Fundamentos de JavaScript: Variables, Operadores, Control de Flujo y Objetos del Navegador

1. JavaScript: Fundamentos del Lenguaje Principal del Entorno Cliente

JavaScript es la piedra angular del desarrollo web moderno en el lado del cliente. Creado originalmente en Netscape y estandarizado posteriormente a través de la especificación **ECMA-262**, es el lenguaje que permite transformar páginas HTML estáticas en aplicaciones interactivas y dinámicas.

> **Note**
>
> **¿Qué es JavaScript en términos simples?** JavaScript es un lenguaje de programación, o lo que es lo mismo, un mecanismo con el que podemos decirle a nuestro navegador qué tareas debe realizar, en qué orden y cuántas veces. Mientras que HTML define la estructura y CSS el estilo, JavaScript añade el **comportamiento** e **interactividad**.

1.1. Orígenes, características y desafíos

JavaScript fue creado por Brendan Eich en Netscape en solo **10 días** en 1995, inicialmente bajo los nombres de Mocha y LiveScript. A pesar de su nombre, **no tiene relación con Java**; fue una decisión de marketing. Su estandarización a través de **ECMA-262** asegura la compatibilidad entre diferentes navegadores.

¿Qué puede hacer JavaScript?

- Manipular el contenido de una página (DOM)
- Reaccionar a eventos del usuario (clics, teclas, scroll)
- Validar formularios antes de enviarlos
- Crear animaciones y efectos visuales
- Comunicarse con servidores (AJAX/Fetch)
- Almacenar datos en el navegador (localStorage)

Características clave:

- **AJAX (Asynchronous JavaScript and XML)**: Permite comunicarse con el servidor en segundo plano (sin recargar la página), solicitando o enviando datos para actualizar solo una parte de la interfaz. Hoy se usa casi siempre con **JSON**, no con XML.
- **Orientación a eventos**: Reacciona a interacciones del usuario, controla ventanas, valida datos en el cliente y modifica dinámicamente el DOM.
- **No reemplaza al servidor**: JavaScript no puede escribir archivos directamente en el servidor ni acceder a recursos del sistema del usuario.

Aunque el código JavaScript es interpretado por el cliente, impone desafíos de compatibilidad y seguridad:

1. **Compatibilidad**: Se debe lidiar con las disparidades de implementación entre navegadores (Chrome, Firefox, Safari) y plataformas, lo que exige pruebas exhaustivas.
2. **Seguridad (Modelo "Sandbox")**: El navegador impone un estricto modelo de seguridad con dos restricciones clave:
    - **Espacio Seguro de Ejecución**: JavaScript opera en un entorno aislado que le impide realizar tareas del sistema, como leer archivos locales.

- **Política de "Mismo Origen" (Same-origin policy)**: Un script de un dominio no puede acceder a información sensible (como cookies) de otro dominio diferente.

1.2. Integración y prácticas del Código en el Cliente

Para una programación eficaz, es crucial la correcta integración del código. Para la integración con HTML, existen dos métodos:

1. **Código en línea**: Utilizando las etiquetas `<script> ... </script>` para escribir el código directamente en el archivo HTML.
2. **Archivo Externo**: Enlazando un archivo `.js` externo mediante `<script src="tucodigo.js"></script>`. **Este es el método recomendado**, ya que promueve la modularidad, facilita la reutilización, mejora el rendimiento (gracias al caché) y mantiene una clara separación entre estructura (HTML) y comportamiento (JS).

Los comentarios en el código para documentar el código es una práctica fundamental. JavaScript admite dos tipos de comentarios:

- `// Comentario de una sola línea`: Ignora todo el texto hasta el final de la línea.
- `/* Comentario de bloque */`: Ignora todo el texto entre símbolos de apertura y cierre, lo que permite comentarios de varias líneas y la desactivación temporal de bloques de códigos.

1.2.1. Variables, Tipos de Datos y Ámbito

Una **variable** es un contenedor con nombre para almacenar datos. Piensa en ella como una "caja etiquetada" donde guardas información que puedes usar más tarde.

> javascript

```javascript
// Declaración y asignación de variables
let nombre = "Ana";        // Variable que puede cambiar
const PI = 3.14159;        // Constante que no cambia
let edad = 25;             // Número
let esEstudiante = true;   // Booleano
```

En JavaScript moderno, existen tres formas de declarar variables:

Característica	`var`	`let`	`const`
Ámbito (Scope)	Función/Global	Bloque (`{}`)	Bloque (`{}`)
Re-declaración	Permitida	No permitida	No permitida
Modificación	Permitida	Permitida	No permitida
Hoisting (Elevación)	Se eleva y se inicializa como `undefined`	Se eleva al inicio del bloque pero no se inicializa. Acceder antes de la declaración da `ReferenceError`.	Se eleva al inicio del bloque, pero no se inicializa. Acceder antes de la declaración da `ReferenceError`.

> **Tip**
>
> **Regla moderna**: Usa `const` por defecto, `let` cuando necesites reasignar, y **nunca** uses `var` en código nuevo.

Tipado Dinámico

JavaScript es un lenguaje de **tipado dinámico**: las variables no tienen un tipo fijo y pueden cambiar de tipo durante la ejecución.

> javascript

```javascript
let dato = 42;          // Ahora es un número
dato = "Hola";          // Ahora es un string (¡válido en JS!)
dato = true;            // Ahora es un booleano
```

Los tipos de datos principales son:

Tipo	Descripción	Ejemplo
`String`	Secuencia de caracteres (texto)	`"Hola"`, `'Mundo'`
`Number`	Números enteros y decimales	`42`, `3.14`, `-7`
`Boolean`	Valor lógico verdadero/falso	`true`, `false`
`null`	Ausencia **intencionada** de valor	`let x = null;`
`undefined`	Variable declarada sin valor	`let x;`
`Object`	Colección de pares clave-valor	`{nombre: "Ana"}`
`Array`	Lista ordenada (tipo especial de Object)	`[1, 2, 3]`
`Function`	Bloque de código ejecutable	`function() {}`

Coerción de tipos (conversión automática)

JavaScript realiza conversiones de tipo automáticas, lo cual puede causar resultados inesperados:

> javascript

```javascript
// Concatenación (+ con string convierte todo a string)
4 + "5"           // "45" (número → string)
"Precio: " + 100  // "Precio: 100"

// Conversión explícita (recomendado)
parseInt("89.76", 10)   // 89 (string → entero)
parseFloat("89.76")     // 89.76 (string → decimal)
String(42)              // "42" (número → string)
Number("42")            // 42 (string → número)
Boolean(0)              // false (número → booleano)
```

1.2.2. Panorama General de Operadores

Los **operadores** son símbolos especiales que realizan operaciones sobre valores (llamados **operandos**) y devuelven un resultado.

Operadores Aritméticos

Realizan cálculos matemáticos. El operador `+` también sirve para concatenar strings.

> javascript

```
5 + 3      // 8 (suma)
10 - 4     // 6 (resta)
3 * 4      // 12 (multiplicación)
15 / 3     // 5 (división)
17 % 5     // 2 (módulo: resto de la división)
2 ** 3     // 8 (exponenciación: 2³ - ES2016)

// Incremento y decremento
let x = 5;
x++;       // x ahora es 6 (post-incremento)
++x;       // x ahora es 7 (pre-incremento)
x--;       // x ahora es 6 (post-decremento)

// Concatenación con +
"Hola" + " " + "Mundo"  // "Hola Mundo"
```

Operadores de Asignación

Asignan un valor a una variable. Los operadores compuestos realizan una operación y asignación en un solo paso.

> javascript

```
let a = 10;     // Asignación simple

a += 5;   // a = a + 5  → a es 15
a -= 3;   // a = a - 3  → a es 12
a *= 2;   // a = a * 2  → a es 24
a /= 4;   // a = a / 4  → a es 6
a %= 4;   // a = a % 4  → a es 2
a **= 3;  // a = a ** 3 → a es 8
```

Operadores de Comparación

Comparan dos valores y devuelven un booleano (`true` o `false`).

Operador	Descripción	Ejemplo
`==`	**Igualdad**: Compara valores, realizando conversión de tipo si es necesario.	`"5" == 5` es `true`
`!=`	**Distinto:** Compara si los valores son diferentes.	`"5" != 6` es `true`
`===`	**Igualdad estricta**: Compara valores y tipo, sin conversión. **Recomendado**.	`"5" === 5` es `false`
`!==`	**Desigualdad estricta**: Compara si los valores o los tipos son diferentes.	`"5" !== 5` es `true`
`>` , `<` , `>=` , `<=`	Mayor que, menor que, mayor o igual, menor o igual.	`10 > 5` es `true`

- **Operadores Binarios (Bit a Bit)**: Estos operadores trabajan con los números a nivel de sus bits individuales (valores binarios de 0 y 1). Son utilizados generalmente para tareas de bajo nivel, manipulación de colores, o en criptografía y compresión, donde requiere la máxima eficiencia.

Operador	Descripción	Ejemplo
`&` (AND a Nivel de Bit)	Compara pares de bits. El bit de resultado es 1 si ambos bits son 1.	`5 & 1` es `1` (binario: `101 & 001 = 001`)
**`	` (OR a Nivel de Bit)**	Compara pares de bits. El bit de resultado es 1 si al menos uno de los bits es 1.
`~` (NOT a Nivel de Bit)	Invierte los bits del operando (complemento a uno).	`~5` es `-6` (inversión de bits y complemento a dos)
`<<` (Desplazamiento a Izquierda)	Desplaza los bits a la izquierda, agregando ceros al final. Equivale a multiplicar por potencias de 2.	`5 << 1` es `10`
`>>` (Desplazamiento a Derecha)	Desplaza los bits a la derecha, conservando el signo del número (rellena con el bit de signo).	`5 >> 1` es `2`
`>>>` (Desplazamiento a Derecha Sin Signo)	Desplaza los bits a la derecha, rellenando siempre con ceros (ignora el bit de signo).	`5 >>> 1` es `2`

Operadores Lógicos (Booleanos)

Combinan expresiones booleanas y también se usan para patrones de "cortocircuito".

Operador	Descripción	Ejemplo
`&&` (AND)	`true` solo si AMBOS son verdaderos	`true && false` → `false`
`||` (OR)	`true` si AL MENOS UNO es verdadero	`true || false` → `true`
`!` (NOT)	Invierte el valor booleano	`!true` → `false`

javascript

```javascript
// Uso básico en condiciones
const edad = 25;
const tieneLicencia = true;

if (edad >= 18 && tieneLicencia) {
  console.log("Puede conducir");
}

// Short-circuit evaluation (evaluación de cortocircuito)
// && devuelve el primer valor "falsy" o el último si todos son "truthy"
null && "hola"     // null (se detiene en el primer falsy)
"a" && "b" && "c"  // "c" (todos truthy, devuelve el último)

// || devuelve el primer valor "truthy" o el último si todos son "falsy"
"" || "default"    // "default" (primer truthy)
0 || null || "ok"  // "ok"

// Patrón común: valores por defecto
function saludar(nombre) {
  nombre = nombre || "Invitado";  // Si nombre es falsy, usa "Invitado"
  console.log(`Hola, ${nombre}`);
}
```

```
// Operador nullish coalescing (??) - ES2020
// Más preciso: solo usa el valor por defecto si es null o undefined
let valor = 0;
valor || 10    // 10 (0 es falsy)
valor ?? 10    // 0 (0 NO es null/undefined)
```

> **Tip**
>
> `??` **vs** `||`: Usa `??` cuando `0` o `""` sean valores válidos que no quieres reemplazar. Usa `||` cuando cualquier valor "falsy" deba ser reemplazado.

- **Operadores de Objetos**: Son símbolos que realizan operaciones como de acceso con objetos y sus propiedades.+
  - `.` **(Punto)**: Accede a una propiedad o método de un objeto (ej. `persona.nombre`).
  - `[]` **(Corchetes)**: Accede a elementos de un array o a propiedades de un objeto usando una clave dinámica (ej. `frutas[0]`, `personas["nombre"]`).
  - `delete`: Elimina una propiedad de un objeto (ej. `delete usuario.edad`).
  - `in`: Inspecciona si una propiedad o método existe en un objeto (ej. `'nombre' in persona`).
  - `instanceof`: Comprueba si un objeto es una instancia de una clase o constructor específico (ej. `color instanceof String`).
  - `typeof`: Devuelve una cadena que indica el tipo de dato de una variable (ej. `typeof forma === "string"`).
  - `,` **(Coma)**: Se usa para separar elementos en listas (arrays, objetos, argumentos). En una expresión evalúa todos sus operandos de izquierda a derecha y **retorna el valor del último operando**.
  - `:` **(Dos Puntos)**: Se usa para asignar valores a propiedades en **literales de objeto** (`{clave: valor}`) y en el operador condicional **ternario**.
  - `\` **(Escape)**: El **El Operador de Escape** se utiliza dentro de las cadenas de texto para indicar que el carácter que le sigue debe ser interpretado como un carácter literal, no como parte de la sintaxis del código (ej. `"\"texto entre comillas\""`).
  - `in`: Verifica si una propiedad especificada existe en un objeto. Ej: `"write" in document`.
  - `new`: Crea una instancia (un nuevo objeto) a partir de una función constructora o clase. Ej: `var hoy = new Date();`.
  - `this`: Una palabra clave que hace referencia al objeto en el contexto actual de ejecución. Su valor depende de cómo se invoca una función.
- **Operador Condicional (Ternario)**: Es una forma abreviada de una sentencia `if...else`. Su sintaxis es `condicion ? valor_si_true : valor_si_false`. Ejemplo

  javascript

```
let esActivo = true;
    esActivo ? alert("Usuario activo") : alert("Usuario inactivo");
```

1.2.3. Estructuras de Flujo de Control

Estas estructuras permiten que el código tome decisiones y ejecute acciones repetitivas.

