la asincronía de manera más estructurada. Una promesa representa un valor que puede estar en uno de estos estados:

Estado	Descripción
Pending (Pendiente)	La operación asíncrona aún no ha finalizado.
Fulfilled (Resuelta)	La operación finalizó con éxito.
Rejected (Rechazada)	La operación falló.

Ejemplo Básico de Promesas

```
const promesa = new Promise((resolve, reject) => {
    setTimeout(() => {
        let exito = Math.random() > 0.5; // Simula éxito o fallo
aleatorio
        if (exito) resolve("Operación exitosa");
        else reject("Operación fallida");
    }, 2000);
});

promesa
    .then(resultado => console.log(resultado)) // Si la promesa se
resuelve
    .catch(error => console.error(error)) // Si la promesa falla
    .finally(() => console.log("Proceso terminado"));
```

---

4. Métodos Útiles en Promesas

Método	Descripción	Ejemplo

`Promise.resolve()`	Crea una promesa ya resuelta.	`Promise.resolve("Éxito").then(console.log);`
`Promise.reject()`	Crea una promesa rechazada.	`Promise.reject("Error").catch(console.error);`
`Promise.all()`	Ejecuta varias promesas en paralelo y espera a que todas finalicen.	`Promise.all([p1, p2]).then(console.log);`
`Promise.race()`	Devuelve el resultado de la primera promesa que termine.	`Promise.race([p1, p2]).then(console.log);`

Ejemplo de `Promise.all`

```
const p1 = new Promise(res => setTimeout(() => res("Promesa 1"), 1000));
const p2 = new Promise(res => setTimeout(() => res("Promesa 2"), 2000));
const p3 = new Promise(res => setTimeout(() => res("Promesa 3"), 1500));

Promise.all([p1, p2, p3]).then(console.log); // Espera a que todas terminen
```

---

5. Async/Await

Las funciones `async` y `await` permiten manejar promesas de manera más clara y secuencial.

Característica	Descripción
`async`	Define una función que devuelve una promesa automáticamente.
`await`	Pausa la ejecución de la función `async` hasta que la promesa se resuelva.

Ejemplo de `async/await`

```
async function obtenerDatos() {
```

```
    try {
        let respuesta = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
        let datos = await respuesta.json();
        console.log(datos);
    } catch (error) {
        console.error("Error:", error);
    }
}

obtenerDatos();
```

---

6. AJAX y Fetch API

AJAX permite hacer peticiones HTTP sin recargar la página. La API `fetch` es la forma moderna de hacer AJAX en JavaScript.

Método Fetch	Descripción
`fetch(url)`	Realiza una solicitud GET a la URL indicada.
`.then(res => res.json())`	Convierte la respuesta a JSON.
`.catch(error => console.error(error))`	Captura errores de red.

Ejemplo de Fetch

```
fetch("https://jsonplaceholder.typicode.com/posts")
    .then(res => res.json())
    .then(data => console.log(data))
    .catch(error => console.error("Error:", error));
```

---

7. Envío de Datos con Fetch

Podemos enviar datos mediante `fetch` usando el método `POST`.

```
const datos = { nombre: "Carlos", edad: 30 };

fetch("https://jsonplaceholder.typicode.com/posts", {
```

```
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(datos)
})
.then(res => res.json())
.then(data => console.log("Respuesta:", data))
.catch(error => console.error("Error:", error));
```

---

8. Resumen

- **JavaScript es asíncrono**, lo que significa que puede ejecutar múltiples tareas sin bloquear la ejecución principal.
- **Los callbacks** permiten manejar tareas asíncronas, pero pueden llevar a un **callback hell**.
- **Las promesas** (`Promise`) ayudan a manejar la asincronía de forma más estructurada.
- **Async/Await** simplifica la sintaxis de las promesas y las hace más legibles.
- **Fetch API** permite realizar peticiones HTTP de manera sencilla.

1. ¿Qué es Fetch API?

La **Fetch API** es la forma moderna de hacer peticiones HTTP en JavaScript. Permite recuperar recursos de una URL de manera asíncrona y devuelve una **promesa** que se resuelve con un objeto `Response`.

