

## JavaScript moderno :

**Una función callback** en JavaScript es una función que se pasa como argumento a otra función, diseñada para ser ejecutada después de que la función externa termine su tarea. Son fundamentales en la programación asíncrona (como setTimeout, fetch o eventos) para manejar operaciones que no se completan inmediatamente.

### Ejemplo Básico de Callback:

```
// Función principal que acepta un callback
function procesarUsuario(nombre, callback) {
    console.log("Procesando usuario: " + nombre);
    callback(nombre); // Ejecuta el callback
}

// Función callback
function saludar(nombre) {
    console.log("Hola, " + nombre);
}

// Llamada con el callback como argumento
procesarUsuario("Ana", saludar);

// Salida:
// Procesando usuario: Ana
// Hola, Ana
```

### Conceptos Clave

- **Ejecución diferida:** El callback no se ejecuta inmediatamente, sino cuando el proceso principal lo decide (a menudo al finalizar).
- **Funciones de Orden Superior:** La función que recibe el callback se denomina así.
- **Uso común:** Manejo de eventos (addEventListener), temporizadores (setTimeout), y peticiones asíncronas.

### Ejemplo asíncrono (setTimeout):

```
console.log("Inicio");
setTimeout(() => {
    console.log("Callback ejecutado tras 2 segundos");
}, 2000);
console.log("Fin");
// Orden de salida: Inicio -> Fin -> Callback...
```

Los callbacks permiten una mayor flexibilidad en el código al permitirnos definir la acción a realizar una vez concluida una tarea, siendo un pilar fundamental en el desarrollo con JavaScript.

**Funciones Flecha (=>):** Definen funciones de forma más corta y legible. **No crean su propio this**, por eso son ideales para React.

**Sintaxis:** const fn = (param1, param2) => resultado;

#### Parámetros

- param1, param2: valores de entrada

#### Resultado

- Devuelve automáticamente el resultado (return implícito)

### Ejemplo

```
const sumar = (a, b) => a + b;
```

## FUNCIONES PREDETERMINADAS

### LA FUNCIÓN eval()

- **Propósito:** Evalúa una cadena de texto como código JavaScript y ejecuta ese código.
- **Funcionamiento:** Recibe un string y lo interpreta como si fuera código escrito directamente en el programa.
- **Retorno:** Devuelve el resultado de la última expresión evaluada.
- **Advertencia:** Su uso es peligroso si se evalúa contenido externo o no confiable, ya que puede ejecutar código malicioso. No se recomienda en aplicaciones modernas.

#### Análisis de eval("2 + 3")

1. **Entrada:** "2 + 3" es una cadena.
2. **Procesamiento:** eval interpreta la cadena como código JavaScript.
3. **Salida:** 5

```
console.log(eval("2 + 3")); // Resultado: 5
```

### LA FUNCIÓN isFinite()

- **Propósito:** Determina si un valor es un número finito.
- **Funcionamiento:** Convierte el valor a número y verifica que:
  - No sea Infinity
  - No sea -Infinity
  - No sea NaN
- **Retorno:** Devuelve true si es un número finito válido, false en caso contrario.

#### Análisis de isFinite("10")

1. **Entrada:** "10" es una cadena.
2. **Procesamiento:** Se convierte a número → 10.
3. **Salida:** true

```
console.log(isFinite("10")); // true  
console.log(isFinite(Infinity)); // false  
console.log(isFinite("hola")); // false
```

### LA FUNCIÓN isNaN()

- **Propósito:** Determina si un valor es NaN (Not a Number).
- **Funcionamiento:** Convierte el valor a número antes de evaluarlo.
- **Retorno:**
  - true si el valor no puede convertirse en número.
  - false si es un número válido.

#### Análisis de isNaN("abc")

1. **Entrada:** "abc" es una cadena.
2. **Procesamiento:** Se intenta convertir a número → NaN.
3. **Salida:** true

```
console.log(isNaN("abc")); // true  
console.log(isNaN("123")); // false  
console.log(isNaN(123)); // false
```

### LAS FUNCIONES parseInt() Y parseFloat()

#### parseInt()

- **Propósito:** Convierte una cadena en un número entero.
- **Funcionamiento:** Lee la cadena hasta encontrar un carácter no numérico.
- **Parámetro adicional:** Puede recibir una base (radix), como 10 (decimal) o 16 (hexadecimal).

## Análisis de parseInt("123px")

1. **Entrada:** "123px"
2. **Procesamiento:** Lee números hasta la letra p.
3. **Salida:** 123

```
console.log(parseInt("123px")); // 123  
console.log(parseInt("101", 2)); // 5
```

## parseFloat()

- **Propósito:** Convierte una cadena en un número decimal.
- **Funcionamiento:** Lee números incluyendo punto decimal hasta encontrar un carácter inválido.

## Análisis de parseFloat("12.5em")

1. **Entrada:** "12.5em"
2. **Procesamiento:** Lee hasta la letra e.
3. **Salida:** 12.5

```
console.log(parseFloat("12.5em")); // 12.5
```

## LA FUNCIÓN encodeURI()

- **Propósito:** Codifica una URI completa para que sea válida en la web.
- **Funcionamiento:** Reemplaza caracteres no permitidos por su representación en formato UTF-8 hexadecimal.
- **No codifica:** Caracteres especiales válidos en una URL como :, /, ?, &, #.

### Ejemplo:

```
const url = "https://ejemplo.com/mi pagina";  
console.log(encodeURI(url));  
// https://ejemplo.com/mi%20pagina
```

## LA FUNCIÓN decodeURI()

- **Propósito:** Decodifica una URI previamente codificada con encodeURI().
- **Funcionamiento:** Convierte los códigos %XX nuevamente en caracteres normales.

```
const url = "https://ejemplo.com/mi%20pagina";  
console.log(decodeURI(url));  
// https://ejemplo.com/mi pagina
```

## LA FUNCIÓN encodeURIComponent()

- **Propósito:** Codifica partes individuales de una URI (por ejemplo, parámetros).
- **Diferencia clave:** Codifica más caracteres que encodeURI(), incluyendo &, =, ?.

### Ejemplo:

```
const parametro = "nombre=Juan Pérez";  
console.log(encodeURIComponent(parametro));  
// nombre%3DJuan%20P%C3%A9rez
```

## LA FUNCIÓN decodeURIComponent()

- **Propósito:** Decodifica una cadena codificada con encodeURIComponent().

```
const parametro = "nombre%3DJuan%20P%C3%A9rez";  
console.log(decodeURIComponent(parametro));  
// nombre=Juan Pérez
```

## ARRAYS CON JAVASCRIPT

**INTRODUCCIÓN:** Un **array** es una estructura de datos que permite almacenar múltiples valores en una sola variable.

- 1) Puede contener: Números, Cadenas de texto, Objetos, Funciones, Otros arrays
- 2) Son dinámicos (pueden crecer o reducirse)

## Creación de Arrays

```
let numeros = [1, 2, 3];
let datos = new Array(1, 2, 3);
Recomendado: usar []
```

**EL OBJETO Array():** Array es un objeto global que proporciona métodos para manipular listas.

Propiedades importantes:

numeros.length // Cantidad de elementos

✓ Los índices comienzan en 0

✓ El último índice es length - 1

## Funciones predeterminadas de arrays:

**El método map():** en JavaScript crea un nuevo array aplicando una función a cada elemento de uno existente, sin modificar el original [1, 5]. $n \Rightarrow n * 2$ es una función flecha que toma un elemento `n` y devuelve su doble. Se recorre el array, cada número se multiplica por 2 y se guarda en un nuevo array.

### Ejemplo

```
const nums = [1, 2, 3];
const dobles = nums.map(n => n * 2);
// [2, 4, 6] // Usadísimo en React para renderizar listas.
```

Aquí está el desglose detallado paso a paso:

#### 1. ¿Cómo funciona map()?

map() es una función de orden superior diseñada para la transformación de datos.

- **Recorrido Automático:** Itera sobre cada elemento de la Arrays(vector o matriz) en orden.
- **Transformación:** Ejecuta la función proporcionada ( $n \Rightarrow n * 2$ ) en cada elemento.
- **Nuevo Array:** Almacena los resultados de cada ejecución en un nuevo array.
- **Inmutabilidad:** El array original no se modifica.

#### 2. Explicación detallada: $n \Rightarrow n * 2$

Esta es una función flecha (arrow function) simplificada que sirve como "callback" (función de retorno) para map.

- **n:** Representa el elemento actual que se está procesando del array.
- **=>:** Es el operador de flecha que separa los parámetros de entrada del cuerpo de la función.
- **$n * 2$ :** Es la expresión a ejecutar. Al no usar llaves {}, el resultado se devuelve automáticamente (retorno implícito). Multiplica el elemento **n** por 2.

#### 3. Funcionamiento paso a paso (Ejemplo: [1, 2, 3].map(n => n \* 2))

tenemos const nums = [1, 2, 3]; y ejecutamos const doubles = nums.map(n => n \* 2):

1. **Inicio:** map lee el primer elemento: 1.
2. **Ejecución:** Aplica  $1 \Rightarrow 1 * 2$ . El resultado es 2.
3. **Almacenamiento:** El nuevo array empieza: [2].
4. **Iteración 2:** map lee el segundo elemento: 2.
5. **Ejecución:** Aplica  $2 \Rightarrow 2 * 2$ . El resultado es 4.
6. **Almacenamiento:** El nuevo array va: [2, 4].
7. **Iteración 3:** map lee el tercer elemento: 3.
8. **Ejecución:** Aplica  $3 \Rightarrow 3 * 2$ . El resultado es 6.
9. **Almacenamiento:** El nuevo array finaliza: [2, 4, 6].
10. **Resultado:** Se devuelve [2, 4, 6] y nums sigue siendo [1, 2, 3].

## Resumen Visual:

- a) [ 1, 2, 3] // Array original
- b) [1\*2, 2\*2, 3\*2] // n => n\*2
- c) [ 2, 4, 6] // Nuevo Array

## MÉTODOS PARA GENERAR NUEVAS MATRICES O BUSCAR DATOS

**El método splice():** Es la "navaja suiza" de los arrays. Permite cambiar el contenido de un array eliminando elementos existentes y/o agregando nuevos elementos.

### Ejemplo

```
let a = [1, 2, 3];
a.splice(1, 1);
// [1, 3]
```

**Nota Crítica:** Este método **modifica el array original** (es destructivo).

### Desglose paso a paso:

#### 1. ¿Cómo funciona splice()?

- **Eliminación:** El primer parámetro es el índice de inicio; el segundo es cuántos elementos borrar.
- **Inserción:** Los parámetros adicionales (si existen) se insertan en esa posición.
- **Retorno:** Devuelve un array con los elementos eliminados.

#### 2. Funcionamiento paso a paso ([1, 2, 3].splice(1, 1)):

- **Inicio:** Se sitúa en el índice 1 (donde está el 2).
- **Corte:** Elimina 1 elemento a partir de ahí.
- **Resultado:** El array original queda como [1, 3].

**El método indexOf():** Busca un elemento en el array y devuelve el **primer índice** (posición) donde lo encuentra. Si no existe, devuelve -1.

### Ejemplo

```
const nums = [1, 2, 3];
nums.indexOf(2);
// 1
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona indexOf()?

- **Búsqueda Lineal:** Recorre el array de izquierda a derecha.
- **Comparación Estricta:** Usa el operador === (debe ser el mismo tipo de dato).
- **Sentido:** Se detiene en la primera coincidencia que encuentra.

#### 2. Funcionamiento paso a paso ([1, 2, 3].indexOf(2)):

- **Índice 0:** ¿Es 1 === 2? No.
- **Índice 1:** ¿Es 2 === 2? Sí.
- **Resultado:** Devuelve 1.

**El método lastIndexOf():** Funciona igual que indexOf(), pero busca el elemento empezando desde el **final** del array hacia el principio.

### Ejemplo

```
const nums = [1, 2, 2, 3];
nums.lastIndexOf(2);
// 2
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona lastIndexOf()?