- **Sentencias condicionales**: Permiten ejecutar diferentes bloques de código en función de si una condición es verdadera o falsa.
  - `if...else`: Ejecuta un bloque de codigo si una condición es verdadera y, opcionalmente, otro bloque si es falsa.

- `switch` : Evalúa una expresión y ejecuta el bloque de código (`case`) que coincida con su valor. Las cláusulas `break` son esenciales para evitar la ejecución continúe en los siguientes `case` . La cláusula `default` maneja casos no especificados.

- **Sentencias de Bucle**: Se usan para ejecutar código repetidamente.
  - `for` : Repite un bloque de código un número conocido de veces. Su sintaxis es `for(inicialización; condición; incremento)` .
  - `while` : Repite un bloque de código mientras una condición especificada sea verdadera. La condición se evalúa *antes* de cada iteración.
  - `do...while` : Similar a `while` , pero garantiza que el bloque de código se ejecute al menos una vez, ya que la condición se evalúa *después* de cada iteración.
- **Iteración sobre Colecciones**: Se usa para procesar cada elemento de una colección, como arreglos, cadenas de texto u objetos, mediante bucles.
  - `for...in` : Itera sobre **nombres de propiedades (claves)** de un objeto. No es recomendable para iterar sobre arrays, ya que puede incluir propiedades inesperadas además de los índices numéricos.

javascript

```
const objeto = { a: 1, b: 2, c: 3 };
for (const clave in objeto) {
  console.log(`${clave}: ${objeto[clave]}`);
}
```

  - `for...of` : Introducido en ES6, itera sobre los **valores** de objetos iterables como arrays, strings, etc. Es el método preferido y más moderno para recorrer los elementos de un array.

javascript

```
const numeros = [1, 2, 3, 4, 5];
for (const numero of numeros) {
  console.log(numero);
}
```

  - `Array.prototype.forEach()` : Es un **método** de los arrays que ejecuta una función callback una vez por cada elemento. Se utiliza ampliamente para iterar sobre arrays, aunque no permite usar break o continue para detener la iteración.

javascript

```
const nombres = ["Ana", "Beto", "Carlos"];
nombres.forEach(function(nombre, indice) {
  console.log(`Elemento ${indice}: ${nombre}`);
  });
```

- **Control de bucles**: Para salir de los bucles prematuramente o para omitir iteraciones específicas.
  - `break` : Sale inmediatamente de bucle actual.
  - `continue` : Salta la iteración actual del bucle y pasa a la siguiente.
  - `labeled` : Las etiquetas (`label:`) permiten identificar un bucle. Se pueden usar con break y continue para controlar bucles anidados con mayor precisión, permitiendo salir o continuar no solo el bucle interno, sino también uno externo específico.

javascript

```
let i, j;
```

```
// 1. Etiquetamos el bucle exterior con 'bucleExterior'
bucleExterior: for (i = 0; i < 3; i++) {
console.log(`Bucle Exterior: i = ${i}`);

for (j = 0; j < 3; j++) {
    console.log(`  Bucle Interior: j = ${j}`);

    // 2. Si la condición se cumple, usamos break con la etiqueta
    if (i === 1 && j === 1) {
    console.log('¡Condición Cumplida! Saliendo de bucleExterior');
    break bucleExterior; // ¡Sale del bucle etiquetado, no solo del interno!
    }
}
}

console.log('Ejecución finalizada.');
```

Salida:

```
Bucle Exterior: i = 0
Bucle Interior: j = 0
Bucle Interior: j = 1
Bucle Interior: j = 2
Bucle Exterior: i = 1
Bucle Interior: j = 0
Bucle Interior: j = 1
¡Condición Cumplida! Saliendo de bucleExterior
Ejecución finalizada.
```

javascript

```
let i, j;

// 1. Etiquetamos el bucle exterior con 'bucleExterior'
bucleExterior:
for (i = 0; i < 3; i++) {
  console.log(`Bucle Exterior: i = ${i}`);

  for (j = 0; j < 3; j++) {

    // 2. Si la condición se cumple...
    if (i === 1 && j === 1) {
      console.log('  ¡Condición Cumplida! Saltando a la SIGUIENTE iteración de bucleExterior');

      // 3. ...usamos 'continue' con la etiqueta.
      // Esto detiene el bucle interno y salta a i = 2
      continue bucleExterior;
    }

    console.log(`  Bucle Interior: j = ${j}`);
  }
}

console.log('Ejecución finalizada.');
```

Salida:

```
Bucle Exterior: i = 0
  Bucle Interior: j = 0
```

```
  Bucle Interior: j = 1
  Bucle Interior: j = 2
Bucle Exterior: i = 1
  Bucle Interior: j = 0
  ¡Condición Cumplida! Saltando a la SIGUIENTE iteración de bucleExterior
Bucle Exterior: i = 2
  Bucle Interior: j = 0
  Bucle Interior: j = 1
  Bucle Interior: j = 2
Ejecución finalizada.
```

Estos bloques de construcción fundamentales -variables, operadores y estructuras de control- se combinan para crear la lógica que impulsa las aplicaciones web, un tema que exploraremos desde una perspectiva práctica en la siguiente sección.

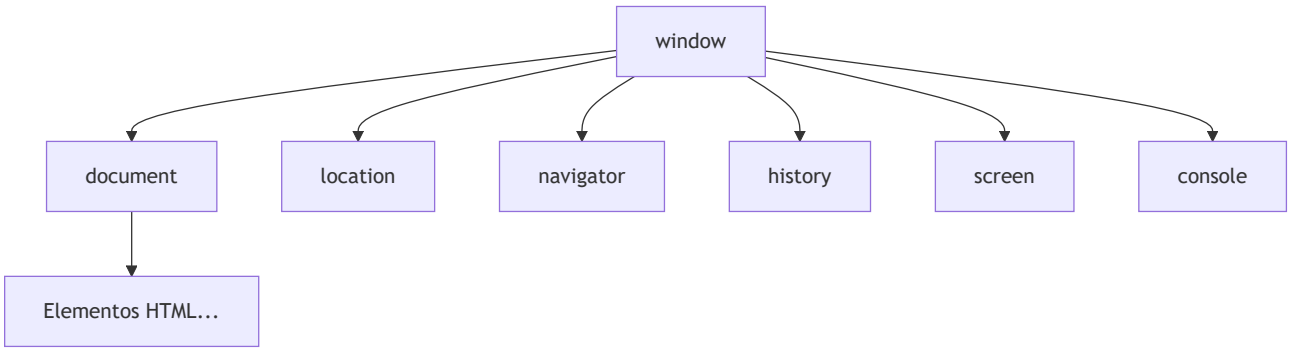
---

2. El Modelo de Objetos del Navegador (BOM)

El **BOM** (Browser Object Model) es la API que permite a JavaScript interactuar con el **navegador** en sí, no solo con el contenido de la página. Mientras que el **DOM** maneja el documento HTML, el **BOM** maneja todo lo demás: la ventana, la URL, el historial, la información del navegador, etc.

¿Qué puedes hacer con el BOM?

- Abrir/cerrar ventanas y pestañas
- Navegar a otras URLs
- Obtener información del navegador del usuario
- Acceder al historial de navegación
- Mostrar alertas, prompts y confirmaciones
- Ejecutar código después de un tiempo (`setTimeout` , `setInterval`)



> **Note**
>
> El BOM **no está estandarizado** oficialmente como el DOM, por lo que puede haber pequeñas diferencias entre navegadores. Sin embargo, la mayoría de propiedades y métodos funcionan igual en todos los navegadores modernos.

2.1. El Objeto Global: window

El objeto `window` es el **objeto de más alto nivel** en JavaScript del navegador. Representa la ventana (o pestaña) que contiene tu página web.

¿Por qué es importante?

- Es el **objeto global**: todas las variables y funciones globales son propiedades de `window`

- Contiene todos los demás objetos del BOM (`document` , `location` , `navigator` , etc.)
- Puedes omitir `window.` al llamar sus métodos: `alert()` es lo mismo que `window.alert()`

javascript

```javascript
// Estas dos formas son equivalentes
window.alert("Hola");
alert("Hola");

// Las variables globales son propiedades de window
var miVariable = "Hola";
console.log(window.miVariable);  // "Hola"

// Pero con let/const NO se añaden a window
let otraVariable = "Mundo";
console.log(window.otraVariable);  // undefined
```

Gestión de Ventanas y Temporizadores

javascript

```javascript
// Abrir una nueva ventana/pestaña
const nuevaVentana = window.open("https://google.com", "_blank");

// Cerrar la ventana (solo funciona si la abrimos nosotros)
nuevaVentana.close();

// setTimeout - ejecutar DESPUÉS de X milisegundos
setTimeout(() => {
  console.log("Han pasado 2 segundos");
}, 2000);

// setInterval - ejecutar CADA X milisegundos
const intervalo = setInterval(() => {
  console.log("Tick");
}, 1000);

// Detener el intervalo
clearInterval(intervalo);
```

Propiedades y Métodos Clave:

Propiedad	Descripción
`closed`	Devuelve un valor booleano que indica si una ventana ha sido cerrada.
`defaultStatus`	Ajusta o devuelve el valor por defecto de la barra de estado de una ventana.
`document`	Devuelve el objeto document de la ventana, que es el punto de entrada al DOM.
`frames`	Devuelve un array de todos los marcos (incluidos iframes) de la ventana actual.
`length`	Devuelve el número de frames (incluyendo iframes) que hay en dentro de una ventana.
`location`	Devuelve el objeto location, que contiene información sobre la URL actual.
`name`	Ajusta o devuelve el nombre de una ventana.
`navigator`	Devuelve el objeto navigator, con información sobre el navegador del cliente.

Propiedad	Descripción
`history`	Devuelve el objeto history, que permite interactuar con el historial de sesión del navegador.
`opener`	Devuelve una referencia a la ventana que abrió la ventana actual.
`parent`	Devuelve la ventana padre de la ventana actual (relevante en contextos de marcos).
`self`	Devuelve una referencia a la ventana actual; es un sinónimo de window.
`status`	Permite ajustar el texto en la barra de estado del navegador (en desuso en navegadores modernos).

Método	Descripción
`alert()`	Muestra un cuadro de diálogo de alerta con un mensaje y un botón de "Aceptar".
`blur()`	Elimina el foco de la ventana actual.
`confirm()`	Muestra un cuadro de diálogo con un mensaje y botones de "Aceptar" y "Cancelar". Devuelve true o false.
`focus()`	Coloca el foco en la ventana actual.
`prompt()`	Muestra un cuadro de diálogo que solicita una entrada al usuario.
`open()`	Abre una nueva ventana del navegador.
`close()`	Cierra la ventana actual.
`setInterval()`	Llama a una función o evalúa una expresión repetidamente a intervalos de tiempo fijos.
`clearInterval()`	Detiene la ejecución repetitiva iniciada con setInterval().
`setTimeout()`	Llama a una función o evalúa una expresión después de un número específico de milisegundos.

2.2. Objetos del Navegador (BOM)

Estos objetos, contenidos dentro de `window`, proporcionan acceso a información y funcionalidades específicas del entorno del navegador.

1. `location` - Información y control de la URL actual.

El objeto `location` te da acceso a todas las partes de la URL y métodos para navegar.

Ejemplo de URL y sus partes:

```
https://www.ejemplo.com:8080/productos/zapatos?color=rojo&talla=42#precio
└─────┘   └─────────────────┘└──────────────┘└──────────────────────┘└─────┘
protocol        host            pathname           search          hash
```

javascript

```javascript
// Suponiendo que estamos en:
// https://www.tienda.com:8080/productos?categoria=ropa#ofertas

location.href;      // "https://www.tienda.com:8080/productos?categoria=ropa#ofertas"
location.protocol;  // "https:"
```

```
location.host;      // "www.tienda.com:8080"
location.hostname;  // "www.tienda.com"
location.port;      // "8080"
location.pathname;  // "/productos"
location.search;    // "?categoria=ropa"
location.hash;      // "#ofertas"

// Navegar a otra página
location.href = "https://google.com";     // Navega (queda en historial)
location.assign("https://google.com");    // Igual que href
location.replace("https://google.com");   // Navega SIN añadir al historial

// Recargar la página
location.reload();  // Recarga desde caché si es posible
```

> **Tip**
>
> Usa `location.replace()` en lugar de `location.href` cuando hagas redirecciones y no quieras que el usuario pueda volver atrás con el botón de retroceso.

Propiedad	Descripción
`hash`	Fragmento después de `#` (ej. `#seccion1`)
`host`	Hostname + puerto (ej. `www.ejemplo.com:8080`)
`href`	URL completa. Modificarla navega a esa URL.
`hostname`	Solo el nombre de dominio
`pathname`	Ruta del recurso (ej. `/productos`)
`search`	Query string incluyendo `?` (ej. `?id=5&sort=asc`)
`port`	Cadena que contiene el número de puerto del servidor, dentro de la URL.
`protocol`	El protocolo utilizado (ej. "http:", "https:").

2. `Navigator`: El objeto `navigator` proporciona información sobre el navegador del cliente (el "agente de usuario").

Propiedad	Descripción
`appCodeName`	Cadena que contiene el nombre en código del navegador.
`appName`	El nombre oficial del navegador.
`appVersion`	La información de la versión del navegador.
`userAgent`	La cadena completa del agente de usuario enviada por el navegador en las peticiones HTTP.
`platform`	El sistema operativo en el que se ejecuta el navegador.
`cookieEnabled`	Un booleano que indica si las cookies están habilitadas.