🔹 **Ventajas de Fetch API:**
✔ Más fácil de usar que `XMLHttpRequest`.
✔ Compatible con `async/await`.
✔ Manejo de promesas para tratar respuestas.
✔ Permite enviar y recibir datos en diferentes formatos (JSON, texto, Blob, etc.).

---

2. Métodos HTTP en Fetch

Fetch permite hacer peticiones con diferentes **métodos HTTP**:

Método HTTP	Descripción
GET	Recupera datos de un servidor. Es el método por defecto.

POST	Envía datos al servidor para ser procesados.
PUT	Actualiza un recurso existente en el servidor.
DELETE	Elimina un recurso en el servidor.

---

3. Propiedades del Objeto Response

Cuando hacemos una petición con Fetch, obtenemos un **objeto Response**, que contiene varias propiedades útiles.

Propiedad	Descripción	Ejemplo
`status`	Código de estado HTTP de la respuesta (ej. 200, 404, 500).	`console.log(response.status);`
`ok`	`true` si el estado es entre 200-299, `false` si hay error.	`if (response.ok) { ... }`
`headers`	Accede a los encabezados de la respuesta.	`console.log(response.headers.get('Content-Type'));`
`json()`	Convierte la respuesta a formato JSON.	`const data = await response.json();`
`text()`	Convierte la respuesta en texto plano.	`const text = await response.text();`
`blob()`	Convierte la respuesta en un objeto Blob (archivos, imágenes).	`const blob = await response.blob();`

---

4. Ejemplo de Fetch con Método GET

El siguiente código recupera datos de una API pública usando **async/await**.

```
async function obtenerDatos() {
    try {
        const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts/1");
```

```
        if (!respuesta.ok) throw new Error("Error en la petición: "
+ respuesta.status);

        const datos = await respuesta.json();
        console.log("Datos obtenidos:", datos);
    } catch (error) {
        console.error("Error en Fetch:", error);
    }
}

obtenerDatos();
```

✅ **Explicación:**

- `fetch(url)`: Realiza la petición a la URL.
- `respuesta.ok`: Verifica si la respuesta es correcta.
- `await respuesta.json()`: Convierte la respuesta en JSON.
- `catch(error)`: Captura errores de conexión o respuesta.

---

5. Ejemplo de Fetch con Método POST

Podemos enviar datos al servidor usando **POST** con el encabezado `Content-Type: application/json`.

```
async function enviarDatos() {
    const datos = {
        nombre: "Juan",
        edad: 30
    };

    try {
        const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(datos)
        });
```

```
        if (!respuesta.ok) throw new Error("Error en la petición: "
+ respuesta.status);

        const resultado = await respuesta.json();
        console.log("Datos enviados:", resultado);
    } catch (error) {
        console.error("Error en Fetch:", error);
    }
}

enviarDatos();
```

✅ **Explicación:**

- `method: "POST"`: Indica el método de la petición.
- `headers`: Define el tipo de contenido enviado.
- `body: JSON.stringify(datos)`: Convierte el objeto en JSON.

---

6. Otros Métodos: PUT y DELETE

Actualizar un recurso con PUT

```
async function actualizarDatos() {
    const nuevosDatos = { nombre: "Carlos", edad: 35 };

    try {
        const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts/1", {
            method: "PUT",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(nuevosDatos)
        });

        if (!respuesta.ok) throw new Error("Error en la petición: "
+ respuesta.status);

        const resultado = await respuesta.json();
        console.log("Datos actualizados:", resultado);
    } catch (error) {
        console.error("Error en Fetch:", error);
```

```
    }
}

actualizarDatos();
```

✅ **Diferencia con POST:**

- `PUT` actualiza un recurso existente, mientras que `POST` crea uno nuevo.

Eliminar un recurso con DELETE

```
async function eliminarDato(id) {
    try {
        const respuesta = await
fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {
            method: "DELETE"
        });

        if (!respuesta.ok) throw new Error("Error en la petición: "
+ respuesta.status);

        console.log(`Recurso con ID ${id} eliminado
correctamente.`);
    } catch (error) {
        console.error("Error en Fetch:", error);
    }
}

eliminarDato(1);
```

✅ **Explicación:**

- `method: "DELETE"` indica que queremos eliminar un recurso.
- La URL incluye el `id` del recurso a eliminar.