- **Búsqueda Inversa:** Ideal para encontrar la ocurrencia más reciente de un dato.
- **Posicionamiento:** Devuelve el índice real (contando siempre desde 0), aunque lo busque de atrás hacia adelante.

## 2. Funcionamiento paso a paso ([1, 2, 2, 3].lastIndexOf(2)):

- **Índice 3:** ¿Es 3 === 2? No.
- **Índice 2:** ¿Es 2 === 2? Sí.
- **Resultado:** Devuelve 2.

**4. El método Array.isArray():** Es un método estático (se llama desde Array, no desde la variable) que determina si el valor pasado es un Array.

### Ejemplo

```
Array.isArray([1, 2]);  
// true  
Array.isArray("Hola");  
// false
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona?

- **Validación de Tipo:** En JS, los arrays técnicamente son objetos. Este método es la única forma 100% segura de saber si algo es una lista.
- **Seguridad:** Muy usado antes de aplicar métodos como map() para evitar errores si el dato no es lo que esperábamos.

**El método toString():** Devuelve una cadena de caracteres (string) que representa al array y sus elementos.

### Ejemplo

```
const lista = [1, 2, 3];  
lista.toString();  
// "1,2,3"
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona?

- **Conversión:** Llama internamente al método join() usando una coma como separador por defecto.
- **Simplificación:** Aplana el array en un solo bloque de texto legible.

#### 2. Funcionamiento paso a paso ([1, 2, 3].toString()):

- **Lectura:** Toma los elementos 1, 2 y 3.
- **Unión:** Los concatena poniendo comas entre ellos.
- **Resultado:** Genera el string "1,2,3".

## Resumen de Búsqueda y Modificación

Método	¿Modifica el Original?	¿Qué devuelve?
splice()	Sí	Array de elementos eliminados.
indexOf()	No	Un número (índice) o -1.
isArray()	No	Booleano (true/false).
toString()	No	Una cadena de texto.

**El método filter():** en JavaScript crea un nuevo array con todos los elementos que cumplen una condición implementada por una función proporcionada. Recorre el array original, aplica la lógica a cada elemento y solo incluye aquellos que devuelven true en el resultado, sin modificar el array original. **Ejemplo**

```
const nums = [1, 2, 3, 4];  
const pares = nums.filter(n => n % 2 === 0);  
// [2, 4]
```

## Aquí tienes el desglose detallado:

### 1. ¿Cómo funciona filter?

**Iteración:** filter() recorre el array elemento por elemento, desde el primero hasta el último.

**Callback:** Por cada elemento, ejecuta una función de "callback" (en este caso,  $n \Rightarrow n \% 2 === 0$ ).

**Condición:** Si la función callback devuelve true (o un valor equivalente a verdadero), el elemento se añade al nuevo array.

**Inmutabilidad:** El array original no se altera.

### 2. Explicación detallada: $n \Rightarrow n \% 2 === 0$

Esta es una **función flecha** (arrow function) que actúa como condición de filtrado:

- **n:** Es el argumento que representa el elemento actual del array que se está procesando en cada iteración.
- **$n \% 2$ :** Es el operador de residuo (módulo). Divide n entre 2 y obtiene el resto.
- **$== 0$ :** Verifica si el resto de la división es exactamente cero.

Si  $n \% 2 == 0$ , el número es **par** y la función devuelve true.

Si  $n \% 2 != 0$ , el número es **ímpar** y la función devuelve false.

### 3. Paso a paso del funcionamiento

```
const numeros = [1, 2, 3, 4, 5, 6];
```

```
const pares = numeros.filter(n => n % 2 === 0);
```

```
// Resultado: [2, 4, 6]
```

1. **Inicio:** El método filter toma el primer elemento, 1.
2. **Evaluación:** ¿ $1 \% 2 == 0$ ? Es  $1 == 0$ , lo cual es false. El 1 **no** se incluye.
3. **Siguiente:** Toma el segundo elemento, 2.
4. **Evaluación:** ¿ $2 \% 2 == 0$ ? Es  $0 == 0$ , lo cual es true. El 2 **se incluye**.
5. **Proceso:** Continúa así con todos los elementos (3 es false, 4 es true, 5 es false, 6 es true).
6. **Final:** Devuelve un nuevo array: [2, 4, 6].

En resumen, la expresión  $n \Rightarrow n \% 2 == 0$  filtra un array para quedarse únicamente con los números pares.

**El método find()** en JavaScript busca en un array y devuelve el **primer elemento** que cumple una condición dada, deteniéndose inmediatamente. La expresión  $u \Rightarrow u.id == 4$  es una función flecha que actúa como callback: evalúa cada objeto (u) del array, comprobando si su propiedad id es exactamente 4.

### Ejemplo

```
const usuarios = [{id:1}, {id:2}];  
const user = usuarios.find(u => u.id == 2);  
// {id: 2}
```

### Funcionamiento Paso a Paso de array.find( $u \Rightarrow u.id == 4$ )

**Iteración:** El método recorre el array elemento por elemento, empezando desde el índice 0.

**Ejecución del Callback:** Por cada elemento (al que llamaremos u), ejecuta la función  $u.id == 4$ .

**Evaluación:**

- Si  $u.id == 4$ , la función devuelve true.
- Si  $u.id \neq 4$ , devuelve false.

**Finalización:**

- **Si encuentra true:** find detiene la búsqueda y devuelve el objeto u completo.
- **Si termina el array sin encontrar true:** Devuelve undefined.

## Explicación detallada de `u => u.id === 4`

- `u` es un parámetro que representa el elemento actual del array que `find` está revisando en ese momento.
- `=>`: Es la flecha de la función (arrow function).
- `u.id === 4`: Es la condición lógica (booleana). Verifica si el valor de la propiedad `id` del elemento actual es estrictamente igual al número 4.

## Ejemplo Práctico

```
const usuarios = [
  { id: 1, nombre: 'Ana' },
  { id: 4, nombre: 'Juan' }, // <--- find() se detendrá aquí
  { id: 5, nombre: 'Pedro' }
];
const resultado = usuarios.find(u => u.id === 4);
console.log(resultado); // Salida: { id: 4, nombre: 'Juan' }
```

### Puntos clave del método `find`:

- No modifica el array original.
- Si no encuentra coincidencia, devuelve `undefined`.
- Solo devuelve el *primer* elemento encontrado, incluso si hay otros con `id: 4` más adelante.

**El método `reduce()`** en JavaScript transforma un array en un único valor acumulado ejecutando una función callback por cada elemento. La estructura `(acc, n) => acc + n` es una función flecha que suma el acumulador (`acc`) con el número actual (`n`), ideal para sumar elementos, devolviendo el total al finalizar.

## Ejemplo

```
const nums = [1, 2, 3];
const suma = nums.reduce((acc, n) => acc + n, 0);
// 6
```

## Funcionamiento Detallado paso a paso: `array.reduce((acc, n) => acc + n, 0)`

1. **Inicialización:** `reduce` comienza con un valor inicial opcional (aquí 0). El acumulador (`acc`) toma este valor inicial en la primera iteración.
2. **Iteración 1:**
  - `acc = 0` (valor inicial)
  - `n = Primer elemento del array`
  - Resultado: `0 + primerElemento`
3. **Iteraciones Siguientes:** La función toma el resultado anterior como `acc` y el siguiente elemento como `n`, sumándolos.
4. **Finalización:** Cuando no quedan elementos, `reduce` devuelve el valor final de `acc`.

## Explicación de `(acc, n) => acc + n`

- **acc (Acumulador):** Es el valor resultante de la operación anterior. Se "acumula" el resultado.
- **n (Valor Actual):** Es el elemento del array que se está procesando actualmente.
- **=> (Arrow Function):** Define una función que toma esos dos parámetros y realiza una operación.
- **acc + n:** Suma el acumulado con el número actual.

**Ejemplo Práctico:** [1, 2, 3].reduce((acc, n) => acc + n, 0)

- **Paso 1:** acc = 0, n = 1 → 0 + 1 = 1
- **Paso 2:** acc = 1, n = 2 → 1 + 2 = 3
- **Paso 3:** acc = 3, n = 3 → 3 + 3 = 6
- **Resultado Final:** 6.

## forEach()

### Ejemplo

```
nums.forEach(n => console.log(n));
```

## 1. La Función forEach

- **Propósito:** Recorrer un array y ejecutar una acción sobre cada elemento.
- **Funcionamiento:** array.forEach(callback(currentValue, index, array)).
  - currentValue: El elemento actual siendo procesado.
  - index (opcional): El índice del elemento actual.
  - array (opcional): El array sobre el que se aplica forEach.
- **Características clave:**
  - **No retorna nada:** Siempre devuelve undefined, por lo que no se puede encadenar con otros métodos (como map o filter).
  - **No es interrumpible:** No se puede usar break o continue para detener el bucle.
  - **Eficiencia:** Es más legible que un bucle for tradicional, pero suele ser similar en rendimiento.

### Ejemplo:

```
const frutas = ["manzana", "banana", "uva"];
```

```
frutas.forEach(fruta => console.log(fruta));
```

```
// Imprime: manzana, luego banana, luego uva
```

## 2. Explicación detallada de n => console.log(n)

Este código es una **Arrow Function (función flecha)** anónima que se pasa como argumento al **forEach**.

- **n:** Es el parámetro que representa el **elemento actual** del array en cada iteración. Se puede llamar de cualquier forma (por ejemplo, elemento, item, num).
- **=>:** Es la "flecha" que separa los parámetros de entrada del cuerpo de la función.
- **console.log(n):** Es la acción (cuerpo) que se ejecuta. Toma el valor de n (el elemento actual) y lo imprime en la consola del navegador o terminal.

### Equivalencia:

Es una forma corta de escribir:

```
function(n) {  
  console.log(n);  
}
```

### Funcionamiento conjunto:

Si tienes const nums = [1, 2, 3];, al ejecutar nums.forEach(n => console.log(n));, el motor de JS hace:

1. **Iteración 1:** n es 1 -> console.log(1)
2. **Iteración 2:** n es 2 -> console.log(2)
3. **Iteración 3:** n es 3 -> console.log(3)

**El método some()** en JavaScript determina si al menos un elemento de un array cumple con una condición dada por una función de prueba, devolviendo true si lo encuentra (y deteniendo la iteración) o false si ninguno cumple. La expresión `n => n > 3` es una **arrow function** (función flecha) que toma un elemento `n` y devuelve true si `n` es mayor que 3.

#### Ejemplo:

```
nums.some(n => n > 3); // true
```

#### Detalle de Array.prototype.some():

- **Propósito:** Comprueba si algún elemento del array pasa la prueba.
- **Funcionamiento:** Itera sobre cada elemento, ejecutando una función callback.
- **Valor de retorno:** Booleano (true o false).
- **Comportamiento de cortocircuito:** En cuanto la función callback devuelve true, `.some()` devuelve true inmediatamente y detiene la iteración sobre el resto de elementos.
- **No mutabilidad:** No modifica el array original.

#### Ejemplo de uso:

```
const numeros = [1, 2, 5, 4];
const hayMayorQueTres = numeros.some(n => n > 3);
console.log(hayMayorQueTres); // Resultado: true
```

#### Detalle de `n => n > 3`

Esta expresión es una sintaxis concisa de ES6 conocida como *arrow function* (función flecha).

- **`n`:** Representa el parámetro de entrada. En el contexto de `.some()`, esto equivale al elemento actual que se está procesando del array.
- **`=>`:** Es el operador de flecha que separa los parámetros de entrada del cuerpo de la función.
- **`n > 3`:** Es el cuerpo de la función y la condición de prueba.
  - Si el valor actual `n` es mayor que 3, la función devuelve true.
  - Si no, devuelve false.