Método	Descripción
`javaEnabled()`	Devuelve `true` si el navegador tiene Java habilitado.

> **Nota sobre Geolocalización**: A través de la propiedad `navigator.geolocation` se puede acceder a la ubicación del usuario.

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
        (position) => {
            console.log("Latitud: " + position.coords.latitude);
            console.log("Longitud: " + position.coords.longitude);
        },
        (error) => {
            console.error("Error obteniendo ubicación: ", error.message);
        }
    );
} else {
    console.log("Geolocalización no soportada por este navegador.");
}
```

3. **`document`**: El objeto `document` es la representación del DOM (Modelo de Objetos del Documento) y sirve como punto de entrada para interactuar con el contenido de la página.

   ¿Qué es el DOM? Imagina que tu página HTML es un **árbol genealógico**. El DOM es ese árbol: cada etiqueta HTML (`<div>` , `<p>` , `<img>`) es un "nodo" del árbol, y `document` es la raíz desde donde accedes a todos los demás.

```
document
   └── html
        ├── head
        │    └── title
        └── body
             ├── h1
             ├── p
             └── div
                  └── button
```

> **Note**
>
> El navegador **convierte automáticamente** tu código HTML en este árbol de objetos (DOM) cuando carga la página. `document` es tu punto de acceso a ese árbol.

Ejemplo práctico: Buscar y modificar elementos

javascript

```
// BUSCAR elementos (las 3 formas más comunes)
const porId = document.getElementById("miBoton");        // Por ID (el más rápido)
const porSelector = document.querySelector(".clase");    // Por selector CSS (el más flexible)
const todos = document.querySelectorAll("p");            // Todos los <p> de la página

// MODIFICAR contenido
porId.textContent = "Nuevo texto";           // Cambia el texto
porId.innerHTML = "<strong>Negrita</strong>"; // Cambia el HTML interno

// MODIFICAR estilos
porId.style.backgroundColor = "blue";
porId.style.padding = "10px";

// CREAR elementos nuevos
const nuevoDiv = document.createElement("div");
nuevoDiv.textContent = "¡Soy nuevo!";
document.body.appendChild(nuevoDiv);  // Lo añade al final del body
```

> **Tip**
>
> **Regla moderna**: Usa `querySelector()` para casi todo. Solo usa `getElementById()` si necesitas máximo rendimiento en búsquedas muy frecuentes.

Colecciones del documento (acceso rápido a grupos de elementos):

Colección	Descripción	Ejemplo de uso
`document.forms`	Todos los `<form>`	`document.forms[0]` o `document.forms["login"]`
`document.images`	Todas las `<img>`	`document.images.length`
`document.links`	Todos los `<a href="...">`	`document.links[0].href`
`document.scripts`	Todos los `<script>`	`document.scripts[0].src`
`document.head`	Elemento `<head>`	`document.head.appendChild(meta)`
`document.body`	Elemento `<body>`	`document.body.innerHTML`
`document.anchors`	Todos los `<a name="...">`	`document.anchors.length`
`document.styleSheets`	Hojas de estilo CSS	`document.styleSheets[0].cssRules`
`document.applets`	Todos los `<applet>`	`document.applets.length`

> **Note**
>
> Estas colecciones son "**vivas**": si añades un nuevo `<form>` al DOM, automáticamente aparecerá en `document.forms`.

Propiedades del documento:

Propiedad	Descripción	Ejemplo
`title`	Título de la pestaña (lectura/escritura)	`document.title = "Nueva pestaña"`
`URL`	URL completa (solo lectura)	`"https://ejemplo.com/pagina"`
`domain`	Dominio del servidor	`"ejemplo.com"`
`referrer`	URL de la página anterior	`"https://google.com"`
`cookie`	Cookies del documento	`"usuario=Ana; tema=oscuro"`
`readyState`	Estado de carga: `loading`, `interactive`, `complete`	`if (document.readyState === "complete")`
`lastModified`	Fecha de última modificación	`"12/11/2024 09:30:00"`

Métodos de búsqueda (los más importantes):

Método	Devuelve	Velocidad	Uso
`getElementById("id")`	1 elemento	Muy rápido	Buscar por ID único
`querySelector("css")`	1 elemento	Rápido	Selectores CSS flexibles

Método	Devuelve	Velocidad	Uso
`querySelectorAll("css")`	NodeList	Rápido	Múltiples elementos
`getElementsByClassName("clase")`	HTMLCollection	Rápido	Colección "viva"
`getElementsByTagName("tag")`	HTMLCollection	Rápido	Todos los `<tag>`

javascript

```javascript
// Ejemplos de búsqueda
document.getElementById("boton");              // <button id="boton">
document.querySelector(".card:first-child");   // Primera .card
document.querySelectorAll("ul > li");          // Todos los <li> hijos directos de <ul>
```

Métodos de creación y manipulación:

Método	Descripción	Ejemplo
`createElement("tag")`	Crea un nuevo elemento	`document.createElement("div")`
`createTextNode("texto")`	Crea un nodo de texto	`document.createTextNode("Hola")`
`write(html)`	Escribe HTML (Borra todo si se usa tras carga)	`document.write("<p>Hola</p>")`
`writeln(html)`	Lo mismo que `write()` pero con un salto de línea al final	`document.writeln("<p>Hola</p>")`
`innerHTML`	Modifica el contenido HTML de un elemento	`elemento.innerHTML = "Nuevo"`
`textContent`	Modifica el contenido de texto de un elemento	`elemento.textContent = "Nuevo"`
`appendChild()`	Añade un nuevo hijo al final del elemento	`elemento.appendChild(nuevo)`
`removeChild()`	Elimina un hijo del elemento	`elemento.removeChild(hijo)`
`replaceChild()`	Reemplaza un hijo del elemento	`elemento.replaceChild(nuevo)`
`insertBefore()`	Inserta un nuevo hijo antes de otro	`elemento.insertBefore(nuevo)`
`cloneNode()`	Clona un elemento	`elemento.cloneNode(true)`
`remove()`	Elimina un elemento	`elemento.remove()`

> **Caution**
>
> **Evita `document.write()`** en código moderno. Si lo usas después de que la página haya cargado, ¡borrará todo el contenido! Usa `innerHTML` o `appendChild()` en su lugar.

2.3. Gestión de Marcos (Frames) y Comunicación

Aunque su uso ha disminuido en favor de técnicas modernas, los marcos (`<frame>` e `<iframe>`) permiten incrustar documentos HTML independientes dentro de una página principal.

1. **Jerarquía**: Un `<frameset>` establece una relación de padre-hijo. La ventana principal que contiene el `<frameset>` es la ventana padre, y cada `<frame>` dentro de él es una ventana hijo. Es importante entender

que cada marco es un objeto window independiente, con su propio objeto document. Esta relación jerárquica, donde la ventana padre que contiene el `<frameset>` alberga objetos window hijos para cada `<frame>`.

2. **Comunicación entre Marcos**: JavaScript permite la comunicación entre estos contextos de ventana anidados.
    - **Padre a Hijo**: Desde la ventana padre, se puede acceder a un marco hijo a través de la colección frames, ya sea por su índice numérico o por su nombre.
    - **Hijo a Padre**: Desde un documento dentro de un marco, se puede acceder a la ventana padre usando la propiedad parent. La propiedad top siempre hace referencia a la ventana de más alto nivel en la jerarquía, útil para "escapar" de múltiples niveles de anidamiento.
    - **Hijo a Hijo (Hermanos)**: La comunicación directa entre marcos hermanos no es posible. Se debe realizar a través de su ancestro común, el padre.
3. **Comunicación entre Ventanas**: Para las ventanas emergentes creadas con `window.open()`, la comunicación se establece a través de la propiedad `opener`. Desde la ventana hija, opener proporciona una referencia a la ventana que la creó, permitiendo la comunicación bidireccional.

2.4. Eventos Comunes en Formularios

Para interactuar con el usuario, se utilizan manejadores de eventos:

- `onclick`: Al pulsar un botón o elemento.
- `onchange`: Cuando cambia el valor de un input (ej. color picker, select).
- `onsubmit`: Al enviar un formulario.

> javascript

```javascript
// Ejemplo de uso de eventos
const boton = document.getElementById('miBoton');
const input = document.getElementById('miInput');
const formulario = document.getElementById('miFormulario');

// onclick
boton.onclick = function() {
    alert("¡Has hecho clic!");
};

// onchange
input.onchange = function() {
    console.log("El valor ha cambiado a: " + this.value);
};

// onsubmit
formulario.onsubmit = function(event) {
    // Evitar el envío real del formulario para validarlo con JS
    event.preventDefault();
    console.log("Formulario enviado");
};
```

Además de los objetos que representan el entorno del navegador, JavaScript proporciona un conjunto de potentes objetos nativos para manipular tipos de datos fundamentales y realizar tareas comunes.

---

3. Objetos Nativos de JavaScript

Los objetos nativos son objetos incorporados en el lenguaje JavaScript, disponibles globalmente en cualquier entorno de ejecución. Estos objetos, como `String`, `Math` y `Date`, no dependen del navegador ni de librerías externas y proporcionan un conjunto de propiedades y métodos esenciales para realizar tareas comunes, desde la manipulación de texto y cálculos matemáticos hasta la gestión de fechas y horas.

3.1. Objeto `String`

El objeto `String` se utiliza para trabajar con cadenas de texto. Aunque las cadenas literales son primitivas, JavaScript las trata como objetos `String` al invocar sus métodos, proporcionando una rica API para su manipulación.

1. **Creación y Caracteres de Escape**: Una cadena se crea envolviendo texto entre comillas simples o dobles. Para incluir caracteres especiales dentro de una cadena, se utilizan secuencias de escape.

Símbolo	Explicación
`\"`	Comillas dobles
`\'`	Comilla simple
`\\`	Barra inclinada
`\n`	Nueva línea
`\t`	Tabulador
`\r`	Retorno de carro

   Además de las comillas simples y dobles, JavaScript moderno (a partir de ES6) introduce los `Template Literals` (o plantillas de texto), que utilizan comillas invertidas (``) (también llamadas *backticks*). Esta sintaxis ofrece dos grandes ventajas sobre las comillas tradicionales:

   - **Interpolación de variables**: Permiten incrustar variables y expresiones dentro de la cadena de forma legible, usando la sintaxis `${...}`. Esto evita la necesidad de concatenar cadenas con el operador `+`.

     javascript

     ```javascript
     let nombre = "Ana";
     let edad = 25;

     // Forma antigua (con concatenación)
     let saludo_antiguo = "Hola, mi nombre es " + nombre + " y tengo " + edad + " años.";

     // Forma moderna (con template literals)
     let saludo_moderno = `Hola, mi nombre es ${nombre} y tengo ${edad} años.`;
     ```

   - **Cadenas multilínea**: Permiten que una cadena de texto ocupe varias líneas en el código sin necesidad de usar el carácter de escape `\n`.

     javascript

     ```javascript
     // Esto es muy útil para generar plantillas HTML
     let html = `

       <h1>Título</h1>
       <p>Esto es un párrafo.</p>
     `;
     ```

2. **Propiedades y Métodos**:

Método/Propiedad	Descripción
`length`	(Propiedad) Devuelve el número de caracteres en la cadena.
`charAt(index)`	Devuelve el carácter en la posición especificada.
`charCodeAt()`	Devuelve el Unicode del carácter especificado por la posición que se indica entre paréntesis.
`concat()`	Une una o más cadenas y devuelve el resultado de esa unión.
`fromCharCode()`	Convierte valores Unicode a caracteres.
`indexOf(substring)`	Devuelve la posición de la primera ocurrencia de una subcadena.
`lastIndexOf(substring)`	Devuelve la posición de la última ocurrencia de una subcadena.
`match()`	Busca una coincidencia entre una expresión regular y una cadena y devuelve las coincidencias o `null` si no ha encontrado nada.
`replace(search, replacement)`	Busca una subcadena y la reemplaza por otra.
`search()`	Busca una subcadena en la cadena y devuelve la posición dónde se encontró.
`slice(start, end)`	Extrae una sección de la cadena y devuelve una nueva cadena.
`split(separator)`	Divide una cadena en un array de subcadenas basándose en un separador. Es un método de String: "Corta" una cadena y te da un `Array`.
`join(separator)`	Une todos los elementos de un array en una sola cadena. Es un método de Array: "Une" un array y te da una `String`.
`substr(start, length)`	Extrae un número de caracteres de una cadena, desde una posición inicial.
`substring(start, end)`	Extrae los caracteres entre dos índices especificados.
`toLowerCase()`	Convierte toda la cadena a minúsculas.
`toUpperCase()`	Convierte toda la cadena a mayúsculas.

3.2. Objeto `Math`

El objeto Math es un objeto estático que proporciona propiedades y métodos para realizar operaciones matemáticas. No es un constructor; sus miembros se invocan directamente sobre el objeto `Math`.

- **Propiedades Constantes**:

Propiedad	Descripción
`PI`	El número Pi (aprox. 3.14159).
`LN2`	El logaritmo neperiano de 2 (aproximadamente 0.693).
`LN10`	El logaritmo neperiano de 10 (aproximadamente 2.302).
`LOG2E`	El logaritmo base 2 de E (aproximadamente 1.442).
`LOG10E`	El logaritmo base 10 de E (aproximadamente 0.434).

Propiedad	Descripción
`SQRT2`	La raíz cuadrada de 2 (aprox. 1.414).

- Métodos Comunes:

Método	Descripción
`abs(x)`	Devuelve el valor absoluto de x.
`ceil(x)`	Redondea x hacia arriba al entero más cercano.
`floor(x)`	Redondea x hacia abajo al entero más cercano.
`round(x)`	Redondea x al entero más próximo.
`log(x)`	Devuelve el logaritmo neperiando (base E) de x.
`max(n1, n2, ...)`	Devuelve el mayor de los números pasados como argumentos.
`min(n1, n2, ...)`	Devuelve el menor de los números pasados como argumentos.
`pow(x, y)`	Devuelve x elevado a la potencia y.
`random()`	Devuelve un número pseudoaleatorio entre 0 (inclusive) y 1 (exclusive).
`sqrt(x)`	Devuelve la raíz cuadrada de x.
`cos(x), sin(x)`	Devuelven el coseno y el seno de x (en radianes).
`tan(x)`	Devuelve la tangente de un ángulo.
`acos(x), asin(x)`	Devuelven el arcocoseno y el arcoseno de x (en radianes).
`atan(x)`	Devuelve el arcotangente de x, en radianes con un valor entre -PI/2 y PI/2.
`atan2(y,x)`	Devuelve el arcotangente del cociente de sus argumentos.

3.3. Objeto `Date`

El objeto `Date` se utiliza para trabajar con fechas y horas. Cada instancia representa un **momento específico en el tiempo**, almacenado internamente como el número de milisegundos transcurridos desde el **1 de enero de 1970 00:00:00 UTC** (conocido como "época Unix" o "timestamp").

> **Note**
>
> JavaScript NO tiene un tipo de dato separado para fechas. Todo se maneja con el objeto `Date`, que incluye tanto fecha como hora.

Formas de crear un objeto Date:

> javascript