---

7. Manejo de Errores en Fetch

Cuando trabajamos con `fetch`, es importante **manejar errores correctamente**, como **errores de red**, **errores del servidor** y **respuestas vacías**.

Ejemplo de Manejo de Errores

```
async function obtenerUsuarios() {
    try {
        const respuesta = await
fetch("https://jsonplaceholder.typicode.com/users");

        if (!respuesta.ok) {
            throw new Error(`Error HTTP: ${respuesta.status}`);
        }

        const datos = await respuesta.json();
        console.log("Usuarios:", datos);
    } catch (error) {
        console.error("Error en Fetch:", error);
    } finally {
        console.log("Finalizó la petición.");
    }
}

obtenerUsuarios();
```

✅ **Explicación:**

- `throw new Error(...)` lanza un error si la petición falla.
- `finally` se ejecuta siempre, ya sea éxito o error.

---

8. Resumen de Fetch API con Async/Await

Método HTTP	Función	Ejemplo
`GET`	Recuperar datos	`fetch(url).then(res => res.json())`
`POST`	Enviar datos al servidor	`fetch(url, { method: "POST", body: JSON.stringify(datos) })`
`PUT`	Actualizar un recurso existente	`fetch(url, { method: "PUT", body: JSON.stringify(datos) })`
`DELETE`	Eliminar un recurso	`fetch(url, { method: "DELETE" })`

---

9. Conclusión

- **Fetch API** es la forma moderna de hacer peticiones HTTP en JavaScript.
- **Métodos HTTP** como **GET, POST, PUT, DELETE** permiten recuperar y modificar datos.
- **`async/await`** simplifica la escritura de código asíncrono, haciéndolo más legible.
- **Manejo de errores** con `try...catch` es esencial para evitar fallos inesperados.

1. Configuración de Fetch API

Cuando usamos `fetch()`, podemos pasar un **segundo parámetro opcional** con un objeto de configuración. Esto nos permite definir detalles como **el método HTTP, los encabezados (headers), el cuerpo de la petición (body), entre otros**.

Sintaxis General de Fetch con Configuración

```
fetch(url, {
    method: "GET",          // Método HTTP
    headers: {              // Encabezados de la petición
        "Content-Type": "application/json"
    },
    body: JSON.stringify({ clave: "valor" }), // Datos a enviar
    mode: "cors",          // Modo de la petición (cors, no-cors,
same-origin)
    credentials: "include" // Cookies y autenticación
})
.then(res => res.json())
.then(data => console.log(data))
.catch(error => console.error("Error:", error));
```

---

2. Propiedades del Objeto de Configuración en Fetch

Propiedad	Descripción
`method`	Define el método HTTP (`GET`, `POST`, `PUT`, `DELETE`, etc.).
`headers`	Especifica los **encabezados HTTP** que acompañan la petición.

`body`	Contiene los datos a enviar en **POST, PUT o PATCH** (no se usa en GET o DELETE).
`mode`	Controla la política de **CORS** para la petición (`cors`, `no-cors`, `same-origin`).
`credentials`	Controla si se incluyen **cookies** y autenticación (`omit`, `same-origin`, `include`).
`cache`	Define la estrategia de caché (`default`, `no-store`, `reload`, `force-cache`).
`redirect`	Controla el comportamiento ante redirecciones (`follow`, `error`, `manual`).
`referrer`	Especifica la URL de referencia (`no-referrer`, `client`, una URL específica).
`signal`	Se usa para **abortar** una petición fetch mediante `AbortController`.

---

3. Métodos HTTP en Fetch

Método	Descripción
`GET`	Recupera datos de un servidor.
`POST`	Envía datos al servidor.
`PUT`	Reemplaza completamente un recurso en el servidor.
`PATCH`	Modifica parcialmente un recurso existente.
`DELETE`	Elimina un recurso del servidor.