#### Desglose en una función tradicional:

```
function(n) {
  return n > 3;
}
```

En conjunto, `array.some(n => n > 3)` se lee como: "¿Existen algunos elementos en el array que sean mayores que 3?".

**El método every()** en JavaScript comprueba si **todos** los elementos de un array cumplen una condición definida por una función callback, devolviendo true solo si la condición es verdadera para cada elemento, y false en caso contrario. Se detiene inmediatamente si encuentra un elemento que no cumple la condición.

#### Ejemplo:

```
nums.every(n => n > 0); // true
```

#### Detalle de Array.prototype.every():

- **Finalidad:** Verifica si todos los elementos superan un test.
- **Funcionamiento:** Itera sobre el array y ejecuta la función `callbackFn` en cada elemento.
- **Valor de Retorno:** true si `callbackFn` devuelve un valor *truthy* para todos los elementos. false si encuentra un valor *falsy*.
- **Comportamiento especial:** Si el array está vacío, devuelve true (vacíamente verdadero).
- **Ejemplo:** `[2, 4, 6].every(n => n > 0) -> true` (todos son > 0).

## Detalle de la función `n => n > 0`:

Este código es una **función flecha (arrow function)** en JavaScript:

- **n**: Representa el argumento, que en este contexto es el elemento actual del array que se está evaluando.
- **=>**: Sintaxis de la función flecha.
- **n > 0**: La expresión a evaluar. Es un booleano que comprueba si el número n es estrictamente mayor que cero.

**En resumen:** `n => n > 0` es una función compacta que toma un número (n) y devuelve true si es positivo, o false si es cero o negativo.

## Uso conjunto:

```
const numeros = [1, 5, 10, 15];
const todosPositivos = numeros.every(n => n > 0);
// Resultado: true
```

## MÉTODOS PARA AÑADIR O ELIMINAR ELEMENTOS EN UN ARRAYS

**concat()**: Une arrays y devuelve uno nuevo (no modifica el original).

```
[1,2].concat([3,4]);
// [1,2,3,4]
```

### ✓ Inmutable

**La función concat()**: en JavaScript es un método predeterminado utilizado para fusionar dos o más arrays, o valores individuales, en uno nuevo, sin modificar los arrays originales. Devuelve una "copia superficial" (*shallow copy*) que contiene los elementos del array base seguidos de los elementos pasados como argumentos.

#### Detalles de `[1,2].concat([3,4])`:

- **Funcionamiento:** Toma el array [1, 2] y le añade los elementos del array [3, 4] al final.
- **Resultado:** Produce un nuevo array: [1, 2, 3, 4].
- **Inmutabilidad:** Los arrays originales [1, 2] y [3, 4] permanecen inalterados.
- **Aplanamiento:** concat() no aplana recursivamente arrays anidados, solo el primer nivel.

#### Características clave de concat():

- **Sintaxis:** array1.concat(valor1, valor2, ...).
- **Parámetros:** Se pueden concatenar arrays múltiples o valores sueltos (números, strings, objetos).
- **Uso:** Ideal para crear combinaciones de datos sin causar efectos secundarios en las estructuras de datos originales.

**join()**: Convierte el array en texto.

```
[1,2,3].join("-");
// "1-2-3"
```

**La función join()** en JavaScript convierte todos los elementos de un array en una sola cadena de texto (string), concatenándolos y separándolos por un carácter específico (o una coma por defecto). El código `[1,2,3].join("-")` toma el array [1, 2, 3], une sus elementos con un guion y devuelve la cadena "1-2-3".

#### Detalles de la función join()

- **Propósito:** Unir elementos de un array en un string.
- **Sintaxis:** array.join(separador)
- **Parámetros:** El separador es opcional. Es una cadena que se inserta entre cada elemento.
  - Si se omite, los elementos se separan por una coma (,).
  - Si es una cadena vacía (""), los elementos se unen sin nada entre ellos.

- **Comportamiento:**

- No modifica el array original; devuelve una nueva cadena.
- Si un elemento es null o undefined, se convierte en una cadena vacía.
- Si el array tiene un solo elemento, ese elemento se devuelve sin el separador.

- **Ejemplos:**

- ['a', 'b'].join() // "a,b"
- ['a', 'b'].join("") // "ab"
- ['a', 'b'].join(' ') // "a b"

### Explicación del código: [1,2,3].join("-")

1. **[1,2,3]:** Se define un array literal que contiene tres elementos numéricos: 1, 2 y 3
2. **.join("-"):** Se invoca el método join sobre este array, pasando el guion ("") como separador.
3. **Proceso:** El método toma el primer elemento (1), añade el separador (-), toma el segundo (2), añade el separador (-) y toma el último (3).
4. **Resultado:** La expresión devuelve la cadena de texto: "**1-2-3**"

**copyWithin():** Copia parte del array dentro del mismo array.

[1,2,3,4].copyWithin(0,2);

// [3,4,3,4]

✓ Modifica el array original

La función **copyWithin()** en JavaScript es un método mutador de arrays de alto rendimiento que copia una secuencia de elementos dentro del mismo array, sobrescribiendo los valores existentes sin cambiar la longitud total del array. Es similar a memmove en C/C++ y gestiona correctamente las solapamientos de memoria.

### Análisis de [1,2,3,4].copyWithin(0,2)

[1, 2, 3, 4].copyWithin(0, 2);

**Resultado:** [3, 4, 3, 4]

#### Desglose paso a paso:

1. **Array Original:** [1, 2, 3, 4]
2. **copyWithin(target, start):**
  - **target (0):** El índice donde se empezará a *escribir* (sobrescribir) los datos.
  - **start (2):** El índice donde se empezará a *leer* (copiar) los datos.
3. **Operación:**
  - Toma la secuencia desde el índice 2 hasta el final del array: [3, 4].
  - Pega esta secuencia empezando en el índice 0.
  - El 1 (índice 0) es reemplazado por 3.
  - El 2 (índice 1) es reemplazado por 4.
4. **Resultado final:** Los elementos copiados 3, 4 sobrescriben los primeros dos elementos, resultando en [3, 4, 3, 4].

#### Detalles Técnicos de copyWithin()

- **Sintaxis:** array.copyWithin(target, start, end)
- **Mutador:** Modifica el array original.
- **No cambia la longitud:** El array resultante tiene el mismo length que el original.

- **Parámetros:**

- target: Índice destino. Si es negativo, cuenta desde el final.
- start (opcional): Índice de inicio de copia (default: 0).
- end (opcional): Índice final de copia (no incluido). Default: array.length.

**pop():** Elimina el último elemento.

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

```
a.pop();
```

```
// [1,2]
```

**La función pop()** en JavaScript es un método nativo de los arreglos (Array.prototype.pop()) que elimina el **último** elemento de un array, modifica la longitud original del arreglo y devuelve el elemento eliminado. Si el array está vacío, devuelve undefined. Es fundamental para gestionar estructuras de datos tipo pila (LIFO - Last In, First Out).

### Explicación detallada de pop()

1. **Eliminación:** Remueve el elemento en la última posición del array (índice length - 1).
2. **Mutación:** Cambia el arreglo original reduciendo su longitud en una unidad.
3. **Retorno:** Devuelve el elemento que fue eliminado, permitiendo guardarlo en una variable.
4. **Uso:** arreglo.pop() no requiere argumentos.
5. **Array Vacío:** Si el array está vacío, devuelve undefined y la longitud permanece en 0.

### Análisis de a.pop():

- **a:** Es una variable que contiene un arreglo (array) de JavaScript (ej. let a = [1, 2, 3];).
- **.**: Operador de acceso a miembros, utilizado para llamar a un método del objeto a.
- **pop()**: El método invocado sobre el objeto a.
- **;**: Punto y coma que finaliza la sentencia.

### Ejemplo de ejecución:

```
let a = ['manzana', 'banana', 'naranja'];
let eliminado = a.pop();
```

```
console.log(a);      // Resultado: ["manzana", "banana"] (Se eliminó "naranja")
console.log(eliminado); // Resultado: "naranja" (Elemento eliminado devuelto)
```

### Puntos clave:

- La longitud de a pasó de 3 a 2.
- El método pop() modifica el arreglo a de manera permanente.
- El elemento 'naranja' ya no forma parte del arreglo a

**El método fill():** Rellena todos los elementos de un array con un valor estático, desde un índice de inicio hasta un índice final.

### Ejemplo

JavaScript

```
const nums = [1, 2, 3];
nums.fill(0);
// [0, 0, 0]
```

**Uso común:** Útil para inicializar arrays vacíos con un valor por defecto o "limpiar" datos de un vector rápidamente.

**El método fill() modifica el array original** (es mutable). Transforma el contenido existente reemplazándolo por el valor que tú decidas.

## Desglose paso a paso:

### 1. ¿Cómo funciona fill()?

- **Sustitución Directa:** Cambia los valores actuales por el nuevo valor proporcionado.
- **Mutabilidad:** A diferencia de map(), este sí altera el array inicial.
- **Rango Personalizable:** Puede recibir tres parámetros: valor, inicio y fin.

### 2. Funcionamiento paso a paso ([1, 2, 3].fill(0)):

- **Inicio:** El método identifica el valor a insertar (0).
- **Recorrido:** Pasa por el índice 0, 1 y 2.
- **Reemplazo:** Cambia 1 por 0, 2 por 0, y 3 por 0.
- **Resultado:** El array original ahora es [0, 0, 0].

**El método push():** Añade uno o más elementos al **final** de un array y devuelve la nueva longitud del mismo.

## Ejemplo

JavaScript

```
let a = [1, 2];
a.push(3);
// [1, 2, 3]
```

**Uso común:** Es la forma más básica de ir acumulando datos en una lista, como agregar productos a un carrito de compras.

## Desglose paso a paso:

### 1. ¿Cómo funciona push()?

- **Extensión:** Agranda el tamaño del array original.
- **Posición:** El nuevo elemento siempre ocupa el último índice (array.length).
- **Retorno:** Aunque modifica el array, si guardas el resultado en una variable, obtendrás el nuevo tamaño (número).

### 2. Funcionamiento paso a paso ([1, 2].push(3)):

- **Identificación:** Se toma el valor 3.
- **Posicionamiento:** Se busca el final de la lista (después del 2).
- **Inserción:** Se añade el 3 al final.
- **Resultado:** a pasa a ser [1, 2, 3].

**3. El método shift():** Elimina el **primer** elemento de un array y devuelve ese elemento eliminado.

## Ejemplo

```
let a = [1, 2, 3];
a.shift();
// [2, 3]
```

**Uso común:** Ideal para gestionar colas (First-In, First-Out), donde el primero en llegar es el primero en ser atendido.

## Desglose paso a paso:

### 1. ¿Cómo funciona shift()?

- **Extracción:** Saca el elemento que está en el índice 0.
- **Reindexación:** Al quitar el primero, todos los demás elementos "se mueven" un lugar hacia la izquierda.
- **Mutabilidad:** Modifica el array original reduciendo su longitud en 1.

### 2. Funcionamiento paso a paso ([1, 2, 3].shift()):

- **Extracción:** Se separa el 1 del resto del grupo.
- **Desplazamiento:** El 2 (que era índice 1) pasa a ser índice 0. El 3 (que era índice 2) pasa a ser índice 1.
- **Resultado:** El array queda como [2, 3].

**El método unshift():** Agrega uno o más elementos al **inicio** de un array y devuelve la nueva longitud.

### Ejemplo

```
let a = [2, 3];
a.unshift(1);
// [1, 2, 3]
```

**Uso común:** Cuando necesitas que un elemento nuevo tenga prioridad máxima o sea el primero en visualizarse (como una notificación nueva en una lista).

### Desglose paso a paso:

#### 1. ¿Cómo funciona unshift()?

- **Inserción Frontal:** Coloca el nuevo valor en la posición 0.
- **Desplazamiento a la derecha:** Mueve todos los elementos existentes una posición hacia adelante para hacer espacio.
- **Mutabilidad:** Altera el array original.