```javascript
// 1. Fecha y hora ACTUAL
const ahora = new Date();
console.log(ahora);  // Ej: "Wed Dec 11 2024 08:40:00 GMT+0100"

// 2. Desde un TIMESTAMP (milisegundos desde 1970)
const desdeMilisegundos = new Date(0);
console.log(desdeMilisegundos);  // "Thu Jan 01 1970 01:00:00 GMT+0100"
```

```
// 3. Desde una CADENA de texto
const desdeCadena = new Date("2024-12-25");
const navidad = new Date("December 25, 2024 10:00:00");

// 4. Con COMPONENTES individuales (¡OJO! los meses van de 0-11)
const finDeAno = new Date(2024, 11, 31, 23, 59, 59);
//                        año  mes día hora min seg
//                             ↑ 11 = Diciembre (0 = Enero)
```

> **Caution**
>
> **¡Los meses van de 0 a 11!** Enero = 0, Febrero = 1, ..., Diciembre = 11. Este es uno de los errores más comunes al trabajar con fechas.

Métodos get y set:

El objeto Date proporciona métodos para obtener (`get`) y establecer (`set`) sus componentes:

Métodos `get`	Métodos `set`	Descripción
`getDate()`	`setDate()`	Obtiene/Establece el día del mes (1-31).
`getDay()`	-	Obtiene el día de la semana (0=Domingo, 6=Sábado).
`getFullYear()`	`setFullYear()`	Obtiene/Establece el año con 4 dígitos.
`getMonth()`	`setMonth()`	Obtiene/Establece el mes (0-11).
`getHours()`	`setHours()`	Obtiene/Establece la hora (0-23).
`getMinutes()`	`setMinutes()`	Obtiene/Establece los minutos (0-59).
`getSeconds()`	`setSeconds()`	Obtiene/Establece los segundos (0-59).
`getMilliseconds()`	`.setMilliseconds()`	Obtiene/Establece los milisegundo.
`getTime()`	`setTime()`	Obtiene/Establece el tiempo en milisegundos desde la época Unix.

javascript

```
const fecha = new Date("2024-12-25T10:30:00");

// Obtener componentes
fecha.getFullYear();  // 2024
fecha.getMonth();     // 11 (diciembre)
fecha.getDate();      // 25
fecha.getDay();       // 3 (miércoles, 0=domingo)
fecha.getHours();     // 10
fecha.getMinutes();   // 30

// Modificar componentes
fecha.setFullYear(2025);  // Cambiar a 2025
fecha.setMonth(0);        // Cambiar a enero (0)
fecha.setDate(1);         // Cambiar al día 1

// Operaciones con fechas
const hoy = new Date();
const manana = new Date(hoy);
manana.setDate(hoy.getDate() + 1);  // Sumar 1 día

// Calcular diferencia entre fechas (en días)
```

```
const fecha1 = new Date("2024-01-01");
const fecha2 = new Date("2024-12-31");
const diferenciaMiliseg = fecha2 - fecha1;
const diferenciaDias = diferenciaMiliseg / (1000 * 60 * 60 * 24);
console.log(diferenciaDias);  // 365
```

> **Tip**
>
> Para formatear fechas de forma legible, usa `toLocaleDateString()`:
>
> ```
> const fecha = new Date();
> fecha.toLocaleDateString("es-ES");  // "11/12/2024"
> fecha.toLocaleDateString("es-ES", {
>   weekday: "long",
>   day: "numeric",
>   month: "long",
>   year: "numeric"
> });  // "miércoles, 11 de diciembre de 2024"
> ```

3.3.1. Librerías Externas para Fechas

Trabajar con fechas usando solo el objeto `Date` nativo puede ser tedioso y propenso a errores. Por eso existen librerías que simplifican las operaciones comunes: manipulación, formateo, comparación y manejo de zonas horarias.

Opciones populares:

Librería	Tamaño	Características
Date.js	~7KB	Sintaxis en lenguaje natural, API fluida
Day.js	~2KB	API similar a Moment.js, muy ligera, **recomendada**
Luxon	~70KB	Soporte completo de zonas horarias, internacionalización
~~Moment.js~~	~70KB	*Obsoleta* - los creadores recomiendan Day.js o Luxon

> **Tip**
>
> Para proyectos nuevos, se recomienda **Day.js** por su tamaño mínimo y API intuitiva. Moment.js ya no se recomienda debido a su gran tamaño y el hecho de que sus creadores lo han declarado en modo "legacy".

Date.js - Ejemplo de uso:

Puedes incluir la librería descargando el archivo localmente o mediante un CDN:

```
<!-- CDN -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/datejs/1.0/date.min.js"></script>
```

Métodos principales de Date.js:

Categoría	Métodos
Creación	`Date.today()`, `Date.parse("next friday")`, `Date.parseExact()`
Manipulación	`addDays()`, `addWeeks()`, `addMonths()`, `addYears()`, `addHours()`, `clearTime()`
Navegación	`next()`, `last()`, `moveToFirstDayOfMonth()`, `moveToLastDayOfMonth()`, `moveToDayOfWeek()`

Categoría	Métodos
Comparación	`compare()`, `equals()`, `isAfter()`, `isBefore()`, `between()`, `isToday()`
Fluent API	`.next().friday()`, `.last().monday()`, `.today().at("9:00")`

javascript

```javascript
// Date.js - Ejemplos con lenguaje natural
const hoy = Date.today();                    // Hoy a las 00:00:00
const proximoViernes = Date.parse("next friday");
const haceDiezDias = Date.today().addDays(-10);

// Operaciones encadenadas
const fechaFutura = Date.today()
  .addMonths(1)
  .addDays(5)
  .at("10:30");

// Comparaciones
Date.today().isAfter(new Date("2020-01-01")); // true

// Formateo
fechaFutura.toString("dd/MM/yyyy HH:mm"); // "16/01/2025 10:30"
```

Alternativa moderna: Day.js

javascript

```javascript
// Day.js - Más ligera y moderna
import dayjs from 'dayjs';

dayjs();                          // Ahora
dayjs().add(7, 'day');            // Dentro de 7 días
dayjs().subtract(1, 'month');     // Hace 1 mes
dayjs().format('DD/MM/YYYY');     // "11/12/2024"
dayjs('2024-12-25').isAfter(dayjs()); // ¿Es después de hoy?
```

3.4. Objetos `Number` y `Boolean`

Aunque menos comunes en el uso diario, `Number` y `Boolean` son **objetos envoltorio** (wrappers) para los tipos de datos primitivos correspondientes. JavaScript automáticamente "envuelve" los primitivos cuando accedes a sus métodos.

> **Note**
>
> Cuando escribes `(42).toString()`, JavaScript temporalmente convierte el número primitivo `42` en un objeto `Number` para poder llamar al método, y luego descarta el objeto. Este proceso se llama **autoboxing**.

1. `Number`: Proporciona propiedades para conocer los límites numéricos del lenguaje y métodos para formatear números.

   Constantes importantes:

Constante	Valor	Descripción
`MAX_VALUE`	$\sim 2^{1024}$	Valor más grande

Constante	Valor	Descripción
`MIN_VALUE`	~ 5×10^{-324}	Valor más pequeño
`MAX_SAFE_INTEGER`	$2^{53}-1$	Valor seguro más grande
`MIN_SAFE_INTEGER`	$-(2^{53}-1)$	Valor seguro más pequeño
`POSITIVE_INFINITY`	Infinity	Infinito positivo: +∞
`NEGATIVE_INFINITY`	-Infinity	Infinito negativo: -∞
`EPSILON`	2^{-52}	Número muy pequeño: ε

Métodos de formateo:

Método	Descripción
`.toFixed(digits)`	Redondea al número de decimales especificado
`.toPrecision(size)`	Formatea con un número total de dígitos significativos
`.toExponential(digits)`	Convierte a notación científica (exponencial)
`.toString(base)`	Convierte a string en la base indicada (2=binario, 16=hexadecimal)
`isNaN(number)`	Comprueba si number no es un número.
`.parseInt(text)`	Convierte un `string` text en un `number` entero.
`.parseFloat(text)`	Convierte un `string` text en un `number` decimal.

> javascript

```javascript
// toFixed() - Ideal para precios y dinero
const precio = 19.99;
precio.toFixed(2);    // "19.99"
precio.toFixed(0);    // "20" (redondea)
precio.toFixed(4);    // "19.9900"

// toPrecision() - Controla el total de dígitos
const num = 123.456;
num.toPrecision(2);   // "1.2e+2" (notación científica si es necesario)
num.toPrecision(4);   // "123.5"
num.toPrecision(6);   // "123.456"

// toExponential() - Notación científica
const grande = 1234567;
grande.toExponential(2);  // "1.23e+6"

// toString(base) - Conversión de base numérica
const decimal = 255;
decimal.toString(2);   // "11111111" (binario)
decimal.toString(8);   // "377" (octal)
decimal.toString(16);  // "ff" (hexadecimal)

// Verificación de números
Number.isNaN(NaN);          // true
Number.isNaN("hola");       // false (más estricto que isNaN global)
Number.isFinite(100);       // true
Number.isFinite(Infinity);  // false
```

```
Number.isInteger(42);        // true
Number.isInteger(42.5);      // false
```

> **Tip**
>
> Usa `toFixed(2)` para mostrar precios con dos decimales. Recuerda que devuelve un **string**, así que si necesitas operar con él, conviértelo de nuevo a número: `Number(precio.toFixed(2))`.

2. **`Boolean`**: El objeto Boolean representa un valor lógico: `true` (verdadero) o `false` (falso). Su principal utilidad es convertir cualquier valor a su equivalente booleano.

Valores "Falsy" vs "Truthy"

En JavaScript, cada valor tiene una "verdad inherente" cuando se evalúa en un contexto booleano (como en un `if`). Los valores **falsy** se convierten a `false`, el resto son **truthy**.

Falsy (se convierten a `false`)	Truthy (se convierten a `true`)
`false`	`true`
`0` y `-0`	Cualquier número que no sea 0
`""` (cadena vacía)	Cualquier cadena no vacía (incluso `"0"` y `" "`)
`null`	Objetos `{}` y arrays `[]` (aunque estén vacíos)
`undefined`	Funciones
`NaN`	Todo lo demás

javascript

```
// Ejemplos de conversión a booleano
Boolean(0);         // false
Boolean("");        // false
Boolean(null);      // false
Boolean(undefined); // false
Boolean(NaN);       // false

Boolean(1);         // true
Boolean("hola");    // true
Boolean([]);        // true (¡los arrays vacíos son truthy!)
Boolean({});        // true (¡los objetos vacíos son truthy!)
Boolean("0");       // true (¡"0" es un string no vacío!)