Ejemplo de una Petición POST con Fetch

```
async function enviarDatos() {
    const datos = { nombre: "Carlos", edad: 28 };

    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
```

```
        },
        body: JSON.stringify(datos)
    });

    const resultado = await respuesta.json();
    console.log("Datos enviados:", resultado);
}

enviarDatos();
```

✅ **Explicación:**

- `method: "POST"` → Especifica que estamos enviando datos.
- `headers` → Indica que enviamos JSON con `Content-Type: application/json`.
- `body: JSON.stringify(datos)` → Convierte el objeto en JSON antes de enviarlo.

---

4. Headers en Fetch

Los **headers** permiten enviar información adicional en una petición HTTP. Se configuran en la propiedad `headers` del objeto de configuración.

4.1. Headers Comunes

Header	Descripción
`Content-Type`	Indica el tipo de contenido enviado (`application/json`, `text/html`, etc.).
`Authorization`	Se usa para autenticación (ejemplo: `Bearer token`).
`Accept`	Indica qué tipo de respuesta espera el cliente (`application/json`).
`User-Agent`	Identifica el tipo de cliente que realiza la petición.
`Cache-Control`	Controla el almacenamiento en caché (`no-cache`, `max-age=3600`).

4.2. Ejemplo de Fetch con Headers Personalizados

```
async function obtenerDatos() {
    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts/1", {
        method: "GET",
        headers: {
            "Content-Type": "application/json",
            "Authorization": "Bearer 1234567890",
            "Accept-Language": "es-ES"
        }
    });

    const datos = await respuesta.json();
    console.log("Datos obtenidos:", datos);
}

obtenerDatos();
```

✅ **Explicación:**

- `Authorization: "Bearer 1234567890"` → Se usa para autenticación con token.
- `Accept-Language: "es-ES"` → Indica que preferimos respuestas en español.

---

5. Control de Caché con Fetch

Fetch permite definir cómo manejar el almacenamiento en caché con la propiedad `cache`.

Valor	Descripción
`default`	Usa la caché del navegador si está disponible.
`no-store`	Siempre obtiene una nueva respuesta, sin almacenar en caché.
`reload`	Fuerza la recarga de datos desde el servidor.
`force-cache`	Siempre usa la caché, incluso si los datos están desactualizados.

Ejemplo de Fetch con Control de Caché

```
fetch("https://jsonplaceholder.typicode.com/posts", { cache: "no-store" })
    .then(res => res.json())
    .then(data => console.log("Datos obtenidos sin caché:", data))
    .catch(error => console.error("Error:", error));
```

---

6. Autenticación y Cookies en Fetch

La propiedad `credentials` define si la petición incluye **cookies o autenticación**.

Valor	Descripción
`omit`	No envía cookies ni credenciales.
`same-origin`	Solo envía cookies si la URL pertenece al mismo origen.
`include`	Siempre envía cookies, incluso a dominios distintos.

Ejemplo de Fetch con Cookies

```
fetch("https://api.ejemplo.com/datos", { credentials: "include" })
    .then(res => res.json())
    .then(data => console.log("Datos obtenidos:", data))
    .catch(error => console.error("Error:", error));
```

✅ **Explicación:**

- `credentials: "include"` → Asegura que se envíen cookies en la petición.

---

7. Abortar una Petición Fetch

Podemos **cancelar** una petición Fetch con `AbortController`.

Ejemplo de Cancelación de Petición

```
const controlador = new AbortController();
```

```
fetch("https://jsonplaceholder.typicode.com/posts", { signal: controlador.signal })
    .then(res => res.json())
    .then(data => console.log("Datos obtenidos:", data))
    .catch(error => console.error("Petición abortada:", error));

// Cancelar la petición después de 1 segundo
setTimeout(() => controlador.abort(), 1000);
```

✅ **Explicación:**

- `AbortController` permite **cancelar una petición en curso**.

---

8. Resumen de Configuración de Fetch

Propiedad	Uso
`method`	Define el método HTTP (`GET`, `POST`, `PUT`, `DELETE`).
`headers`	Agrega encabezados personalizados.
`body`	Define los datos a enviar en POST/PUT.
`mode`	Controla CORS (`cors`, `no-cors`, `same-origin`).
`credentials`	Controla envío de cookies (`omit`, `same-origin`, `include`).
`cache`	Controla almacenamiento en caché (`default`, `no-store`, `reload`).
`signal`	Permite abortar una petición con `AbortController`.