#### 2. Funcionamiento paso a paso ([2, 3].unshift(1)):

- **Espacio:** El método "empuja" al 2 y al 3 hacia la derecha.
- **Inserción:** El 1 se coloca en el espacio vacío del índice 0.
- **Resultado:** El array ahora es [1, 2, 3].

### Resumen de Mutabilidad

Método	¿Modifica el original?	Ubicación del cambio
map()	No (Crea uno nuevo)	Todo el array
fill()	Sí	Todo o parte del array
push()	Sí	Al final
shift()	Sí	Al inicio (Elimina)
unshift()	Sí	Al inicio (Agrega)

## MÉTODOS PARA ORDENAR O EXTRAER EN UN ARRAYS

**El método reverse():** Invierte el orden de los elementos de un array in situ (en el lugar). El primer elemento pasa a ser el último y el último pasa a ser el primero.

### Ejemplo

```
const nums = [1, 2, 3];
nums.reverse();
// [3, 2, 1]
```

**Nota Crítica:** Este método modifica el array original. No crea una copia, sino que altera la estructura del objeto existente.

### Desglose paso a paso:

#### 1. ¿Cómo funciona reverse()?

- **Inversión de Índices:** Intercambia las posiciones de los elementos basándose en su distancia desde los extremos.
- **Mutabilidad:** Altera directamente la variable original.
- **Simetría:** Si lo aplicas dos veces, el array vuelve a su estado original.

#### 2. Funcionamiento paso a paso ([1, 2, 3].reverse()):

- **Inicio:** Identifica el primer elemento (1) y el último (3).
- **Intercambio 1:** El 3 se mueve al índice 0 y el 1 al índice 2.
- **Eje central:** El 2 se queda en su lugar porque es el centro.
- **Finalización:** Se devuelve el array modificado.
- **Resultado:** El array ahora es [3, 2, 1].

**El método sort():** Ordena los elementos de un array y devuelve el array ordenado. Por defecto, el ordenamiento responde a posiciones de valor Unicode (como texto).

### Ejemplo (Comportamiento por defecto)

```
const nums = [10, 2, 5];
nums.sort();
// [10, 2, 5] -> ¡Incorrecto para números!
// "1" de 10 viene antes que "2".
```

### Ejemplo (Orden numérico correcto)

```
const nums = [10, 2, 5];
nums.sort((a, b) => a - b);
// [2, 5, 10]
```

**Desglose paso a paso:**

#### 1. ¿Cómo funciona sort()?

- **Conversión a String:** Si no se provee una función, JavaScript convierte los elementos en "palabras" para compararlos.
- **Función de Comparación:** Recibe una función (a, b) que decide quién va primero.
- **Mutabilidad:** Al igual que reverse(), modifica el array original.

#### 2. Explicación detallada de la función: $(a, b) \Rightarrow a - b$

- a: Representa el "segundo" elemento a comparar.
- b: Representa el "primer" elemento a comparar.
- Lógica de resta:
  - Si  $a - b$  es negativo, a se ordena antes que b.
  - Si  $a - b$  es positivo, b se ordena antes que a.
  - Si es 0, se mantienen iguales.

#### 3. Funcionamiento paso a paso ( $[10, 2, 5].sort((a, b) \Rightarrow a - b)$ ):

**Comparación 1:** Compara 10 y 2. Ejecuta  $10 - 2 = 8$  (Positivo). Intercambia: [2, 10, 5].

**Comparación 2:** Compara 10 y 5. Ejecuta  $10 - 5 = 5$  (Positivo). Intercambia: [2, 5, 10].

**Comparación 3:** Compara 2 y 5. Ejecuta  $2 - 5 = -3$  (Negativo). Se quedan como están.

**Resultado:** El array final ordenado es [2, 5, 10].

### Resumen de Diferencias

Método	Propósito	¿Es destructivo?	Notas
reverse()	Invertir el orden.	Sí	Cambia el sentido de los datos.
sort()	Organizar datos.	Sí	¡Cuidado! Siempre usa función de comparación para números.

## MATRICES O ARRAYS MULTIDIMENSIONALES

Un array puede contener otros arrays.

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

### Acceso:

```
matriz[1][2];
// 6
```

## FUNCIONAMIENTO PASO A PASO (Ejemplo con push)

```
let nums = [1,2];
nums.push(3);
1. nums = [1,2]
2. push(3)
3. Se agrega al final
4. Resultado → [1,2,3]
5. El array original cambia
```

## Funciones predeterminadas de OBJETOS

### Object.keys()

#### Ejemplo:

```
Object.keys(obj)
```

#### Devuelve

- Array de strings

```
Object.keys({a:1, b:2}); // ["a", "b"]
```

La función **Object.keys()** en JavaScript es un método estático que devuelve un array con los nombres de las propiedades enumerables propias de un objeto, en el mismo orden que un bucle `for...in`. No incluye propiedades heredadas del prototipo. La expresión `Object.keys({a:1, b:2})` devuelve `["a", "b"]`.

#### Detalle de Object.keys():

- **Propósito:** Extraer los nombres de las claves (keys) de un objeto para poder iterar sobre ellos.
- **Valor de retorno:** Un array (`[]`) que contiene cadenas de texto (strings) correspondientes a las llaves.
- **Características clave:**
  - **Enumerables:** Solo devuelve propiedades que tienen su flag enumerable en true.
  - **Propias:** Solo devuelve propiedades definidas directamente en el objeto, no en su cadena de prototipos.
  - **Orden:** Sigue el mismo orden de un bucle `for...in`.
- **Sintaxis:** `Object.keys(obj)` donde `obj` es el objeto cuyas propiedades se enumeran.

#### Análisis detallado del código `Object.keys({a:1, b:2})`:

1. **{a:1, b:2}:** Se define un objeto literal nuevo.
  - a y b son las claves (keys).
  - 1 y 2 son los valores asociados (values).
2. **Object.keys(...):** Se invoca el método estático sobre este objeto.
  - El método escanea las propiedades propias y enumerables: a y b.
3. **Resultado:** La función recopila estas llaves y crea un nuevo array de strings.
  - **Salida:** `["a", "b"]`.

#### Ejemplo de uso:

```
const objeto = {a: 1, b: 2};
const llaves = Object.keys(objeto);
console.log(llaves); // Resultado: ["a", "b"]
```

**Object.keys** es fundamental para transformar objetos en arreglos, permitiendo usar métodos de arreglos modernos como `.map()`, `.filter()` o `.reduce()` sobre las propiedades de un objeto.

## **Object.values()**

### **Ejemplo:**

```
Object.values({a:1, b:2}); // [1,2]
```

**Object.values()** en JavaScript es una función estática que toma un objeto como argumento y devuelve un *array* (arreglo) con los valores de sus propiedades enumerables, en el mismo orden que lo haría un bucle `for...in`. Es útil para iterar sobre los valores de un objeto directamente sin usar sus claves.

### **Explicación detallada de Object.values({a:1, b:2});**

1. **Entrada ({a:1, b:2})**: Se pasa un objeto literal como argumento con dos pares clave-valor:
  - Clave 'a' -> Valor 1
  - Clave 'b' -> Valor 2
2. **Proceso**: El método `Object.values()` inspecciona el objeto, ignora las claves ('a' y 'b') y extrae solo los valores.
3. **Salida ([1, 2])**: Se crea y devuelve un nuevo arreglo que contiene los valores encontrados en el orden de aparición.

### **Ejemplo paso a paso:**

```
const objeto = {a: 1, b: 2};  
const resultado = Object.values(objeto);  
console.log(resultado); // Muestra: [1, 2]
```

### **Notas Clave:**

- Si el objeto no tiene propiedades enumerables, devuelve un arreglo vacío [].
- Si se le pasa un valor primitivo (como string o number), intenta convertirlo en objeto.
- Si recibe null o undefined, lanzará un error `TypeError`

## **Object.entries()**

### **Ejemplo**

```
Object.entries({a:1});  
// [["a",1]]  
Ideal para usar con map().
```

**Object.entries()** es un método de JavaScript que convierte un objeto enumerable en una matriz (*array*) de pares [clave, valor]. Retorna un array de arrays, permitiendo iterar fácilmente sobre las propiedades propias. El código `Object.entries({a:1})` toma el objeto `{a:1}` y devuelve `[['a', 1]]`.

### **Detalles de Object.entries()**

- **Propósito**: Transforma objetos en estructuras tipo lista, facilitando su recorrido con métodos como `forEach`, `map` o `for...of`.
- **Funcionamiento**: Devuelve un *array* cuyos elementos son pares [clave, valor] correspondientes a las propiedades enumerables directas del objeto.
- **Orden**: Sigue el mismo orden que un bucle `for...in` manual.
- **Comportamiento con tipos**: Si se pasa un objeto que no sea primitivo, se convierte a objeto. Si se pasa null o undefined, lanza un `TypeError`.
- 

### **Análisis de Object.entries({a:1})**

1. **Entrada**: `{a:1}` es un objeto literal con una propiedad enumerable: la clave es 'a' y su valor es 1.
2. **Procesamiento**: `Object.entries` analiza el objeto y extrae cada par [clave, valor].

3. **Salida:** Produce un array de arrays: `[['a', 1]]`.

- El primer elemento `[['a', 1]][0]` es un array que contiene la clave y el valor: `['a', 1]`.
- 'a' es el índice 0 del par (la clave).
- 1 es el índice 1 del par (el valor).

#### Ejemplo de uso:

```
const obj = { a: 1 };
console.log(Object.entries(obj)); // Resultado: [['a', 1]]
```

Este método es considerado más conciso que los bucles tradicionales para acceder a las propiedades de un objeto.

## Funciones predeterminadas del DOM

La función `document.querySelector()` en JavaScript es un método esencial del DOM que devuelve el **primer** elemento del documento que coincide con un selector CSS específico. Si no encuentra coincidencias, devuelve null. Se utiliza para seleccionar elementos HTML de manera precisa y realizar manipulaciones en ellos, como cambiar estilos o añadir eventos.

#### Ejemplo:

```
document.querySelector("button");
```

Devuelve

- Elemento DOM o null

Aquí te explico ambos puntos detalladamente:

### 1. La función predeterminada `querySelector`

- **¿Qué hace?** Busca en el árbol DOM (el documento HTML) y devuelve el primer elemento (tipo nodo Element) que cumple con el selector CSS proporcionado.
- **Sintaxis:** `let elemento = document.querySelector(selector);`
  - **Selector:** Es una cadena de texto (string) que sigue las reglas de los selectores CSS (ej. ".clase", "#id", "tag").
  -
- **Comportamiento clave:**
  - **Primer Elemento:** Si múltiples elementos coinciden con el selector, solo se devuelve el primero.
  - **Búsqueda Rápida:** Utiliza un recorrido profundo (pre-order) del DOM.
  - **Alternativa:** Si se necesita obtener *todos* los elementos coincidentes, se usa `querySelectorAll()`.
  - **Retorno:** Devuelve el elemento encontrado o null si no hay coincidencias.

### 2. Explicación detallada de `document.querySelector("button")`:

```
const boton = document.querySelector("button");
```

- **document:** Hace referencia a la interfaz que representa toda la página web cargada en el navegador (el árbol DOM). Es el punto de entrada para manipular el contenido.
- **.querySelector(...):** Es el método llamado sobre document para buscar un elemento.
- **"button":** Es el selector CSS pasado como argumento. En este caso, busca una etiqueta HTML de tipo botón (`<button>`).
- **Resultado:** Esta línea busca la primera etiqueta `<button>` que encuentre en la página.
  - Si el HTML es: `<button class="btn1">Enviar</button><button class="btn2">Cancelar</button>`.
  - `document.querySelector("button")` seleccionará el **primer** botón ("Enviar").