// Uso práctico en condiciones
let nombre = "";
if (nombre) {
  console.log("Tiene nombre");
} else {
  console.log("No tiene nombre"); // ← Se ejecuta esto
}

// Doble negación (!!) como atajo para Boolean()
!!0         // false
!!"texto"   // true
```

> **Tip**

> El operador `!!` (doble negación) es una forma rápida de convertir cualquier valor a su equivalente booleano: `!!"texto"` es equivalente a `Boolean("texto")`.

La combinación de los fundamentos del lenguaje, el control sobre el entorno del navegador a través del BOM y las utilidades proporcionadas por los objetos nativos, otorgan a JavaScript la capacidad de construir aplicaciones web ricas y complejas.

---

4. Estructuras de Datos: Arrays y Colecciones

Elegir la estructura de datos correcta es fundamental para organizar y gestionar la información de manera eficiente. JavaScript ofrece varias estructuras de datos nativas, siendo el **Array** la más versátil y fundamental, complementada por colecciones especializadas como **Set** y **Map** introducidas en ES6.

4.1. El Objeto Array: La Colección Fundamental

Un **Array** es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa. En algunas ocasiones también se les denomina "arreglos" o "vectores". Es la estructura más utilizada en JavaScript para gestionar listas, colecciones y secuencias de información.

Sus características principales son:

- **Indexación base-cero**: JavaScript utiliza **exclusivamente** indexación base-cero, donde el primer elemento está en el índice `0`.
- **Naturaleza dinámica**: La propiedad `length` se ajusta automáticamente al añadir o eliminar elementos.
- **Tipado flexible**: Al contrario que muchos otros lenguajes, JavaScript permite arrays de tipo mixto, no siendo obligatorio que todos los elementos sean del mismo tipo.

4.1.1. Creación e Inicialización

JavaScript ofrece múltiples formas de crear arrays:

> javascript

```javascript
// 1. Constructor vacío
const miArray = new Array();

// 2. Constructor con tamaño predefinido (40 posiciones undefined)
const arrayGrande = new Array(40);

// 3. Constructor con elementos iniciales
const sistemaSolar = new Array("Mercurio", "Venus", "Tierra");

// 4. Sintaxis literal (RECOMENDADA - más concisa y legible)
const planetas = ["Mercurio", "Venus", "Tierra", "Marte"];

// 5. Asignación directa por índice
const numeros = [];
numeros[0] = 10;
numeros[1] = 20;
numeros[5] = 60; // El array tendrá posiciones vacías entre 1 y 5

// 6. Array mixto (String, Number, Boolean)
const mixto = ["a", 5, true, null, { nombre: "Ana" }];
```

> Caution

Al crear un array con `new Array(size)` donde `size` es un único número, JavaScript crea un array vacío de `size` posiciones (no inicializadas). No es lo mismo que `[3]` (array con un elemento `3`). Además, `new Array(3)` no es exactamente igual a `[undefined, undefined, undefined]`, ya que realmente crea posiciones vacías no rellenadas.

> **Note**
>
> Los objetos literales (`{}`) no son arrays. Se definen con llaves y carecen de propiedades y métodos del prototipo de Array (como `.length`, `.push()`, etc.).

4.1.2. Recorrido de Arrays

Existen varias formas de iterar sobre los elementos de un array:

javascript

```javascript
const sistemaSolar = ["Mercurio", "Venus", "Tierra", "Marte"];

// 1. Bucle for clásico (control total sobre el índice)
for (let i = 0; i < sistemaSolar.length; i++) {
  console.log(`Índice ${i}: ${sistemaSolar[i]}`);
}

// 2. Bucle while
let j = 0;
while (j < sistemaSolar.length) {
  console.log(sistemaSolar[j]);
  j++;
}

// 3. for...of (ES6 - RECOMENDADO para iterar valores)
for (const planeta of sistemaSolar) {
  console.log(planeta);
}

// 4. forEach (método funcional - sin break/continue)
sistemaSolar.forEach((planeta, indice) => {
  console.log(`${indice}: ${planeta}`);
});
```

> **Warning**
>
> No uses `for...in` para iterar arrays. `for...in` itera sobre las claves (incluyendo propiedades heredadas) y puede producir resultados inesperados. `for...of` se centra exclusivamente en los valores.

4.1.3. Añadir y Eliminar Elementos

JavaScript ofrece varios métodos para añadir o eliminar elementos de un array. Los métodos actúan en diferentes posiciones:

Método	Posición	Acción	Devuelve
`push()`	Final	Añade elemento/s	Nueva longitud
`pop()`	Final	Elimina elemento	El elemento eliminado
`unshift()`	Inicio	Añade elemento/s	Nueva longitud
`shift()`	Inicio	Elimina elemento	El elemento eliminado

Método	Posición	Acción	Devuelve
`splice()`	Cualquier	Añade/elimina	Array con elementos eliminados

javascript

```javascript
const elementos = ["a", "b", "c"];

// ===== AÑADIR ELEMENTOS =====
elementos.push("d");      // Añade al final → ["a", "b", "c", "d"] (devuelve 4)
elementos.unshift("Z");   // Añade al inicio → ["Z", "a", "b", "c", "d"] (devuelve 5)

// ===== ELIMINAR ELEMENTOS =====
elementos.pop();          // Elimina del final → ["Z", "a", "b", "c"] (devuelve "d")
elementos.shift();        // Elimina del inicio → ["a", "b", "c"] (devuelve "Z")
```

El método `splice()`: La navaja suiza

`splice(inicio, cantidad, ...elementosNuevos)` es el más versátil porque puede **eliminar, reemplazar o insertar** elementos en cualquier posición.

javascript

```javascript
const frutas = ["Manzana", "Banana", "Cereza", "Dátil"];

// ELIMINAR: desde índice 1, eliminar 2 elementos
const eliminados = frutas.splice(1, 2);
console.log(frutas);     // ["Manzana", "Dátil"]
console.log(eliminados); // ["Banana", "Cereza"] (lo que se eliminó)

// INSERTAR: desde índice 1, eliminar 0, insertar nuevos
frutas.splice(1, 0, "Limon", "Kiwi");
console.log(frutas);     // ["Manzana", "Limon", "Kiwi", "Dátil"]

// REEMPLAZAR: desde índice 2, eliminar 1, insertar nuevo
frutas.splice(2, 1, "Fresa");
console.log(frutas);     // ["Manzana", "Limon", "Fresa", "Dátil"]
```

> **Caution**
>
> **¡Nunca uses `delete` para eliminar elementos!** `delete` deja un "hueco" (`undefined`) en el array y **no actualiza la propiedad `length`**. Esto puede causar bugs difíciles de detectar.

javascript

```javascript
// EVITAR: delete deja "huecos"
const colores = ["rojo", "verde", "azul"];
delete colores[1];
console.log(colores);        // ["rojo", empty, "azul"] o ["rojo", undefined, "azul"]
console.log(colores.length); // 3 (¡no cambió!)
console.log(1 in colores);   // false (el índice ya no existe)

// CORRECTO: splice() elimina limpiamente
const animales = ["perro", "gato", "pájaro"];
animales.splice(1, 1);
console.log(animales);       // ["perro", "pájaro"]
console.log(animales.length); // 2 (actualizado correctamente)
```

4.1.4. Propiedades y Métodos Principales

Propiedad	Descripción
`length`	Número de elementos en el array. Es ajustable.
`constructor`	Función que creó el prototipo (`function Array()`).
`prototype`	Permite añadir propiedades y métodos al prototipo Array.

Método	Descripción	Mutabilidad
`at(index)`	Accede a un elemento. Acepta índices negativos (-1 = último).	No modifica
`with(index, val)`	Devuelve copia con elemento reemplazado (ES2023).	Nuevo array
`push()`	Añade al final y devuelve nueva longitud.	Modifica
`pop()`	Extrae y devuelve el último elemento.	Modifica
`unshift()`	Añade al inicio y devuelve nueva longitud.	Modifica
`shift()`	Extrae y devuelve el primer elemento.	Modifica
`concat()`	Une arrays y devuelve un nuevo array.	Nuevo array
`slice(start, end)`	Extrae una sección y devuelve un nuevo array.	Nuevo array
`splice(start, count, ...items)`	Elimina, reemplaza o agrega elementos.	Modifica
`reverse()`	Invierte el orden de los elementos.	Modifica
`toReversed()`	Versión inmutable de reverse() (ES2023).	Nuevo array
`sort()`	Ordena los elementos (por defecto, lexicográficamente).	Modifica
`toSorted()`	Versión inmutable de sort() (ES2023).	Nuevo array
`join(separator)`	Une elementos en una cadena con un separador.	No modifica
`indexOf(value)`	Devuelve el primer índice del valor, o -1.	No modifica
`includes(value)`	Devuelve `true` si el valor está en el array.	No modifica

> javascript

```javascript
// Ejemplos de métodos comunes
const numeros = [3, 1, 4, 1, 5, 9];

// reverse() - modifica el array original
const invertido = numeros.reverse();
console.log(numeros); // [9, 5, 1, 4, 1, 3] - ¡El original cambió!

// slice() - NO modifica el original
const original = [1, 2, 3, 4, 5];
const porcion = original.slice(1, 4);
console.log(porcion); // [2, 3, 4]
console.log(original); // [1, 2, 3, 4, 5] - Sin cambios

// sort() - ¡Cuidado con números!
const nums = [1, 10, 2, 21];
nums.sort(); // Ordenación lexicográfica
```

```javascript
console.log(nums); // [1, 10, 2, 21] - ¡Incorrecto para números!

// Solución: función de comparación
nums.sort((a, b) => a - b);
console.log(nums); // [1, 2, 10, 21] - Correcto
```

Métodos modernos de acceso (ES2022+)

> javascript

```javascript
const letras = ["a", "b", "c", "d", "e"];

// at() - Acepta índices negativos para acceder desde el final
letras.at(0);    // "a" (igual que letras[0])
letras.at(-1);   // "e" (último elemento)
letras.at(-2);   // "d" (penúltimo)

// Antes de at(), acceder al último era más verboso:
letras[letras.length - 1];  // "e" (forma antigua)

// with() - Devuelve un NUEVO array con el elemento modificado (inmutable)
const nuevasLetras = letras.with(1, "Z");
console.log(nuevasLetras);  // ["a", "Z", "c", "d", "e"]
console.log(letras);        // ["a", "b", "c", "d", "e"] - Sin cambios
```

> **Tip**
>
> El método `at()` es especialmente útil para acceder al último elemento (`at(-1)`) sin calcular `length - 1` .
> El método `with()` es ideal para actualizaciones inmutables en programación funcional.

4.2. Arrays Multidimensionales

JavaScript simula arrays multidimensionales creando **arrays que contienen otros arrays**. Esto permite crear estructuras bidimensionales (tablas), tridimensionales (cubos), etc.

> javascript

```javascript
// Array bidimensional (tabla de datos)
const datos = [
  ["Cristina", "Seguridad", 24],
  ["Catalina", "Bases de Datos", 17],
  ["Vieites", "Sistemas Informáticos", 28],
  ["Benjamin", "Redes", 26]
];

// Acceso a elementos: datos[fila][columna]
console.log(datos[1][0]); // "Catalina" (fila 1, columna 0)
console.log(datos[2][1]); // "Sistemas Informáticos"
console.log(datos[3][2]); // 26

// Recorrido con bucle anidado
document.write("<table border=1>");
for (let i = 0; i < datos.length; i++) {
  document.write("<tr>");
  for (let j = 0; j < datos[i].length; j++) {
    document.write("<td>" + datos[i][j] + "</td>");
  }
  document.write("</tr>");
```

```
}
document.write("</table>");
```

Alternativa: Arrays Paralelos

Otra técnica es usar múltiples arrays donde el mismo índice conecta datos relacionados:

> javascript

```javascript
const profesores = ["Cristina", "Catalina", "Vieites", "Benjamin"];
const asignaturas = ["Seguridad", "Bases de Datos", "Sistemas", "Redes"];
const alumnos = [24, 17, 28, 26];

// El índice 1 conecta: "Catalina" + "Bases de Datos" + 17
function imprimeDatos(indice) {
  console.log(`${profesores[indice]} imparte ${asignaturas[indice]} con ${alumnos[indice]} alumnos.`);
}

for (let i = 0; i < profesores.length; i++) {
  imprimeDatos(i);
}
```

4.3. El Objeto Set: Colecciones de Valores Únicos

Un objeto **Set** almacena una colección de valores donde **cada valor solo puede ocurrir una vez**. Es ideal para eliminar duplicados o verificar membresía de forma eficiente, es decir, para comprobar rápidamente si un elemento ya existe en la colección.

Propiedad/Método	Descripción
`size`	Número de elementos en el Set.
`add(valor)`	Añade un elemento. Devuelve el Set (encadenable).
`has(valor)`	Devuelve `true` si el valor existe.
`delete(valor)`	Elimina un valor. Devuelve `true` si existía.
`clear()`	Elimina todos los elementos.
`forEach()`, `values()`, `keys()`	Métodos de iteración (en orden de inserción).

> javascript

```javascript
// Crear y manipular un Set
const mySet = new Set();

mySet.add(1);          // Set { 1 }
mySet.add(5);          // Set { 1, 5 }
mySet.add(5);          // Set { 1, 5 } - ¡Los duplicados se ignoran!
mySet.add("texto");    // Set { 1, 5, 'texto' }

console.log(mySet.has(1));   // true
console.log(mySet.has(3));   // false
console.log(mySet.size);     // 3

mySet.delete(5);
console.log(mySet.has(5));   // false

// Iteración
```

```javascript
for (const valor of mySet) {
  console.log(valor);
}
```

Caso de uso crítico: Eliminar duplicados de un array

javascript

```javascript
const numbers = [2, 3, 4, 4, 2, 3, 3, 4, 5, 5, 6, 7, 32, 3, 4, 5];

// Forma concisa usando Set y operador de propagación
const uniqueNumbers = [...new Set(numbers)];

console.log(uniqueNumbers); // [2, 3, 4, 5, 6, 7, 32]
```

4.4. El Objeto Map: Claves de Cualquier Tipo

Un objeto **Map** almacena pares clave-valor donde **las claves pueden ser de cualquier tipo** (objetos, funciones, primitivos), a diferencia de los objetos literales que convierten todas las claves a strings.

Propiedad/Método	Descripción
`size`	Número de pares clave-valor.
`set(clave, valor)`	Almacena un par. Devuelve el Map (encadenable).
`get(clave)`	Devuelve el valor asociado, o `undefined`.
`has(clave)`	Devuelve `true` si la clave existe.
`delete(clave)`	Elimina el par clave-valor.
`clear()`	Elimina todos los pares.

javascript

```javascript
// Problema con objetos literales: las claves se convierten a string
let john = { name: "John" };
let ben = { name: "Ben" };
let visitsCountObj = {};

visitsCountObj[john] = 123;  // Clave: "[object Object]"
visitsCountObj[ben] = 234;   // ¡Sobrescribe la anterior!
console.log(visitsCountObj["[object Object]"]); // 234