1. Content-Type en Fetch API

El encabezado `Content-Type` indica **el formato de los datos** que se están enviando o esperando recibir en una petición HTTP. Es fundamental cuando enviamos datos con `POST`, `PUT` o `PATCH`.

Content-Type	Descripción
`application/json`	Envío de datos en formato JSON (más común en APIs modernas).
`application/x-www-form-urlencoded`	Envío de datos como un formulario HTML (`key=value&key2=value2`).
`multipart/form-data`	Para enviar archivos e imágenes.
`text/plain`	Envío de datos como texto sin formato.
`application/xml`	Envío de datos en formato XML.

---

2. Formas de Enviar el `body` en Fetch

Dependiendo del tipo de datos que queremos enviar, el `body` de la petición varía.

Formato	Ejemplo de Configuración en Fetch
JSON	`body: JSON.stringify({nombre: "Juan", edad: 30})`
Form Data (Archivos/Imágenes)	`const formData = new FormData(); formData.append("archivo", file); body: formData`
URL Encoded (Como Formularios)	`body: new URLSearchParams({nombre: "Juan", edad: "30"})`
Texto Plano	`body: "Este es un mensaje de prueba"`

---

3. Ejemplos Prácticos

Veamos ejemplos donde enviamos y recibimos datos en diferentes formatos.

3.1. Enviar y Recibir JSON

```
async function enviarDatosJSON() {
```

```
    const datos = { nombre: "Carlos", edad: 28 };

    try {
        const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(datos)
        });

        const resultado = await respuesta.json();
        console.log("Respuesta del servidor:", resultado);
    } catch (error) {
        console.error("Error en Fetch:", error);
    }
}

enviarDatosJSON();
```

✅ **Explicación**:

- `Content-Type: "application/json"` → Indica que estamos enviando JSON.
- `JSON.stringify(datos)` → Convierte el objeto en JSON antes de enviarlo.

3.2. Enviar Datos como Formulario URL-Encoded

```
async function enviarFormulario() {
    const datos = new URLSearchParams();
    datos.append("usuario", "maria");
    datos.append("clave", "1234");

    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts", {
        method: "POST",
        headers: { "Content-Type":
"application/x-www-form-urlencoded" },
        body: datos
    });

    const resultado = await respuesta.json();
```

```
    console.log("Respuesta del servidor:", resultado);
}

enviarFormulario();
```

✅ **Explicación**:

- `Content-Type: "application/x-www-form-urlencoded"` → Enviamos datos en formato formulario.
- `new URLSearchParams()` → Construye la cadena `key=value&key2=value2`.

3.3. Enviar Archivos con `multipart/form-data`

```
async function enviarArchivo() {
    const formData = new FormData();
    const fileInput = document.querySelector("#archivo");

    formData.append("imagen", fileInput.files[0]);

    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts", {
        method: "POST",
        body: formData
    });

    const resultado = await respuesta.json();
    console.log("Respuesta del servidor:", resultado);
}

document.querySelector("#btnEnviar").addEventListener("click",
enviarArchivo);
```

✅ **Explicación**:

- **No se necesita `Content-Type`** porque `fetch` lo define automáticamente con `multipart/form-data`.
- `formData.append("imagen", file)` → Adjunta el archivo a la petición.

📌 **HTML para el ejemplo**:

```
<input type="file" id="archivo">
<button id="btnEnviar">Subir Archivo</button>
```

3.4. Recibir Respuesta en Diferentes Formatos

```
async function obtenerDatos() {
    const respuesta = await
fetch("https://jsonplaceholder.typicode.com/posts/1");

    if (!respuesta.ok) throw new Error("Error en la petición");

    const contenidoJSON = await respuesta.json();
    console.log("JSON recibido:", contenidoJSON);

    const textoPlano = await respuesta.text();
    console.log("Texto recibido:", textoPlano);
}

obtenerDatos();
```

✅ **Explicación**:

- `.json()` → Convierte la respuesta en JSON.
- `.text()` → Convierte la respuesta en texto plano.