- **Uso:** Generalmente se guarda en una variable (const botón) para luego usarla, por ejemplo, para agregar un evento: botón.addEventListener("click", ...).

En resumen, **document.querySelector("button")** es la forma estándar y moderna de seleccionar el primer botón del DOM para interactuar con él.

**La función querySelectorAll():** es un método esencial de JavaScript utilizado para seleccionar múltiples elementos del DOM que coinciden con un selector CSS específico, devolviendo una {Link: NodeList https://developer.mozilla.org/es/docs/Web/API/NodeList} estática.

**Ejemplo:**

```
document.querySelectorAll("li");
```

**Devuelve:**

- **NodeList (similar a array)**

```
document.querySelectorAll("li")
.forEach(li => console.log(li.textContent));
```

**El código .forEach(li => console.log(li.textContent)):** itera sobre esa lista y muestra el texto de cada elemento en la consola.

**Detalles de querySelectorAll()**

- **Función:** Busca en el documento (o un elemento padre) todos los nodos que cumplen con los selectores CSS proporcionados (clases, IDs, etiquetas, etc.).
- **Valor de Retorno:** Devuelve una **NodeList estática**. Esto significa que si el DOM cambia después de la selección, la lista no se actualiza automáticamente.
- **Sintaxis:** elementoPadre.querySelectorAll('selectorCSS'). Ejemplo: document.querySelectorAll('.lista-item').
- **Diferencia:** A diferencia de querySelector(), que devuelve solo el primer elemento, querySelectorAll() devuelve **todos** los elementos.

### Explicación detallada de .forEach(li => console.log(li.textContent))

Este fragmento de código toma la NodeList generada por querySelectorAll() y aplica una función a cada elemento contenido en ella.

1. **.forEach(...):** Es un método disponible en las NodeList que permite ejecutar una función callback una vez por cada elemento de la lista.
2. **li => ...:** Es una función flecha (arrow function) que actúa como callback. li es un parámetro que representa el elemento actual de la iteración (se llama 'li' por convención si son elementos de lista, pero puede ser cualquier nombre).
3. **console.log(...):** Imprime en la consola del navegador.
4. **li.textContent:** Es una propiedad del nodo DOM (li) que obtiene el texto contenido dentro de ese elemento, excluyendo las etiquetas HTML.

**Ejemplo completo:**

```
// 1. Selecciona todos los elementos <li> dentro de un elemento con id "menu"
const items = document.querySelectorAll('#menu li');
```

```
// 2. Itera y muestra el texto de cada uno
items.forEach(li => console.log(li.textContent));
```

En resumen, la línea items.forEach(li => console.log(li.textContent)) toma cada elemento seleccionado, accede a su texto (textContent) y lo imprime en la consola, siendo útil para depuración o procesamiento de datos de una lista.

## **addEventListener()**

### **¿Qué hace?**

Escucha eventos.

```
element.addEventListener("click", e => {console.log(e.target)});
```

### **1. ¿Qué es addEventListener()?**

Es un método de JavaScript que se aplica a un elemento del DOM (como un botón, un enlace o un div). Su función es "quedarse escuchando" a que ocurra un evento específico para luego ejecutar una función.

Piénsalo como configurar una alarma: tú le dices a JavaScript **qué sonido esperar y qué hacer** cuando suene.

### **La sintaxis básica**

```
elemento.addEventListener(evento, función, opciones);
```

- **Evento:** El tipo de acción (clic, movimiento del ratón, presionar una tecla). Se escribe como un *string*.
- **Función:** El bloque de código que se ejecutará cuando ocurra el evento.
- **Opciones (opcional):** Ajustes avanzados como si el evento debe ocurrir solo una vez.

### **2. Desglose del código: element.addEventListener("click", e => {console.log(e.target)})**

Este código es una forma moderna y concisa de manejar un clic. Vamos a desmembrarlo pieza por pieza:

#### **A. element**

Es la referencia al objeto del HTML sobre el que queremos actuar. Normalmente lo habrás obtenido antes con algo como `const element = document.querySelector('#miBoton');`

#### **B. "click"**

Es el **primer argumento**. Indica que la función solo se activará cuando el usuario haga clic izquierdo (o toque en una pantalla táctil) sobre ese elemento.

#### **C. e => { ... } (La función flecha)**

Este es el **segundo argumento**. Es una función anónima (no tiene nombre) que se ejecuta "en respuesta" al clic.

- **e (o event):** Es el objeto del evento. Cuando haces clic, el navegador crea automáticamente un objeto lleno de información sobre lo que acaba de pasar (coordenadas del ratón, qué teclas se presionaron, etc.) y lo pasa como argumento a tu función.

#### **D. e.target**

Aquí es donde ocurre la magia.

- **target** es una propiedad del objeto de evento e.
- Representa al **elemento exacto donde se originó el evento**.
- **console.log(e.target)**: Imprime en la consola del navegador la etiqueta HTML completa del elemento que recibió el clic.

### **¿Por qué usar e.target en lugar de solo element?**

Aunque en este código simple podrían parecer lo mismo, hay una diferencia clave:

1. **element** es el nodo al que le pusiste el "oído" (el receptor).
2. **e.target** es el elemento que realmente recibió el clic.

Si tu element es un `<div>` que contiene un `<span>` y un `<img>`, y haces clic en la imagen, `e.target` te devolverá la imagen (`<img>`), mientras que el receptor sigue siendo el `<div>`. Esto es fundamental para una técnica llamada **Delegación de Eventos**.

### **Ejemplo de lo que verías en consola:**

Si el elemento es un botón: `<button id="btn-enviar">Enviar</button>` Al hacer clic, la consola mostrará: `<button id="btn-enviar">Enviar</button>`

## Funciones de TIEMPO

La función **setTimeout()** en JavaScript se utiliza para programar la **ejecución de una función específica una única vez, después de un retraso de tiempo determinado**. Es una función asíncrona, lo que significa que el resto del código JavaScript sigue ejecutándose sin ser bloqueado mientras espera el tiempo especificado.

**Ejemplo:**

`setTimeout(() => {console.log("Hola");}, 1000);` Ejecuta una vez.

### Funcionamiento detallado de **setTimeout()**

- **Propósito:** Retrasar la ejecución de una tarea.
- **Asincronía:** setTimeout no pausa la ejecución del hilo principal de JavaScript. En su lugar, delega la tarea de esperar al entorno del navegador (Web API) o al entorno de Node.js.
- **Bucle de Eventos (Event Loop):** Una vez que el temporizador finaliza, la función de callback no se ejecuta inmediatamente, sino que se añade a una cola de tareas (cola de callbacks o macrotareas). El "bucle de eventos" se encarga de mover la función de la cola a la pila de ejecución (call stack) solo cuando la pila principal está vacía, asegurando un flujo de ejecución sin bloqueos.
- **Cancelación:** La llamada a setTimeout() devuelve un identificador (ID numérico) que se puede usar posteriormente para cancelar la ejecución programada mediante la función clearTimeout(id).
- 

**La sintaxis básica es:**

`setTimeout(funcion, retraso, [param1, param2, ...]);`

**funcion:** La función de callback (o una cadena de código, aunque no se recomienda) que se ejecutará tras el retraso.

**retraso:** El tiempo en **milisegundos** (1000 ms = 1 segundo) que debe transcurrir antes de que se ejecute la función. Si se omite, se usa un valor predeterminado de 0.

**param1, param2, ... (opcionales):** Argumentos adicionales que se pasan a la función cuando se ejecuta.

**Explicación detallada del código:**

`setTimeout(() => { console.log("Hola"); }, 1000);`

Este fragmento de código programa la impresión de la palabra "Hola" en la consola después de que haya transcurrido exactamente 1 segundo.

- **setTimeout(...):** Llama a la función global de JavaScript para programar una ejecución retrasada.
- **() => { console.log("Hola"); }:** Este es el primer argumento, una función de flecha (arrow function) anónima, que sirve como la función de *callback*. Contiene el código que queremos ejecutar más tarde. En este caso, la acción es imprimir "Hola" en la consola.
- **1000:** Este es el segundo argumento, el retraso especificado en milisegundos. Indica que la función de callback debe ejecutarse después de 1000 ms (1 segundo).

**En resumen:**

Cuando se ejecuta esta línea de código, JavaScript le dice al navegador/entorno de ejecución: "Espera 1000 milisegundos. Después de ese tiempo, por favor, coloca la función () => { console.log("Hola"); } en la cola de tareas para que sea ejecutada tan pronto como el hilo principal esté libre".

## **setInterval()**

setInterval(() => {console.log("Tick"); }, 1000);      Ejecuta **repetidamente**.

### **1. ¿Qué es setInterval()?**

Es una función global que le indica al navegador que debe ejecutar un bloque de código **repetidamente**, esperando un intervalo de tiempo fijo entre cada ejecución.

A diferencia de setTimeout() (que solo se ejecuta una vez tras una espera), setInterval() crea un bucle infinito que solo se detendrá si tú se lo ordenas explícitamente o si cierras la pestaña.

### **La sintaxis**

```
let idIntervalo = setInterval(funcion, milisegundos, parametro1, parametro2...);
```

- **Función:** El código que quieras que se repita.
- **Milisegundos:** El tiempo de espera entre repeticiones ( ms = segundo).
- **Parámetros:** (Opcional) Valores que puedes pasarle a la función.
- **Retorno:** Devuelve un **ID único** (un número). Este ID es fundamental para poder detener el intervalo después usando clearInterval(idIntervalo).

### **2. Desglose del código: setInterval(() => { console.log("Tick"); }, 1000);**

Este código configura un "reloj" básico que escribe en la consola cada segundo.

#### **A. La estructura () => { ... }**

Es una **función flecha** (arrow function) anónima. Se usa aquí porque es una forma limpia de definir la tarea que queremos repetir sin necesidad de declarar una función por separado en otro lugar. En este caso, la tarea es simplemente ejecutar el console.log.

#### **B. "Tick"**

Es el mensaje de texto que verás aparecer en la consola cada vez que el cronómetro llegue a cero.

#### **C. El valor 1000**

Es el segundo argumento de la función. Representa el tiempo en **milisegundos**.

- Si pones 1000, se ejecuta cada segundo.
- Si pones 500, se ejecuta dos veces por segundo.
- Si pones 60000, se ejecuta una vez por minuto.

### **3. ¿Cómo funciona por dentro? (El Event Loop)**

Es importante entender que setInterval() no garantiza que el código se ejecute **exactamente** cada 1000ms.

1. JavaScript coloca la tarea en una cola de espera.
2. Si el procesador está muy ocupado con un cálculo pesado, el "Tick" podría retrasarse unos milisegundos.
3. Sin embargo, el intervalo intentará recuperar el ritmo en la siguiente vuelta.

### **4. Ejemplo avanzado: Cómo detenerlo**

Si dejas el código tal como lo pusiste, el "Tick" seguirá sonando para siempre. Para controlarlo, normalmente hacemos esto:

```
let contador = 0;
const miReloj = setInterval(() => {
  contador++;
  console.log("Tick número: " + contador);
  if (contador === 5) {
    clearInterval(miReloj); // Esto detiene el intervalo
    console.log("¡Reloj detenido!"); }
}, 1000);
```

## Diferencias clave:

Característica	<b>setTimeout()</b>	<b>setInterval()</b>
<b>Frecuencia</b>	Una sola vez.	Infinitas veces (hasta detenerlo).
<b>Uso común</b>	Retrasos, alertas, esperas.	Reloj, animaciones, monitoreo de datos.

## BOM (BROWSER OBJECT MODEL)

**INTRODUCCIÓN:** El BOM (Browser Object Model) permite que JavaScript interactúe con el navegador. A diferencia del DOM (que trabaja con el documento HTML), el BOM trabaja con: Ventanas, Pestañas, URL, Historial, Pantalla, Información del navegador .

**El objeto principal del BOM es: window**

### EL OBJETO window()

window es el objeto global del navegador. Todo en el navegador depende de window.