// Solución con Map: mantiene los tipos de clave
const visitsCountMap = new Map();

visitsCountMap.set(john, 123);
visitsCountMap.set(ben, 234);

console.log(visitsCountMap.get(john)); // 123
console.log(visitsCountMap.get(ben));  // 234

// Iteración sobre Map
for (const [user, count] of visitsCountMap) {
  console.log(`${user.name} visitó el sitio ${count} veces.`);
}
```

Conversiones Object ↔ Map

javascript

```javascript
// Objeto a Map
const obj = { nombre: "Ana", edad: 25 };
const map = new Map(Object.entries(obj));

// Map a Objeto
const nuevoObj = Object.fromEntries(map);
```

---

5. Funciones: Estructurando la Lógica

Las funciones son una de las herramientas más potentes en JavaScript. Permiten encapsular un conjunto de acciones bajo un nombre único, facilitando la reutilización de código y la construcción de programas modulares. En JavaScript, las funciones son **ciudadanos de primera clase**: pueden asignarse a variables, pasarse como argumentos y devolverse como valores.

> **Note**
>
> **¿Qué significa "ciudadanos de primera clase"?** Significa que las funciones se tratan como cualquier otro valor. Puedes guardar una función en una variable, meterla en un array, pasarla como argumento a otra función, o devolverla desde una función.

5.1. Sintaxis y Tipos de Funciones

JavaScript ofrece tres formas principales de crear funciones, cada una con sus particularidades:

Tipo	Sintaxis	Hoisting	`this`	Uso recomendado
Declaración	`function nombre() {}`	Sí	Dinámico	Funciones principales, métodos
Expresión	`const fn = function() {}`	No	Dinámico	Callbacks, funciones condicionales
Flecha	`const fn = () => {}`	No	Léxico (heredado)	Callbacks cortos, programación funcional

1. Declaración de función (Function Declaration)

La forma tradicional de definir funciones. Son "elevadas" (hoisted), lo que significa que **puedes llamarlas antes de declararlas** en el código.

javascript

```javascript
//  Funciona aunque la llamemos antes de la declaración
console.log(saludar("Ana")); // "¡Hola, Ana!"

function saludar(nombre) {
  return `¡Hola, ${nombre}!`;
}
```

2. Expresión de función (Function Expression)

Una función anónima asignada a una variable. **No se puede usar antes de su declaración** (no hay hoisting).

javascript

```javascript
// Error: no puedes usar 'despedir' antes de asignarla
// console.log(despedir("Ana")); // ReferenceError

const despedir = function(nombre) {
  return `¡Adiós, ${nombre}!`;
};

console.log(despedir("Ana")); // "¡Adiós, Ana!"
```

3. Función de flecha (Arrow Function - ES6)

Sintaxis más corta y moderna. La diferencia clave es que **no tiene su propio** `this`: hereda el `this` del contexto donde fue definida (ámbito léxico).

javascript

```javascript
// Sintaxis básica: (parámetros) => expresión
const duplicar = (x) => x * 2;

// Con un solo parámetro, los paréntesis son opcionales
const triplicar = x => x * 3;

// Sin parámetros, los paréntesis son obligatorios
const saludar = () => "¡Hola!";

// Con cuerpo de bloque (múltiples líneas), necesitas 'return' explícito
const dividir = (a, b) => {
  if (b === 0) return "Error: División por cero";
  return a / b;
};

console.log(duplicar(5));    // 10
console.log(dividir(10, 2)); // 5
```

> ⚠ Warning
>
> **Las funciones de flecha NO deben usarse como métodos de objetos** porque no tienen su propio `this`. En un método, `this` se refiere al objeto, pero en una flecha heredaría el `this` del contexto exterior (probablemente `window` o `undefined`).

javascript

```javascript
const persona = {
  nombre: "Carlos",

  // INCORRECTO: función flecha - 'this' no es el objeto
  saludarMal: () => {
    console.log(`Hola, soy ${this.nombre}`); // undefined
  },

  // CORRECTO: función tradicional - 'this' es el objeto
  saludarBien: function() {
    console.log(`Hola, soy ${this.nombre}`); // "Carlos"
  }
};

persona.saludarMal();  // "Hola, soy undefined"
persona.saludarBien(); // "Hola, soy Carlos"
```

> Tip
>
> **¿Cuándo usar cada tipo?**
>
> - **Declaración**: Funciones principales que necesitas disponibles en todo el archivo
> - **Expresión**: Cuando quieres controlar explícitamente cuándo está disponible la función
> - **Flecha**: Callbacks, funciones cortas, `map` / `filter` / `reduce`, y cuando necesitas preservar el `this` del contexto

5.2. Parámetros: Por Valor vs. Por Referencia

Cuando pasas un argumento a una función, JavaScript **siempre pasa una copia**. Pero lo que se copia depende del tipo de dato:

Analogía para entenderlo:

- **Primitivos** (Number, String, Boolean): Es como **fotocopiar un documento**. La función recibe una fotocopia, y si la modifica, el original sigue intacto.
- **Objetos** (Object, Array, Function): Es como **compartir la dirección de una casa**. Ambos tienen la misma dirección, así que si uno cambia algo dentro de la casa, el otro también lo ve.

Tipo	¿Qué se copia?	¿Afecta al original?
Primitivos (Number, String, Boolean, etc.)	El valor en sí (10)	No, la función trabaja con la copia
Objetos (Object, Array, Function)	La referencia (dirección en memoria)	Sí, si modificas propiedades/elementos

javascript

```javascript
// ========== PRIMITIVOS: Paso por valor ==========
// La función recibe una COPIA del valor

function intentarCambiar(num) {
  num = 999;  // Solo cambia la copia local
  console.log("Dentro de la función:", num); // 999
}

let miNumero = 10;
intentarCambiar(miNumero);
console.log("Variable original:", miNumero); // 10 - ¡Sin cambios!
```

javascript

```javascript
// ========== OBJETOS: Paso por referencia (del valor) ==========
// La función recibe una COPIA de la dirección de memoria
// Ambas "direcciones" apuntan al mismo objeto

function agregarElemento(arr) {
  arr.push("nuevo");  // Modifica el objeto al que ambos apuntan
  console.log("Dentro:", arr);
}

let miArray = ["a", "b", "c"];
agregarElemento(miArray);
console.log("Fuera:", miArray); // ["a", "b", "c", "nuevo"] - ¡SÍ cambió!
```

> **Caution**
>
> **¡Cuidado!** Reasignar el parámetro dentro de la función NO afecta la variable original. Solo "rompe" el enlace local a otra dirección.

javascript

```javascript
function intentarReasignar(arr) {
  // Esto NO funciona como esperarías
  arr = ["completamente", "nuevo"]; // Solo cambia la copia local de la referencia
  console.log("Dentro:", arr); // ["completamente", "nuevo"]
}

let original = [1, 2, 3];
intentarReasignar(original);
console.log("Fuera:", original); // [1, 2, 3] - ¡Sin cambios!
// La variable 'original' sigue apuntando al array original
```

> **Tip**
>
> **Para evitar modificar el original accidentalmente**, crea una copia antes de pasarlo:
>
> ```javascript
> // Clonar arrays
> modificarArray([...miArray]);  // Spread operator
> modificarArray(miArray.slice()); // slice sin argumentos
>
> // Clonar objetos
> modificarObjeto({...miObjeto}); // Spread operator
> modificarObjeto(Object.assign({}, miObjeto));
> ```

5.3. El Alcance (Scope) de las Variables

El **scope** (alcance) de una variable determina **dónde puede ser accedida** dentro del código. Piensa en el scope como las "paredes" que limitan la visibilidad de una variable.

Tipos de Scope en JavaScript:

- **Scope Global**: Variables declaradas fuera de cualquier función o bloque. Son accesibles desde **cualquier parte** del código.
- **Scope de Función**: Variables declaradas dentro de una función. Solo son accesibles **dentro de esa función**.
- **Scope de Bloque**: Variables declaradas dentro de un bloque `{}` (if, for, while). Solo accesibles **dentro de ese bloque** (solo para `let` y `const`).

javascript

```javascript
// SCOPE GLOBAL - accesible desde cualquier lugar
const global = "Soy global";

function ejemplo() {
  // SCOPE DE FUNCIÓN - solo accesible dentro de 'ejemplo'
  const funcionScope = "Solo existo en esta función";

  if (true) {
    // SCOPE DE BLOQUE - solo accesible dentro de este 'if'
    const bloqueScope = "Solo existo en este bloque";
    console.log(global);       // Accesible
    console.log(funcionScope); // Accesible
```

```
    console.log(bloqueScope);  // Accesible
  }

  // console.log(bloqueScope); // ReferenceError - fuera de su scope
}

// console.log(funcionScope); // ReferenceError - fuera de su scope
```

`var` vs `let` / `const`: La diferencia crucial

Característica	`var`	`let` / `const`
Ámbito	Función	Bloque (`{}`)
Re-declaración	Permitida	No permitida
Hoisting	Se eleva e inicializa como `undefined`	Se eleva pero NO se inicializa (Temporal Dead Zone)

javascript

```
// El problema de 'var' con bucles
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log("var:", i), 100);
}
// Salida: var: 3, var: 3, var: 3 (¡todas son 3!)
// Porque 'var' tiene scope de función, no de bloque

for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log("let:", j), 100);
}
// Salida: let: 0, let: 1, let: 2 (¡correcto!)
// Porque 'let' crea una nueva variable en cada iteración
```

javascript

```
// Problema de "shadowing" (ocultación) con var
var chica = "Aurora"; // Variable global

function demo() {
  var chica = "Raquel"; // Variable LOCAL que oculta la global
  console.log("Dentro:", chica); // "Raquel"
}

demo();
console.log("Fuera:", chica); // "Aurora" - La global no cambió

// let previene redeclaración accidental
let x = 10;
// let x = 20; // SyntaxError: Identifier 'x' has already been declared
```

> **Tip**
>
> **Regla de oro**: Usa `const` por defecto para todo. Solo usa `let` cuando **realmente necesites** reasignar el valor. Evita `var` completamente en código moderno.

5.4. Funciones Anidadas (Encapsulación)

JavaScript permite definir funciones dentro de otras funciones. Esto se conoce como **anidación de funciones** y es una técnica poderosa para la **encapsulación**: la función interna es **privada** y solo accesible desde la función contenedora.

¿Por qué usar funciones anidadas?

- **Ocultar detalles de implementación**: La lógica auxiliar queda "escondida" dentro de la función principal.
- **Evitar contaminar el scope global**: Las funciones internas no existen fuera de su contenedor.
- **Acceso al scope padre**: La función interna puede acceder a las variables de la función que la contiene (esto es la base de los **closures**).

javascript

```javascript
function hipotenusa(a, b) {
  // Función PRIVADA: solo existe dentro de 'hipotenusa'
  function cuadrado(x) {
    return x * x;
  }

  return Math.sqrt(cuadrado(a) + cuadrado(b));
}

console.log(hipotenusa(3, 4)); // 5

// cuadrado(3); // ReferenceError: cuadrado is not defined
// No podemos llamar a 'cuadrado' desde fuera
```

Closures: La función interna "recuerda" su entorno

Un **closure** (clausura) ocurre cuando una función interna "recuerda" las variables del scope donde fue creada, incluso después de que la función externa haya terminado de ejecutarse.

javascript

```javascript
function crearContador() {
  let cuenta = 0;  // Variable privada

  // La función interna "recuerda" la variable 'cuenta'
  return function() {
    cuenta++;
    return cuenta;
  };
}

const contador1 = crearContador();
console.log(contador1()); // 1
console.log(contador1()); // 2
console.log(contador1()); // 3

// Cada llamada a crearContador() crea un closure independiente
const contador2 = crearContador();
console.log(contador2()); // 1 (su propia 'cuenta' separada)
```

> **Tip**
>
> Los closures son muy útiles para crear **datos privados** en JavaScript. La variable `cuenta` del ejemplo no puede ser modificada desde fuera, solo a través de la función que devolvemos.

5.5. Funciones Globales Predefinidas

JavaScript proporciona **funciones globales** que están disponibles en cualquier parte del código sin necesidad de importar nada. Estas funciones no pertenecen a ningún objeto específico y se pueden usar directamente.

Función	Descripción
`parseInt(string, radix)`	Convierte una cadena a entero (especificar radix 10).
`parseFloat(string)`	Convierte una cadena a número decimal.
`isNaN(value)`	Comprueba si el valor es `NaN`.
`isFinite(value)`	Comprueba si es un número finito.
`encodeURI()` / `decodeURI()`	Codifica/decodifica una URI.
`eval(string)`	Ejecuta código JavaScript en una cadena. **Evitar por seguridad.**

Conversión de cadenas a números

> javascript

```javascript
// parseInt - convierte a número ENTERO
parseInt("42");        // 42
parseInt("42.9");      // 42 (trunca decimales)
parseInt("42px");      // 42 (ignora caracteres no numéricos al final)
parseInt("abc");       // NaN (no puede convertir)

// IMPORTANTE: Siempre especifica la base (radix) para evitar sorpresas
parseInt("08");        // Puede dar 0 en navegadores antiguos (octal)
parseInt("08", 10);    // 8 - Correcto, base 10

// parseFloat - convierte a número DECIMAL
parseFloat("3.14");      // 3.14
parseFloat("3.14.159");  // 3.14 (solo el primer decimal válido)
parseFloat("314e-2");    // 3.14 (notación científica)
```

Validación de números

> javascript