---

4. Comparación de Métodos de Envío de Datos

Formato	Content-Type	Método de Envío	Usado Para
JSON	`application/json`	`JSON.stringify(objeto)`	Enviar datos estructurados a APIs
Form Data	`multipart/form-data`	`formData.append("clave", valor)`	Subir archivos e imágenes
URL Encoded	`application/x-www-form-urlencoded`	`new URLSearchParams()`	Formularios HTML

Texto Plano	`text/plain`	`body: "Texto aquí"`	Mensajes simples o logs

---

5. Conclusión

- **El `Content-Type`** es fundamental para indicar el formato de los datos que enviamos.
- **El `body` varía según el tipo de contenido**:
  - **JSON** con `JSON.stringify()`.
  - **Form Data** con `FormData()`.
  - **URL Encoded** con `URLSearchParams()`.
- **`fetch()` permite recibir datos en diferentes formatos** (`json()`, `text()`, `blob()`).
- **Para enviar archivos** usamos `multipart/form-data`, sin necesidad de definir `Content-Type`.

Pasos para usar JSON Server:

1. Instalar JSON Server: Asegúrate de tener Node.js instalado y luego ejecuta:

2. `npm install -g json-server`
3. Crear un archivo `db.json`: Este archivo actuará como la base de datos falsa. Por ejemplo:

4. `{ "usuarios": [] }`
5. Iniciar JSON Server: En el directorio donde está `db.json`, ejecuta:

6. `json-server --watch db.json --port 3000`
7. Esto iniciará un servidor en `http://localhost:3000`. Las solicitudes POST a `http://localhost:3000/usuarios` agregarán datos al array `usuarios`.
8. Probar tu código: Cambia la URL en tu código `fetch` a:
   ```
   const respuesta = await fetch("http://localhost:3000/usuarios", {
     method: "POST",
     headers: {
       "Content-Type": "application/json", // Trailing comma aquí
     },
     body: JSON.stringify(datosUsuario), // Trailing comma aquí
   });
   ```

   Cada vez que envíes datos, se agregarán al archivo `db.json`.

Ejemplo de abortController

```
<script>
        const formulario = document.getElementById('miFormulario');
        const cancelarBtn = document.getElementById('cancelar');
        const resultadoDiv = document.getElementById('resultado');

        let controller; // Para poder reutilizarlo y cancelarlo después

        formulario.addEventListener('submit', async function (e) {
            e.preventDefault();

            // Si hay una solicitud anterior, la cancelamos
            if (controller) {
                controller.abort();
            }

            // Creamos un nuevo controlador para esta petición
            controller = new AbortController();
            const signal = controller.signal;

            // Obtenemos los datos del formulario
            const formData = new FormData(formulario);
            const datos = Object.fromEntries(formData.entries());

            try {
                const respuesta = await fetch('https://jsonplaceholder.typicode.com/posts', {
                    method: 'POST',
                    headers: {
                        'Content-Type': 'application/json'
                    },
                    body: JSON.stringify(datos),
                    signal, // Asociamos el controlador para poder abortar la petición
                });
```

Ejemplo de cookies

```
// Función para establecer cookies
function setCookie(nombre, valor, minutos) {
    const fecha = new Date();
    fecha.setTime(fecha.getTime() + (minutos * 60 * 1000));
    const expira = "expires=" + fecha.toUTCString();
    document.cookie = `${nombre}=${valor};${expira};path=/`;
}

// Función para obtener cookies
function getCookie(nombre) {
```

```
    const nombreEQ = nombre + "=";
    const cookies = document.cookie.split(';');
    for (let i = 0; i < cookies.length; i++) {
        let c = cookies[i].trim();
        if (c.indexOf(nombreEQ) == 0) {
            return c.substring(nombreEQ.length, c.length);
        }
    }
    return null;
}

function updateCookieStatus() {
    const consentValue = getCookie('cookieConsent');
    cookieStatus.textContent = consentValue ? consentValue : 'No establecida';
}

function deleteCookie(nombre) {
    document.cookie = `${nombre}=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/`;
}

function deleteAllCookies(){
    const cookies = document.cookie.split(';');
    cookies.forEach(cookie => {
        const cookieName = cookie.split('=')[0].trim();
        deleteCookie(cookieName);
    });
}
```