**Ejemplo:**

```
window.alert("Hola");  
También puede escribirse:  
alert("Hola"); // Porque window es el objeto global.
```

## MÉTODOS DEL OBJETO window

**1. El método setTimeout():** Ejecuta una función después de que transcurra un tiempo específico.

**Ejemplo**

```
const id = setTimeout(() => {  
  console.log("¡Sorpresa!");  
}, 2000);  
// Espera 2 segundos y muestra el mensaje.
```

**Desglose paso a paso:**

### 1. ¿Cómo funciona?

- **Temporizador:** Inicia una cuenta regresiva en milisegundos.
- **No bloqueante:** El resto del código sigue ejecutándose mientras el reloj corre.
- **ID de retorno:** Devuelve un número identificador único.

### 2. Funcionamiento paso a paso:

- **Inicio:** El navegador recibe la tarea y el tiempo (2000ms).
- **Espera:** El motor de JS sigue con otras líneas de código.
- **Ejecución:** Al llegar a cero, la función se envía a la cola de ejecución.
- **Resultado:** Se ejecuta la lógica interna una sola vez.

**2. El método setInterval():** Ejecuta una función de forma repetitiva cada vez que se cumple un intervalo de tiempo.

**Ejemplo**

```
let contador = 0;  
const id = setInterval(() => {  
  contador++;  
  console.log(contador);  
}, 1000);  
// Imprime 1, 2, 3... cada segundo.
```

**Desglose paso a paso:**

### 1. ¿Cómo funciona?

- **Bucle Temporal:** Crea un ciclo infinito basado en tiempo.
- **Persistencia:** Solo se detiene si se cierra la pestaña o se usa clearInterval().

## 2. Funcionamiento paso a paso:

- **Inicio:** Se establece el ciclo (ej. 1000ms).
- **Ciclo 1:** Pasa 1 segundo -> Ejecuta código.
- **Ciclo 2:** Pasa otro segundo -> Re-ejecuta código.
- **Resultado:** La función se repite indefinidamente hasta ser cancelada.

3. El método **confirm()**: Muestra un cuadro de diálogo con un mensaje y dos botones: "Aceptar" y "Cancelar".

### Ejemplo

```
const respuesta = confirm("¿Borrar archivo?");
```

```
// Si pulsa Aceptar -> true
```

```
// Si pulsa Cancelar -> false
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona?

- **Interrupción:** Detiene la ejecución del script hasta que el usuario responda.
- **Booleano:** Devuelve un valor lógico basado en la elección.

#### 2. Funcionamiento paso a paso:

- **Llamada:** Se lanza el cuadro emergente en el navegador.
- **Espera:** El navegador queda "congelado" para el usuario en esa pestaña.
- **Respuesta:** El usuario elige.
- **Resultado:** Si eligió OK, respuesta es true. Si no, es false.

## EL OBJETO location

1. Propiedad **location.href**: Es la propiedad que contiene la dirección URL completa de la página actual.

### Ejemplo

```
console.log(location.href);
```

```
// "https://www.tuweb.com/inicio"
```

```
location.href = "https://google.com";
```

```
// Redirige inmediatamente.
```

### Desglose paso a paso:

#### 1. ¿Cómo funciona?

- **Lectura/Escritura:** Funciona como un "getter" (obtener) y un "setter" (cambiar).
- **Navegación:** Al asignarle un valor, el navegador interpreta que debe cambiar de página.

#### 2. Funcionamiento paso a paso:

- **Asignación:** Se le da un nuevo string con una URL.
- **Petición:** El navegador corta la conexión actual.
- **Carga:** Se inicia la carga del nuevo destino.
- **Resultado:** El usuario aterriza en la nueva web.

## EL OBJETO history

1. El método **history.go()**: Carga una página específica del historial de sesión, basada en su posición relativa a la actual.

### Ejemplo

```
history.go(-1);
```

```
// Retrocede una página (igual que el botón atrás).
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Desplazamiento:** Usa números enteros para moverse (negativos atrás, positivos adelante).
- **Relatividad:** 0 recarga la página actual.

### 2. Funcionamiento paso a paso:

- **Lectura:** El navegador busca en su memoria de pestañas la posición -1.
- **Validación:** Si existe una página previa, inicia la vuelta.
- **Resultado:** La ventana muestra el contenido de la URL anterior.

## EL OBJETO navigator

### 1. Propiedad navigator.onLine: Indica si el navegador tiene conexión a la red en este momento.

#### Ejemplo

```
if (navigator.onLine) {  
    console.log("Conectado");  
} else {  
    alert("Revisa tu conexión");  
}
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Estado:** Es una propiedad de solo lectura.
- **Booleano:** Devuelve true si hay conexión y false si no.

### 2. Funcionamiento paso a paso:

- **Consulta:** El sistema operativo informa al navegador sobre el estado del hardware de red.
- **Actualización:** El navegador actualiza esta propiedad en tiempo real.
- **Resultado:** Permite a la web actuar de forma distinta si el usuario pierde el Wi-Fi.

## EL OBJETO screen

### 1. Propiedad screen.availWidth: Devuelve el ancho de la pantalla del usuario, excluyendo las barras de tareas del sistema operativo.

#### Ejemplo

```
const anchoUtil = screen.availWidth;  
console.log(anchoUtil);  
// Ejemplo: 1920 (en un monitor Full HD)
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Medición:** Detecta los píxeles totales del monitor.
- **Resta Inteligente:** A diferencia de screen.width, este resta el espacio que ocupan cosas como la barra de Windows o el Dock de Mac.

### 2. Funcionamiento paso a paso:

- **Cálculo:** El navegador pregunta al sistema operativo por el tamaño del escritorio "limpio".
- **Resultado:** Devuelve un número entero que representa el espacio máximo donde una ventana podría expandirse sin tapar la barra de tareas.

### 1. El método alert(): Muestra un cuadro de diálogo de aviso con un mensaje y un botón de aceptar.

#### Ejemplo

```
window.alert("Sesión expirada");  
// Muestra el mensaje y detiene el código hasta que se cierra.
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Bloqueo Síncrono:** Detiene por completo la ejecución de la página y la interacción del usuario hasta que se acepta.
- **Propósito:** Notificar información crítica que no requiere respuesta del usuario más que su lectura.

### 2. Funcionamiento paso a paso:

- **Llamada:** Se invoca el método con un string.
- **Congelación:** El navegador pausa el renderizado y los scripts.
- **Cierre:** El usuario pulsa "Aceptar".
- **Continuación:** El script sigue en la línea inmediatamente inferior.

**2. El método prompt():** Muestra un cuadro de diálogo que solicita una entrada de texto al usuario.

## Ejemplo

```
const nombre = prompt("¿Cómo te llamas?", "Invitado");
// Si escribe "Alex" -> nombre = "Alex"
// Si cancela -> nombre = null
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Entrada de datos:** Recibe dos argumentos: el mensaje y un valor por defecto (opcional).
- **Retorno dinámico:** Devuelve un string si el usuario acepta, o null si cancela.

### 2. Funcionamiento paso a paso:

- **Apertura:** Aparece la ventana con un campo de texto.
- **Interacción:** El usuario escribe y pulsa OK o Cancelar.
- **Resultado:** Se asigna el valor escrito a la variable.

**3. El método open():** Abre una nueva ventana del navegador o una pestaña.

## Ejemplo

```
window.open("https://google.com", "_blank");
// Abre Google en una pestaña nueva.
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Navegación externa:** Permite abrir URLs fuera de la página actual.
- **Control de destino:** \_blank (nueva pestaña) o \_self (misma pestaña).

### 2. Funcionamiento paso a paso:

- **Petición:** El script solicita al navegador abrir una URL.
- **Seguridad:** El navegador verifica si fue una acción iniciada por el usuario (para evitar spam).
- **Resultado:** Se crea el nuevo contexto de navegación.

**4. El método scrollTo():** Desplaza la ventana a un conjunto específico de coordenadas

**El método blur() y focus():** Controlan si la ventana del navegador es la "activa" o no para el sistema operativo.

## Ejemplo

```
window.blur(); // Quita el foco de la ventana
window.focus(); // Devuelve el foco a la ventana
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Estado de Ventana:** Gestiona si la ventana está en primer plano (focus) o se envía al fondo (blur).
- **Limitación:** Por seguridad, muchos navegadores modernos solo permiten focus() si la ventana fue abierta por un script (window.open).

### 2. Funcionamiento paso a paso:

- **Llamada:** El script solicita el cambio de estado.
- **Interacción:** El navegador intenta resaltar la pestaña o enviarla detrás de otras aplicaciones.
- **Resultado:** La ventana gana o pierde la capacidad de recibir eventos de teclado inmediatamente.

**El método getComputedStyle():** Extrae los valores reales de CSS de un elemento después de que el navegador haya aplicado todas las hojas de estilo y cálculos.

## Ejemplo

```
const elemento = document.querySelector("h1");
const colorReal = window.getComputedStyle(elemento).color;
// "rgb(255, 0, 0)"
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Lectura Final:** A diferencia de elemento.style, que solo lee estilos en línea, este lee el CSS externo y calculado (ej: convierte 1em a píxeles reales).
- **Solo lectura:** El objeto devuelto no se puede modificar.

### 2. Funcionamiento paso a paso:

- **Entrada:** Recibe el nodo del DOM.
- **Cálculo:** El motor de renderizado suma el CSS del archivo, el estilo en línea y la herencia.
- **Resultado:** Devuelve un objeto con todas las propiedades CSS finales.

**Los métodos moveBy() y moveTo():** Cambian la posición de la ventana en la pantalla del usuario.

## Ejemplo

```
window.moveTo(100, 100); // Se mueve a la coordenada absoluta (100,100)
window.moveBy(50, 50); // Se mueve 50px más desde donde está ahora
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?

- **Coordenadas:** moveTo usa el eje X/Y de la pantalla. moveBy suma píxeles a la posición actual.

### 2. Funcionamiento paso a paso:

- **Inicio:** Lee la posición actual de la ventana.
- **Cálculo:** Suma (en moveBy) o reemplaza (en moveTo) las coordenadas.
- **Resultado:** La ventana se desplaza físicamente por el monitor.

**El método print():** Activa la interfaz de impresión del navegador.

## Ejemplo

```
window.print();
```

## Desglose paso a paso:

### 1. ¿Cómo funciona?: Captura de Pantalla:

Genera una versión del documento actual optimizada para papel (usando @media print si existe).

## 2. Funcionamiento paso a paso:

- **Llamada:** Se pausa la ejecución del script.
- **Interfaz:** Aparece la ventana nativa del sistema (PDF, Impresora).
- **Resultado:** El usuario decide si imprimir o cancelar.

**Los métodos `resizeBy()` y `resizeTo()`:** Cambian el ancho y alto de la ventana del navegador.

### Ejemplo

```
window.resizeTo(800, 600); // La ventana pasa a medir exactamente 800x600
```

### Desglose paso a paso:

1. **¿Cómo funciona?: Redimensionamiento:** Ajusta el marco exterior del navegador.

## 2. Funcionamiento paso a paso:

- **Solicitud:** El script pide un nuevo tamaño.
- **Validación:** El navegador verifica que no sea una ventana "fija".
- **Resultado:** Los elementos internos se reajustan (responsive design) al nuevo ancho/alto.

**El método `stop()`:** Detiene la carga de los recursos de la página.

### Ejemplo

```
window.stop();
```

### Desglose paso a paso:

1. **¿Cómo funciona?: Interrupción:** Es el equivalente programático a pulsar el botón "X" del navegador mientras carga.

## 2. Funcionamiento paso a paso:

- **Ejecución:** Se cortan las descargas de imágenes, scripts pendientes o frames.
- **Resultado:** La página se queda tal cual estaba en el milisegundo en que se llamó la función.

## EL OBJETO `location` (MÉTODOS)

### `location.reload()`

Recarga la página actual.