```javascript
// isNaN - ¿Es "Not a Number"?
isNaN(NaN);       // true
isNaN(123);       // false
isNaN("hola");    // true (no se puede convertir a número)
isNaN("123");     // false (se convierte a 123)

// Mejor usar Number.isNaN() - más estricto, no convierte
Number.isNaN(NaN);     // true
Number.isNaN("hola");  // false (no intenta convertir)

// isFinite - ¿Es un número finito?
isFinite(42);         // true
isFinite(Infinity);   // false
isFinite(-Infinity);  // false
isFinite(NaN);        // false
```

Codificación de URIs

> javascript

```javascript
// encodeURI - codifica una URL completa (preserva :, /, ?, &, =)
const url = "https://ejemplo.com/buscar?q=hola mundo";
encodeURI(url);  // "https://ejemplo.com/buscar?q=hola%20mundo"

// encodeURIComponent - codifica TODO (para parámetros de URL)
const parametro = "nombre=Juan&edad=30";
encodeURIComponent(parametro);  // "nombre%3DJuan%26edad%3D30"

// decodeURI / decodeURIComponent - deshacen la codificación
decodeURI("hola%20mundo");  // "hola mundo"
```

> **Caution**
>
> **Evita `eval()` siempre que sea posible.** Ejecutar código desde una cadena es un riesgo de seguridad grave (ataques de inyección de código) y hace el código difícil de depurar y optimizar.

---

6. El Modelo de Objetos en JavaScript

JavaScript es un lenguaje **basado en objetos** cuyo modelo se fundamenta en **prototipos**, no en clases tradicionales como Java o C++.

¿Qué es un objeto en JavaScript?

Un objeto es simplemente una **colección de pares clave-valor**, donde:

- Las **claves** (o propiedades) son strings o Symbols
- Los **valores** pueden ser de cualquier tipo: números, strings, arrays, funciones, incluso otros objetos

Cuando el valor de una propiedad es una función, la llamamos **método**.

> javascript

```javascript
// Un objeto es como una "ficha" con información
const usuario = {
  nombre: "María",           // propiedad con valor string
  edad: 30,                  // propiedad con valor número
  hobbies: ["leer", "nadar"],// propiedad con valor array
  saludar: function() {      // propiedad con valor función = MÉTODO
    console.log(`Hola, soy ${this.nombre}`);
  }
};

usuario.saludar(); // "Hola, soy María"
```

> **Note**
>
> A diferencia de lenguajes como Java donde necesitas definir una clase antes de crear un objeto, en JavaScript puedes crear objetos "al vuelo" sin definir ninguna estructura previa. Esto hace que JavaScript sea muy flexible pero también requiere más disciplina.

6.1. Creación de Objetos con Funciones Constructoras

Una **función constructora** actúa como una "plantilla" o "molde" para crear múltiples objetos con la misma estructura. Por convención, su nombre comienza con **mayúscula** para distinguirla de funciones normales.

¿Cómo funciona?

1. Se llama con la palabra clave `new`
2. JavaScript crea automáticamente un objeto vacío
3. `this` dentro de la función apunta a ese nuevo objeto
4. Las propiedades se asignan al objeto usando `this.propiedad = valor`
5. La función devuelve automáticamente el nuevo objeto

> javascript

```javascript
// Función constructora
function Coche(marca, combustible) {
  // Propiedades
  this.marca = marca;
  this.combustible = combustible;
  this.cantidad = 0;

  // Método (función anónima)
  this.rellenarDeposito = function(litros) {
    this.cantidad = litros;
  };
}

// Crear instancias con 'new'
const cocheMartin = new Coche("Volkswagen Golf", "gasolina");
const cocheSilvia = new Coche("Mercedes SLK", "diesel");

console.log(cocheMartin.marca); // "Volkswagen Golf"
cocheSilvia.rellenarDeposito(50);
console.log(cocheSilvia.cantidad); // 50
```

6.2. Objetos Literales

Un **objeto literal** es la forma más directa y concisa de crear un objeto en JavaScript. Se define usando llaves `{}` con pares `clave: valor` separados por comas.

¿Cuándo usar objetos literales vs constructores?

- **Objeto literal**: Cuando necesitas un objeto **único** (configuración, datos, estado)
- **Constructor/Clase**: Cuando necesitas crear **múltiples objetos** con la misma estructura

> javascript

```javascript
// Objeto literal - ideal para objetos únicos
const avion = {
  marca: "Boeing",
  modelo: "747",
  pasajeros: 450,

  // Método (sintaxis ES6 abreviada)
  despegar() {
    console.log(`El ${this.marca} ${this.modelo} está despegando.`);
  }
};
```

Dos formas de acceder a las propiedades:

> javascript

```javascript
// 1. Notación de punto - más común y legible
console.log(avion.marca);  // "Boeing"

// 2. Notación de corchetes - útil cuando:
//    - La propiedad tiene caracteres especiales o espacios
//    - La propiedad viene de una variable
console.log(avion["modelo"]);  // "747"

const propiedad = "pasajeros";
console.log(avion[propiedad]);  // 450

// Añadir o modificar propiedades dinámicamente
avion.color = "blanco";           // Añade nueva propiedad
avion["año"] = 2020;              // Añade con nombre especial
delete avion.color;               // Elimina una propiedad
```

Arrays de objetos literales (muy común en APIs y bases de datos):

javascript

```javascript
const usuarios = [
  { id: 1, nombre: "Ana", activo: true },
  { id: 2, nombre: "Carlos", activo: false },
  { id: 3, nombre: "María", activo: true }
];

// Filtrar usuarios activos
const activos = usuarios.filter(u => u.activo);

// Obtener solo los nombres
const nombres = usuarios.map(u => u.nombre);
```

6.3. Iteración sobre Propiedades

Existen varias formas de recorrer las propiedades de un objeto:

Método	Devuelve	Uso
`Object.keys(obj)`	`["clave1", "clave2"]`	Solo las claves
`Object.values(obj)`	`[valor1, valor2]`	Solo los valores
`Object.entries(obj)`	`[["clave", valor], ...]`	Pares clave-valor
`for...in`	Claves (strings)	Bucle tradicional

javascript

```javascript
const persona = { nombre: "Ana", edad: 28, ciudad: "Madrid" };

// Object.keys() - obtener array de claves
Object.keys(persona);  // ["nombre", "edad", "ciudad"]

// Object.values() - obtener array de valores
Object.values(persona);  // ["Ana", 28, "Madrid"]

// Object.entries() - obtener array de pares [clave, valor]
Object.entries(persona);
// [["nombre", "Ana"], ["edad", 28], ["ciudad", "Madrid"]]
```

```
// Iterar con destructuring (muy elegante)
for (const [clave, valor] of Object.entries(persona)) {
  console.log(`${clave}: ${valor}`);
}
// nombre: Ana
// edad: 28
// ciudad: Madrid
```

> **Tip**
>
> `Object.entries()` + destructuring es la forma más moderna y legible de iterar sobre las propiedades de un objeto.

6.4. El Sistema de Prototipos

El **prototipo** es el mecanismo que JavaScript usa para la **herencia**. Piensa en él como un "padre" del que un objeto puede heredar propiedades y métodos.

¿Cómo funciona?

- Cada objeto tiene un enlace interno (`[[Prototype]]`) a otro objeto
- Cuando accedes a una propiedad que NO existe en el objeto, JavaScript la busca en su prototipo
- Si tampoco está ahí, sigue subiendo por la **cadena de prototipos** hasta llegar a `null`

¿Por qué usar el prototipo?

Cuando defines un método dentro del constructor, **se crea una copia nueva** del método para cada instancia. Usando `prototype` , **todas las instancias comparten el mismo método**, ahorrando memoria.

> javascript

```
// INEFICIENTE: el método se duplica en cada instancia
function PersonaMala(nombre) {
  this.nombre = nombre;
  this.saludar = function() {  // Se crea una copia nueva cada vez
    console.log(`Hola, soy ${this.nombre}`);
  };
}

// EFICIENTE: el método se comparte vía prototipo
function Persona(nombre) {
  this.nombre = nombre;
}

// Un solo método compartido por TODAS las instancias
Persona.prototype.saludar = function() {
  console.log(`Hola, soy ${this.nombre}`);
};

const persona1 = new Persona("Alice");
const persona2 = new Persona("Bob");

persona1.saludar(); // "Hola, soy Alice"
persona2.saludar(); // "Hola, soy Bob"

// Ambos usan el MISMO método
console.log(persona1.saludar === persona2.saludar); // true
```

> **Warning**

> **No uses funciones de flecha para métodos en `prototype`**. Las arrow functions no tienen su propio `this`, heredan el del contexto donde fueron definidas, causando comportamientos inesperados.

6.5. Clases de ES6: Sintaxis Moderna

Las clases de ES6 son **azúcar sintáctico** (syntactic sugar) sobre el sistema de prototipos. **No cambian cómo funciona JavaScript internamente**, solo ofrecen una sintaxis más clara y familiar para quienes vienen de otros lenguajes.

Ventajas de las clases:

- Sintaxis más limpia y organizada
- Todo el código de la "plantilla" está en un solo lugar
- Soporte nativo para herencia con `extends`
- Getters, setters y métodos estáticos más fáciles de definir

> javascript

```javascript
// Definición de clase
class Coche {
  constructor(marca, combustible) {
    this.marca = marca;
    this.combustible = combustible;
    this.cantidad = 0;
  }

  // Método de instancia
  rellenarDeposito(litros) {
    this.cantidad = litros;
  }

  // Getter
  get info() {
    return `${this.marca} (${this.combustible})`;
  }

  // Método estático
  static compararMarcas(coche1, coche2) {
    return coche1.marca === coche2.marca;
  }
}

const miCoche = new Coche("Tesla Model S", "eléctrico");
miCoche.rellenarDeposito(100);
console.log(miCoche.info); // "Tesla Model S (eléctrico)"
```

Herencia con `extends` y `super`

> javascript

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

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

```javascript
class Perro extends Animal {
  constructor(nombre, raza) {
    super(nombre); // Llama al constructor de Animal
    this.raza = raza;
  }

  hablar() {
    super.hablar(); // Llama al método del padre
    console.log(`${this.nombre} ladra.`);
  }
}

const miPerro = new Perro("Max", "Labrador");
miPerro.hablar();
// "Max hace un ruido."
// "Max ladra."
```

---

7. Paradigma de Programación Funcional

La **programación funcional (PF)** es un paradigma que se centra en el uso de **funciones puras**, **inmutabilidad** y **composición de funciones**. En lugar de dar instrucciones paso a paso (imperativo), describe **qué** se quiere lograr de manera **declarativa**.

7.1. Enfoque Imperativo vs. Declarativo

> javascript

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

// IMPERATIVO: describe CÓMO hacerlo
const doblesImperativo = [];
for (let i = 0; i < numeros.length; i++) {
  doblesImperativo.push(numeros[i] * 2);
}

// DECLARATIVO/FUNCIONAL: describe QUÉ hacer
const doblesDeclarativo = numeros.map(n => n * 2);

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

7.2. Inmutabilidad: Evitando Efectos Secundarios

La inmutabilidad significa que **los datos no se modifican** después de su creación. En lugar de cambiar estructuras existentes, se crean nuevas.

> javascript

```javascript
// PROBLEMA: Efecto secundario no deseado
let foo = [1, 2, 3];
let bar = foo; // bar es una REFERENCIA, no una copia

bar.push(10000);
console.log(foo); // [1, 2, 3, 10000] - ¡foo fue alterado!

// SOLUCIÓN: Crear una copia
let baz = [1, 2, 3];
```

```javascript
let qux = [...baz]; // Operador spread crea nueva copia

qux.push(10000);
console.log(baz); // [1, 2, 3] - Sin cambios
```

Object.freeze() y Object.seal(): Controlando la mutabilidad

JavaScript proporciona dos métodos para limitar la modificación de objetos:

- `Object.freeze(obj)` : "Congela" un objeto completamente. Una vez congelado, **no puedes hacer nada**: ni modificar valores, ni añadir propiedades, ni eliminarlas. Es como meter el objeto en hielo.
- `Object.seal(obj)` : "Sella" un objeto. **Puedes modificar los valores existentes**, pero no puedes añadir ni eliminar propiedades. Es como cerrar una caja con candado, pero aún puedes cambiar lo que hay dentro.

Característica	Object.freeze()	Object.seal()
Modificar valores existentes	No	Sí
Agregar propiedades	No	No
Eliminar propiedades	No	No
Inmutabilidad profunda	No (solo superficial)	No

javascript

```javascript
// ========== Object.freeze() ==========
const usuario = { nombre: "Ana", edad: 25 };
Object.freeze(usuario);

usuario.edad = 30;           // No hace nada (silenciosamente ignorado)
usuario.email = "a@mail.com"; // No se añade
delete usuario.nombre;       // No se elimina

console.log(usuario); // { nombre: "Ana", edad: 25 } - Sin cambios

// Verificar si está congelado
console.log(Object.isFrozen(usuario)); // true


// ========== Object.seal() ==========
const producto = { nombre: "Laptop", precio: 1000 };
Object.seal(producto);

producto.precio = 1200;       // Sí funciona - modificar valores permitido
producto.stock = 50;          // No se añade
delete producto.nombre;       // No se elimina

console.log(producto); // { nombre: "Laptop", precio: 1200 } - Precio actualizado

// Verificar si está sellado
console.log(Object.isSealed(producto)); // true
```

> ⚠ Warning
>
> **Limitación importante**: Tanto `freeze()` como `seal()` son **superficiales**. Si el objeto tiene propiedades que son objetos anidados, esos objetos internos **siguen siendo mutables**.

javascript

```javascript
// Problema: Inmutabilidad superficial
const config = {
  tema: "oscuro",
  opciones: { sonido: true, notificaciones: false }
};

Object.freeze(config);

config.tema = "claro";              // No funciona (primer nivel congelado)
config.opciones.sonido = false;     // SÍ funciona! (objeto anidado NO congelado)

console.log(config.opciones.sonido); // false - ¡El objeto interno cambió!


// Solución: Deep Freeze (congelación profunda recursiva)
function deepFreeze(obj) {
  Object.freeze(obj);
  Object.values(obj)
    .filter(value => typeof value === "object" && value !== null)
    .forEach(value => deepFreeze(value));
  return obj;
}

const configSegura = deepFreeze({
  tema: "oscuro",
  opciones: { sonido: true }
});

configSegura.opciones.sonido = false; // Ahora tampoco funciona
console.log(configSegura.opciones.sonido); // true - ¡Protegido!
```

7.3. Funciones Puras

Una **función pura** es como una máquina de calcular: metes los mismos números, siempre obtienes el mismo resultado, y la máquina no hace nada más que calcular (no enciende luces, no guarda datos).

Las dos reglas de una función pura:

1. **Determinista**: Misma entrada → misma salida (siempre, sin importar cuándo la llames)
2. **Sin efectos secundarios**: No modifica nada fuera de sí misma

Ejemplos de "efectos secundarios" (cosas que hacen una función impura):

Efecto secundario	Ejemplo
Modificar un argumento	`arr.push()` , `obj.propiedad = x`
Modificar variable externa	`contador++` , `global = valor`
Escribir en consola	`console.log()`
Hacer peticiones HTTP	`fetch()` , `XMLHttpRequest`
Acceder a la fecha/hora	`Date.now()` , `new Date()`
Generar números aleatorios	`Math.random()`

javascript

```javascript
// IMPURA: modifica el array de entrada
function appendSumImpure(arr) {
```

```
  const total = arr.reduce((acc, val) => acc + val, 0);
  arr.push(total);  // EFECTO SECUNDARIO! Muta el array original
  return arr;
}

const original = [3, 2];
console.log(appendSumImpure(original)); // [3, 2, 5]
console.log(appendSumImpure(original)); // [3, 2, 5, 10] - ¡Resultado DIFERENTE!
// La misma función con la misma variable da resultados diferentes
```

javascript

```
// PURA: devuelve un nuevo array sin tocar el original
function appendSumPure(arr) {
  const total = arr.reduce((acc, val) => acc + val, 0);
  return [...arr, total];  // Spread operator crea nueva copia
}

const datos = [3, 2];
console.log(appendSumPure(datos)); // [3, 2, 5]
console.log(appendSumPure(datos)); // [3, 2, 5] - Siempre el MISMO resultado
console.log(datos);                // [3, 2] - Original SIN cambios
```

Más ejemplos para comparar:

javascript

```
// IMPURA: usa variable externa y console.log
let total = 0;
function sumarImpuro(n) {
  total += n;        // Modifica variable externa
  console.log(n);    // Efecto secundario
  return total;
}

// PURA: solo depende de sus argumentos
function sumar(a, b) {
  return a + b;
}

// IMPURA: no determinista (resultado diferente cada vez)
function obtenerTiempo() {
  return Date.now();
}

// PURA: determinista y sin efectos
function formatearFecha(timestamp) {
  return new Date(timestamp).toLocaleDateString();
}
```

> Tip
>
> ¿Por qué importan las funciones puras?
>
> - Son **predecibles**: fáciles de entender y depurar
> - Son **testeables**: no necesitas configurar estado externo
> - Son **paralelizables**: no hay conflictos de estado
> - Facilitan la **reutilización** del código

7.4. Funciones de Primer Orden vs. Funciones de Orden Superior

En JavaScript, las funciones son **ciudadanos de primera clase**, lo que significa que pueden tratarse como cualquier otro valor: asignarse a variables, pasarse como argumentos y devolverse como resultado.

Función de Primer Orden (First-Order Function)

Una función de primer orden es una función "normal" que:

- Solo opera sobre datos (números, strings, objetos, etc.)
- **No recibe funciones como parámetros**
- **No devuelve funciones**

javascript

```javascript
// Función de primer orden: solo trabaja con datos
function sumar(a, b) {
  return a + b;
}

function saludar(nombre) {
  return `Hola, ${nombre}`;
}

console.log(sumar(2, 3));        // 5
console.log(saludar("María"));   // "Hola, María"
```

Función de Orden Superior (Higher-Order Function)

Una función de orden superior es aquella que cumple **al menos una** de estas condiciones:

1. **Recibe una o más funciones como argumentos (callback)**
2. **Devuelve una función como resultado**

javascript

```javascript
// 1. Recibe una función como argumento
function ejecutarOperacion(a, b, operacion) {
  return operacion(a, b);
}

const resultado = ejecutarOperacion(5, 3, (x, y) => x * y);
console.log(resultado); // 15

// 2. Devuelve una función
function crearMultiplicador(factor) {
  return function(numero) {
    return numero * factor;
  };
}

const duplicar = crearMultiplicador(2);
const triplicar = crearMultiplicador(3);

console.log(duplicar(5));   // 10
console.log(triplicar(5));  // 15
```

> Note

> Los métodos `map`, `filter` y `reduce` son funciones de orden superior porque reciben una función (callback) como argumento.

`map()` - Transformar cada elemento

> javascript

```javascript
const precios = [10, 20, 30];
const preciosConIVA = precios.map(precio => precio * 1.21);
console.log(preciosConIVA); // [12.1, 24.2, 36.3]
```

> Error común con `parseInt`:
>
> ```javascript
> ["1", "2", "3"].map(parseInt); // [1, NaN, NaN] ¡Inesperado!
>
> // map pasa (valor, índice, array), parseInt usa índice como base
> // Solución:
> ["1", "2", "3"].map(str => parseInt(str, 10)); // [1, 2, 3]
> ```

`filter()` - Seleccionar elementos

> javascript

```javascript
const productos = [
  { nombre: "Camiseta", precio: 25 },
  { nombre: "Pantalón", precio: 60 },
  { nombre: "Gorra", precio: 15 }
];

const ofertas = productos.filter(p => p.precio < 30);
console.log(ofertas); // [{ nombre: "Camiseta", ... }, { nombre: "Gorra", ... }]
```

`reduce()` - Reducir a un único valor

> javascript

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

// Suma de todos los elementos
const suma = numeros.reduce((acumulador, actual) => acumulador + actual, 0);
console.log(suma); // 15

// Contar frecuencia de elementos
const letras = ["a", "b", "a", "c", "b", "a"];
const frecuencia = letras.reduce((acc, letra) => {
  acc[letra] = (acc[letra] || 0) + 1;
  return acc;
}, {});
console.log(frecuencia); // { a: 3, b: 2, c: 1 }
```

> **Note**
>
> **Buena Práctica**: Proporciona siempre un valor inicial a `reduce()` para evitar errores con arrays vacíos.

7.5. Implementando filter, map y reduce desde Cero

Para comprender profundamente cómo funcionan estas funciones de orden superior, es muy instructivo implementarlas manualmente. Esto demuestra el poder del sistema de **prototipos** de JavaScript y cómo podemos extender los objetos nativos.

> ⚠ Warning
>
> Modificar el `prototype` de objetos nativos como `Array` no es recomendable en código de producción, ya que puede causar conflictos. Aquí lo hacemos con fines didácticos.

Implementando `filter` alternativo

El método `filter` recorre cada elemento del array, aplica una función de prueba (callback), y devuelve un **nuevo array** solo con los elementos que pasan la prueba (devuelven `true`).

javascript

```javascript
// Añadimos nuestro método al prototipo de Array
Array.prototype.miFilter = function(callback) {
  // Creamos un nuevo array para almacenar los resultados (inmutabilidad)
  const nuevoArray = [];

  // Recorremos cada elemento del array original (this)
  for (let i = 0; i < this.length; i++) {
    // Si el callback devuelve true, añadimos el elemento
    if (callback(this[i], i, this)) {
      nuevoArray.push(this[i]);
    }
  }

  return nuevoArray;
};

// Probamos nuestra implementación
const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const pares = numeros.miFilter(num => num % 2 === 0);
console.log(pares); // [2, 4, 6, 8, 10]

const mayoresQue5 = numeros.miFilter(num => num > 5);
console.log(mayoresQue5); // [6, 7, 8, 9, 10]
```

Implementando `map` alternativo

El método `map` recorre cada elemento del array, aplica una función de transformación, y devuelve un **nuevo array** con los elementos transformados. El array resultante siempre tiene la misma longitud que el original.

javascript

```javascript
Array.prototype.miMap = function(callback) {
  const nuevoArray = [];

  for (let i = 0; i < this.length; i++) {
    // Añadimos el RESULTADO de aplicar el callback a cada elemento
    nuevoArray.push(callback(this[i], i, this));
  }

  return nuevoArray;
};

// Probamos nuestra implementación
```

```javascript
const precios = [10, 20, 30, 40];

const preciosConIVA = precios.miMap(precio => precio * 1.21);
console.log(preciosConIVA); // [12.1, 24.2, 36.3, 48.4]

const duplicados = precios.miMap(precio => precio * 2);
console.log(duplicados); // [20, 40, 60, 80]
```

Implementando `reduce` alternativo

El método `reduce` es el más versátil. Recorre el array acumulando un valor que se pasa de iteración en iteración, reduciendo todo el array a un **único valor** (que puede ser un número, string, objeto, o incluso otro array).

> javascript

```javascript
Array.prototype.miReduce = function(callback, valorInicial) {
  // El acumulador empieza con el valor inicial proporcionado
  let acumulador = valorInicial;

  for (let i = 0; i < this.length; i++) {
    // En cada iteración, el callback recibe:
    // - El acumulador actual
    // - El elemento actual
    // - El índice actual
    // - El array completo
    // Y devuelve el nuevo valor del acumulador
    acumulador = callback(acumulador, this[i], i, this);
  }

  return acumulador;
};

// Probamos nuestra implementación
const numeros = [1, 2, 3, 4, 5];

// Suma de todos los elementos
const suma = numeros.miReduce((acc, num) => acc + num, 0);
console.log(suma); // 15

// Encontrar el máximo
const maximo = numeros.miReduce((acc, num) => num > acc ? num : acc, numeros[0]);
console.log(maximo); // 5

// Aplanar un array de arrays
const arrayDeArrays = [[1, 2], [3, 4], [5, 6]];
const aplanado = arrayDeArrays.miReduce((acc, arr) => acc.concat(arr), []);
console.log(aplanado); // [1, 2, 3, 4, 5, 6]
```

> **Tip**
>
> Implementar estas funciones manualmente te ayuda a entender que `filter`, `map` y `reduce` son simplemente **abstracciones** sobre bucles `for`. La diferencia es que encapsulan el patrón de iteración, permitiéndote enfocarte en la lógica de transformación.

---

8. Catálogo de Métodos de Array

Esta sección sirve como referencia rápida de los métodos de Array más importantes, organizados por categoría.

8.1. Creación

Método	Descripción	Mutabilidad
`Array.from(iterable)`	Crea array desde iterable.	Nuevo array
`Array.of(el1, el2, ...)`	Crea array con los argumentos.	Nuevo array
`Array.isArray(obj)`	Comprueba si es un Array.	N/A

8.2. Acceso y Modificación de Extremos

Método	Descripción	Mutabilidad
`at(index)`	Elemento en índice (acepta negativos).	No modifica
`push(items)`	Añade al final.	Modifica
`pop()`	Elimina del final.	Modifica
`unshift(items)`	Añade al inicio.	Modifica
`shift()`	Elimina del inicio.	Modifica

8.3. Transformación (Inmutables)

Método	Descripción	Mutabilidad
`map(callback)`	Transforma cada elemento.	Nuevo array
`filter(callback)`	Filtra según condición.	Nuevo array
`slice(start, end)`	Extrae porción.	Nuevo array
`concat(arrays)`	Une arrays.	Nuevo array
`flat(depth)`	Aplana arrays anidados.	Nuevo array
`flatMap(callback)`	map + flat(1).	Nuevo array
`toSorted(compareFn)`	Ordena (inmutable, ES2023).	Nuevo array
`toReversed()`	Invierte (inmutable, ES2023).	Nuevo array

8.4. Modificación (Mutables)

Método	Descripción	Mutabilidad
`sort(compareFn)`	Ordena el array.	**Modifica**
`reverse()`	Invierte el orden.	**Modifica**
`splice(start, count, items)`	Elimina/añade elementos.	**Modifica**
`fill(value, start, end)`	Rellena con valor.	**Modifica**

8.5. Búsqueda y Validación

Método	Descripción	Retorno
`find(callback)`	Primer elemento que cumple.	Valor o `undefined`
`findIndex(callback)`	Índice del primero que cumple.	Índice o `-1`
`indexOf(value)`	Primer índice del valor.	Índice o `-1`
`includes(value)`	¿Contiene el valor?	`boolean`
`every(callback)`	¿Todos cumplen?	`boolean`
`some(callback)`	¿Alguno cumple?	`boolean`

8.6. Reducción e Iteración

Método	Descripción	Retorno
`reduce(callback, initial)`	Reduce a un valor (izq→der).	Valor único
`reduceRight(callback, initial)`	Reduce (der→izq).	Valor único
`forEach(callback)`	Ejecuta por cada elemento.	`undefined`

8.7. Conversión

Método	Descripción
`join(separator)`	Une elementos en string.
`toString()`	Convierte a string (comas).
`entries()`	Iterador de [índice, valor].
`keys()`	Iterador de índices.
`values()`	Iterador de valores.