- **Paso a paso:** El navegador vacía el contenido actual y vuelve a pedir la URL al servidor.

### `location.replace(url)`

Reemplaza la URL actual por una nueva **sin dejar rastro en el historial**.

- **Paso a paso:** El usuario no puede pulsar el botón "Atrás" para volver a la página anterior porque esta fue "pisada" por la nueva.

## EL OBJETO `screen` (PROPIEDADES)

### `screen.width / screen.height`

Devuelven el tamaño total del monitor.

- **Paso a paso:** El sistema informa al navegador de la resolución física (ej: 1920x1080).

## EL OBJETO `navigator` (PROPIEDADES)

### `navigator.userAgent`

Devuelve el string de identificación del navegador.

- **Paso a paso:** El navegador consulta su propia versión y la del sistema operativo y la devuelve en un solo texto largo.

### `navigator.language`

Devuelve el idioma preferido del usuario.

- **Paso a paso:** Lee la configuración del sistema (ej: "es-ES") para que la web pueda traducirse automáticamente.

## Resumen Visual de la Estructura Aplicada

Objeto	Función	Estado Original	Resultado Final
window	confirm()	Script corriendo	Pausa -> Booleano
location	reload()	Página cargada	Página reiniciada
history	back()	URL actual	URL anterior
navigator	onLine	Desconocido	True/False

## DOM: EL MOTOR DE LA WEB DINÁMICA

El **DOM** es el puente. Por un lado tienes el **HTML** (estático, un archivo de texto) y por el otro el **Navegador** (un software que renderiza visualmente). El DOM es la estructura intermedia que permite que JavaScript "toque" el contenido.

### El Concepto de "Nodo" (La Unidad Básica)

**En el DOM, todo es un nodo. No solo las etiquetas:**

- **Element Node:** Las etiquetas (<div>, <a>, <h1>).
- **Text Node:** El texto dentro de las etiquetas.
- **Attribute Node:** Los atributos como class="rojo" o src="foto.jpg".

**Dato Clave:** Si tienes <div>Hola</div>, el div es un nodo de elemento y "Hola" es un **nodo de texto** hijo del div.

## Selección Avanzada: El "Escáner" del DOM

Cuando usas métodos de búsqueda, el navegador realiza un recorrido por el árbol:

Método	Tipo de Selector	Velocidad	Qué devuelve
getElementById()	ID (#id)	🚀 Ultra rápido	Un único elemento.
querySelector()	CSS (.clase, tag > li)	⚡ Rápido	El <b>primer</b> elemento que coincide.
querySelectorAll()	CSS (div.card)	🐢 Normal	Una <b>NodeList</b> (parecido a un array).

### Proceso Interno de querySelector:

1. **Llamada:** Ejecutas document.querySelector(".btn-save").
2. **Búsqueda:** El motor del navegador empieza desde el nodo document y baja rama por rama.
3. **Coincidencia:** En cuanto encuentra el primer elemento con la clase btn-save, detiene la búsqueda.
4. **Retorno:** Te entrega una **referencia** (un puntero) a ese objeto en memoria.

## Manipulación Proactiva (Cambiar la Realidad)

Una vez que tienes el elemento en una variable, puedes "operarlo":

### A. Propiedades de Contenido

- **.textContent:** Solo texto. Es el más seguro porque **no interpreta** HTML. Si intentas meter un <b>, lo escribirá literalmente en pantalla.
- **.innerHTML:** El "maestro constructor". Interpreta el texto como código HTML. Útil pero peligroso (riesgo de ataques XSS).

### B. La API de Estilos (.style)

JavaScript no edita el archivo .css, sino que añade **estilos en línea** (inline styles) al elemento, los cuales tienen la prioridad más alta en CSS.

```
const box = document.querySelector(".caja");
box.style.borderRadius = "50%"; // Nota: camelCase en vez de kebab-case
box.style.transition = "all 0.5s";
```

## El Ciclo de Vida: Crear y Destruir

Imagina que recibes datos de una API y quieres mostrarlos. El proceso es como armar un mueble de IKEA:

1. **createElement()**: Fabricas la pieza en el taller (memoria).
  - o const card = document.createElement("div");
2. **Personalizar**: Le das forma y color.
  - o card.className = "card-active";
  - o card.textContent = "Usuario Conectado";
3. **appendChild() / prepend()**: Lo colocas en la casa (el documento).
  - o document.body.appendChild(card); // Lo pone al final.

## Eventos: La Respuesta al Usuario

El DOM no solo es visual, también es "sensible". Puedes decirle a un nodo que "escuche" lo que pasa:

JavaScript

```
const boton = document.querySelector("#miBtn");
```

```
// Sintaxis: elemento.addEventListener(evento, función)
```

```
boton.addEventListener("click", () => {  
    console.log("¡El DOM ha detectado un clic!");  
    document.body.style.backgroundColor = "black";  
});
```

## Ejemplo Maestro: "Lista de Tareas Dinámica"

Supongamos que el usuario escribe "Comprar pan" y pulsa Enter:

1. **Captura**: Se lee el valor del <input>.
2. **Creación**: Se crea un <li>.
3. **Inyección**: Se le asigna el texto del input al li.
4. **Vinculación**: Se añade un botón de "Borrar" dentro del li.
5. **Renderizado**: Se hace listaUl.appendChild(li).
6. **Resultado**: El DOM se actualiza y el navegador dibuja la nueva tarea sin que la página parpadee.

## EVENTOS EN JAVASCRIPT

Los eventos son la base de la **interactividad**. Son "señales" que el navegador emite cuando algo sucede (un clic, una tecla presionada o la carga de una imagen) para que JavaScript pueda reaccionar a ello.

### 1. Anatomía de un Evento

Para que un evento funcione, necesitamos tres elementos:

1. **Target (Objetivo)**: El elemento HTML que recibirá la acción (ej. un <button>).
2. **Trigger (Disparador)**: La acción física o del sistema (ej. click).
3. **Handler (Manejador)**: La función de JavaScript que se ejecuta como respuesta.

### 2. El Objeto event (La caja negra)

Cada vez que ocurre un evento, el navegador crea automáticamente un objeto **event** (abreviado a menudo como e) que contiene todos los detalles de lo sucedido.

- **e.target**: ¿Quién originó el evento? (Ej: el botón específico que pulsaste).
- **e.type**: ¿Qué pasó? (Ej: "mousedown").
- **e.key**: En eventos de teclado, ¿qué letra se pulsó?
- **e.preventDefault()**: Detiene la acción natural (ej. evita que un link te lleve a otra página o que un formulario se envíe).

### 3. Clasificación de Eventos Principales

#### A. Eventos de Ratón (Mouse): Controlan la interacción física con el puntero.

- **onclick / ondblclick:** Clic simple o doble.
- **onmouseover / onmouseout:** Cuando el puntero entra o sale del área de un elemento (ideal para menús desplegables).
- **onmousemove:** Se dispara cada milímetro que mueves el ratón. Permite obtener coordenadas clientX y clientY.

#### B. Eventos de Teclado: Ideales para juegos o atajos de teclado.

- **onkeydown:** Se activa al hundir la tecla.
- **onkeyup:** Se activa al soltarla. Es el mejor momento para validar lo que el usuario escribió.

#### C. Eventos de Formulario y Estado

- **onsubmit:** Fundamental para validar datos antes de enviarlos al servidor.
- **onchange:** Se activa cuando cambias el valor (muy usado en <select>).
- **onfocus / onblur:** Cuando entras o sales de un campo de texto (para iluminar el borde del input, por ejemplo).

### 4. Funcionamiento Paso a Paso: onsubmit

Imagina un formulario de registro:

1. **Usuario hace clic en "Enviar".**
2. **Disparo:** El evento onsubmit se activa en el formulario.
3. **Validación:** JavaScript ejecuta una función que revisa si el campo nombre está vacío.
4. **Decisión:** \* Si está vacío: Se ejecuta return false o e.preventDefault(). El formulario **no se envía**.
  - Si es correcto: Se ejecuta return true. El formulario viaja al servidor.

### 5. El Estándar Moderno: addEventListener()

Aunque existen los atributos HTML (como onclick=""), la forma profesional de trabajar es usar el "Escuchador de Eventos".

#### ¿Por qué es mejor?

- **Limpieza:** El HTML queda puro, sin código JS mezclado.
- **Multiplicidad:** Puedes añadir 10 funciones distintas al mismo clic de un solo botón. Los atributos onclick solo permiten una.

#### Ejemplo de flujo:

JavaScript

```
const btn = document.querySelector("#miBoton");
```

```
// 1. Escuchar el evento
```

```
// 2. Definir la acción
```

```
btn.addEventListener("click", (e) => {
    console.log("Botón pulsado en la posición: " + e.clientX);
});
```

#### Resumen de Comportamientos

Categoría	Evento	Momento de ejecución
Ventana	onload	Cuando todo (fotos incluidas) terminó de cargar.
Interfaz	onresize	Cada vez que el usuario estira o encoge la ventana.
Seguridad	onerror	Si un archivo (imagen/script) no se encuentra.
Formulario	onreset	Cuando el usuario limpia los campos del formulario.

## Conclusión

Los eventos transforman una página estática en una **aplicación**. Sin ellos, no habría buscadores que sugieran palabras mientras escribes, ni carritos de compra que se actualizan al instante.

## EXPRESIONES REGULARES (RegEx)

Las **Expresiones Regulares** son, en esencia, un lenguaje de programación en miniatura dentro de JavaScript. Se basan en la teoría de lenguajes formales para definir "patrones de búsqueda" que permiten realizar operaciones complejas sobre cadenas de texto que serían imposibles con simples comparaciones.

### 1. Fundamento Teórico: ¿Qué es un patrón?

Un patrón es una **abstracción**. En lugar de buscar la palabra "2025", buscas "cuatro dígitos seguidos". Esto permite que tu código sea flexible y pueda procesar datos que aún no conoce (como el email que un usuario inventará mañana).

Existen dos formas de instanciarlas:

1. **Literal (/patrón/)**: Se compila cuando se evalúa el script. Es más eficiente si el patrón no cambiará.
2. **Constructor (new RegExp())**: Se compila en tiempo de ejecución. Es ideal cuando el patrón depende de una variable (ej. lo que el usuario escribe en un buscador).

### 2. Anclas y Metacaracteres: Definiendo el Territorio

Para que una RegEx sea precisa, debemos decirle **dónde** buscar.

- **Anclas (^ y \$)**: No representan caracteres, sino posiciones. ^ asegura que el texto empiece ahí y \$ que termine ahí. Sin ellas, /hola/ encontrará "hola" dentro de "pachola", lo cual podría ser un error en una validación.
- **Clases de Caracteres**: Son "atajos" para grupos comunes.
  - \d: (Digital) Equivale a [0-9].
  - \w: (Word) Caracteres alfanuméricos y guion bajo.
  - \s: (Space) Espacios, tabulaciones y saltos de línea.

### 3. Cuantificadores: La Lógica de la Repetición

Los cuantificadores definen la **codicia** (greediness) del patrón. Por defecto, intentan encontrar la cadena más larga posible que coincida.

- **\*** (**Cero o más**): Es opcional y puede repetirse infinitamente.
- **+** (**Uno o más**): Es obligatorio al menos una vez.
- **?** (**Cero o uno**): Hace que un elemento sea opcional (ej. /https?/ coincide con "http" y "https").
- **{n,m}**: El control total. Permite definir rangos exactos, como en el caso de los códigos postales o años.

### 4. Agrupamiento y Captura ()

Los paréntesis hacen dos cosas:

1. **Agrupar**: Aplicar un cuantificador a todo un bloque. Ej: /(abc)+/ busca "abcabcabc".
2. **Capturar**: Guardar lo encontrado en una "memoria temporal". Si buscas /(\d{2})-(\d{2})/, el motor guarda el primer par de números en una posición y el segundo en otra, permitiéndote extraer partes específicas de un texto (como el día y el mes de una fecha).

### 5. Flags: Modificando el Comportamiento

Los flags cambian las reglas del juego para todo el patrón:

- **i (case-insensitive)**: Elimina la distinción entre "A" y "a".
- **g (global)**: No se detiene en la primera coincidencia; busca en todo el texto.
- **m (multiline)**: Hace que ^ y \$ funcionen línea por línea en lugar de considerar todo el bloque de texto como uno solo.

## 6. Métodos de búsqueda: ¿Cuál elegir?

Método	Origen	Propósito
<code>test()</code>	RegEx	<b>Validar.</b> Devuelve true/false. Es el más rápido.
<code>exec()</code>	RegEx	<b>Extraer.</b> Devuelve un objeto detallado con grupos de captura.
<code>match()</code>	String	<b>Listar.</b> Devuelve un array con todas las coincidencias (si usas /g).
<code>replace()</code>	String	<b>Transformar.</b> Sustituye lo encontrado por otro texto o lógica.
<code>split()</code>	String	<b>Dividir.</b> Corta el texto usando el patrón como tijeras.

### Ejemplo Desarrollado: Validación de Contraseña

Queremos una contraseña que:

1. Tenga al menos una mayúscula (`?=.*[A-Z]`) (esto es un *lookahead* o búsqueda hacia adelante).
2. Tenga al menos un número (`?=.*\d`).
3. Mínimo 8 caracteres `{8,}`.

**Patrón:** `^(?=.*[A-Z])(?=.*\d){8,}$`

1. `^`: Empezamos aquí.
2. `(?=.*[A-Z])`: Mira hacia adelante y asegúrate de que haya una mayúscula en algún lugar.
3. `(?=.*\d)`: Mira hacia adelante y asegúrate de que haya un número.
4. `{8,}`: Ahora sí, consume al menos 8 caracteres de cualquier tipo.
5. `$`: Aquí debe terminar.

### Cuadro Resumen de Lógica RegEx

Símbolo	Teoría	Aplicación Real
<code>.</code>	Comodín absoluto	Cualquier carácter excepto salto de línea.
<code>[a-z]</code>	Rango definido	Cualquier letra minúscula del alfabeto.
<code>\`</code>	<code>\`</code>	Operador lógico OR
<code>\</code>	Carácter de escape	Para buscar un punto real <code>\.</code> o un signo <code>\?.</code>

### Conclusión

Las RegEx no se deben "memorizar", sino "construir". Son la herramienta más potente para la limpieza de datos (data scrubbing) y validación de seguridad en el cliente antes de enviar información al servidor.

## JQUERY: LA LIBRERÍA "ESCRIBE MENOS, HAZ MÁS"

**jQuery** no es un lenguaje nuevo, es una **librería de JavaScript**. En su momento de gloria, resolvió el mayor dolor de cabeza de los desarrolladores: la inconsistencia entre navegadores (lo que funcionaba en Chrome fallaba en Internet Explorer). jQuery creó una capa de abstracción que unificó la web.

### 1. Filosofía y el Selector Maestro \$

La genialidad de jQuery reside en el objeto `$`. Es un "envoltorio" (wrapper) que convierte elementos simples del DOM en **Objetos jQuery**, otorgándoles superpoderes (métodos) que el JS nativo no tenía de forma sencilla.

- **Selección potente:** Utiliza selectores de CSS para encontrar elementos.
  - `$(".clase")`: Selecciona por clase.
  - `$("#id")`: Selecciona por ID.
  - `$("p:first")`: Selecciona solo el primer párrafo.

## 2. El Ciclo de Vida: `$(document).ready()`

Históricamente, si ponías tu script al principio del HTML, fallaba porque los elementos aún no existían.

- **Teoría:** jQuery introdujo este método para asegurar que el código solo se ejecute cuando el **árbol del DOM** esté completamente construido en memoria, aunque las imágenes pesadas sigan cargando.
- **Hoy:** Es el estándar para evitar errores de "elemento no encontrado".

## 3. Manipulación Dinámica del DOM

jQuery hace que modificar la estructura de la página sea como jugar con bloques de construcción:

Acción	Método jQuery	Equivalente Teórico
Cambiar Texto	<code>.text("Hola")</code>	Modifica el nodo de texto.
Cambiar HTML	<code>.html("&lt;b&gt;!&lt;/b&gt;")</code>	Renderiza etiquetas nuevas.
Añadir después	<code>.append()</code>	Inserta un hijo al final del contenedor.
Añadir antes	<code>.prepend()</code>	Inserta un hijo al inicio.
Eliminar	<code>.remove()</code>	Arranca el elemento del árbol.

## 4. Animaciones y la "Cola de Efectos"

A diferencia del CSS, las animaciones de jQuery son **programáticas**.

- **Encadenamiento (Chaining):** Puedes poner una acción tras otra:  
`$("#caja").slideUp().fadeIn().animate({left: '100px'});`
- **Funcionamiento:** jQuery crea una "cola" (queue) interna. Ejecuta la primera animación y, al terminar, dispara la siguiente automáticamente.

## 5. AJAX Simplificado: Comunicación Silenciosa

Antes de `fetch()`, hacer una petición al servidor era un proceso largo y complejo. jQuery lo redujo a una función:

1. **Petición:** Envía una señal al servidor (URL).
2. **Espera:** El navegador sigue funcionando (asíncrono).
3. **Callback (success):** Cuando el servidor responde, se ejecuta la función que procesa los datos y actualiza el DOM sin parpadeos.

**Dato:** El método `.load()` es único porque no solo trae datos, sino que los inyecta directamente en el HTML seleccionado en una sola línea.

## 6. Delegación de Eventos con `.on()`

Esta es la forma más avanzada de manejar eventos.

- **Problema:** Si creas un botón nuevo dinámicamente con JS, el click normal no funcionará porque el botón no existía cuando cargó la página.
- **Solución:** `.on()` escucha al "padre" y detecta cuando el evento ocurre en un "hijo" nuevo.
  - `$("#contenedor").on("click", ".btn-nuevo", function(){...});`

## Conclusión

Aunque los frameworks modernos (React, Vue, Angular) han desplazado a jQuery, entenderlo es vital por dos razones:

1. **Legado:** Millones de sitios web (incluyendo WordPress) dependen de él.
2. **Productividad:** Para prototipos rápidos o pequeñas interacciones, sigue siendo una herramienta de "navaja suiza" sumamente veloz.

Aquí tienes el desarrollo completo y detallado de los últimos temas solicitados: **Cookies** y **Formularios**, estructurados paso a paso, con profundidad técnica y claridad conceptual.

## COOKIES: PERSISTENCIA Y MEMORIA EN LA WEB

Las **cookies** son pequeñas piezas de datos que el servidor envía al navegador. Debido a que el protocolo HTTP es "stateless" (sin estado), las cookies son el pegamento que permite que un sitio web reconozca que eres el mismo usuario que hizo clic en la página anterior.

### 1. Anatomía de una Cookie

Una cookie no es solo un nombre y un valor; es un objeto con propiedades que definen su vida y seguridad:

- **Nombre=Valor:** La información real (ej: theme=dark).
- **Expires/Max-Age:** Define cuándo muere la cookie. Si no se pone, es una **cookie de sesión** (muere al cerrar el navegador).
- **Path:** Define en qué rutas del sitio es válida (ej: path=/admin).
- **Domain:** Define si es válida para subdominios.

### 2. Gestión con JavaScript: document.cookie

A diferencia de otras variables, document.cookie es un **descriptor de acceso especial**. No puedes borrar todas las cookies asignando un string vacío; debes gestionarlas una a una.

#### A. Creación y Ciclo de Vida

Para que una cookie dure más allá de la sesión actual, debemos convertir una fecha a formato UTC:

```
let expiracion = new Date();
expiracion.setTime(expiracion.getTime() + (30 * 24 * 60 * 60 * 1000)); // 30 días
document.cookie = "id_usuario=12345; expires=" + expiracion.toUTCString() + "; path=/";
```

#### B. El desafío de la Lectura

Cuando pides document.cookie, obtienes un solo string con todas las cookies separadas por ;.

Por eso, la función **getCookie()** es vital:

1. **Divide** el string por cada punto y coma.
2. **Limpia** los espacios en blanco sobrantes con .trim().
3. **Compara** el inicio de cada trozo con el nombre que buscas.

#### C. Seguridad Esencial

- **Secure:** La cookie solo viaja sobre HTTPS.
- **SameSite=Strict:** Evita que la cookie se envíe en peticiones desde otros sitios (prevención de ataques CSRF).

## FORMULARIOS: VALIDACIÓN Y CONTROL DE DATOS

El formulario es el principal punto de entrada de datos. JavaScript actúa como un **filtro de calidad** para asegurar que la información que llega al servidor sea correcta, ahorrando tiempo y recursos.

### 1. Acceso Jerárquico al DOM

Existen tres formas de llegar a un dato de formulario, de la más antigua a la más moderna:

1. **Colección legacy:** document.forms[0].elements[0].value (Muy frágil si cambia el orden).
2. **Acceso por Name:** document.miForm.email.value (Legible y común).
3. **ID directo:** document.getElementById("email").value (Preciso y estándar hoy).

## 2. El Proceso de Validación Paso a Paso

Cuando un usuario pulsa "Enviar", ocurre lo siguiente:

1. **Evento onsubmit:** El formulario lanza una señal de intento de envío.
2. **Captura:** JavaScript intercepta la señal.
3. **Evaluación:** Se ejecutan los tests (¿Está vacío?, ¿Es un email?, ¿Aceptó términos?).
4. **Bloqueo/Paso:** \* Si algo falla: Se muestra un mensaje y se devuelve false (el formulario se queda estático).
  - o Si todo es correcto: Se devuelve true y los datos viajan al servidor.

## 3. Técnicas de Validación Avanzada

### A. Verificación de Tipos (isNaN)

No basta con que el campo tenga texto; a veces necesitamos números. isNaN() (Is Not a Number) es la herramienta clave:

JavaScript

```
if (isNaN(campoEdad.value)) {  
    alert("Por favor, introduce un número válido.");  
}
```

### B. Validación de Fechas Reales

Un error común es validar que algo parezca una fecha (XX/XX/XXXX) pero sea inválida (31/02/2024).

- **Técnica:** Se crea un objeto new Date() con los datos del usuario y se compara si el día, mes y año resultantes coinciden con los ingresados. Si el objeto Date "autocorrege" la fecha (ej. convierte 32 de enero en 1 de febrero), sabemos que la fecha era falsa.

### C. Expresiones Regulares (RegEx)

Es el estándar de oro para emails y contraseñas. El patrón `/[^@\s@]+@[^\s@]+\.[^\s@]+\$/` asegura que haya un texto, un @, otro texto, un . y una extensión.

## 4. Usabilidad y Estado del Formulario

Un buen desarrollador no solo valida, sino que guía:

- **focus() / blur():** Resalta campos mientras se escriben.
- **disabled:** Bloquea el botón de envío hasta que todos los campos obligatorios estén llenos.
- **replaceState:** Una técnica crucial para que, si el usuario refresca la página después de enviar, el navegador no le pregunte "¿Desea reenviar los datos?".

### Tabla Comparativa de Validación

Tipo de Validación	Herramienta JS	Propósito
Presencia	<code>value === ""</code>	Asegurar que el campo no esté vacío.
Formato	<code>RegExp.test()</code>	Validar emails, teléfonos, códigos postales.
Lógica	<code>checked</code>	Confirmar aceptación de términos o selección.
Numérica	<code>parseInt() / isNaN()</code>	Validar edades, precios o cantidades.

**Conclusión General:** Tanto las **Cookies** como los **Formularios** gestionan la relación entre el usuario y los datos. Las primeras mantienen la identidad a través del tiempo, mientras que los segundos aseguran que la comunicación sea limpia y segura.