Lenguaje de Marcas: Javascript

Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

Evaluación:

- Para la evaluación del módulo, se aplicará el siguiente criterio al finalizar cada una de las evaluaciones:
  - El 30% de la calificación corresponderá a la entrega de las prácticas.
  - El 70% de la calificación lo determinará dos examenes teórico/prácticos.
- Es importante destacar que se debe obtener al menos un **4,5 en cada parte** para poder realizar el promedio final entre ambas.

Introducción a JavaScript

¿Qué es JavaScript?

- Es un lenguaje de **programación interpretado**.
- Comúnmente usado para el desarrollo web.
- Ejecutado por un motor integrado en navegadores (como V8 de Chrome).

Características principales:

- Lenguaje de debilmente tipado y dinámico.
- Multi-paradigma: Soporta programación orientada a objetos, funcional e imperativa.
- Interactúa con HTML y CSS para crear experiencias dinámicas.

ECMAScript y JavaScript

¿Qué es ECMAScript?

- Es el estándar que define las características de JavaScript.
- Versiones importantes: ES5, ES6 (ES2015), y posteriores (hasta ES2023).

¿Por qué es importante?

Define mejoras en el lenguaje, como:

- `let`, `const`, y `var` para declarar variables.
- Funciones flecha (`=>`)
- Clases y módulos.

Herramientas para JavaScript

Editores recomendados:

- **Visual Studio Code** : Ligero, extensible y con soporte para JavaScript y Node.js.
    - Descarga: https://code.visualstudio.com/
- Otros: WebStorm, Atom, Sublime Text.

Editores online:

JSFiddle: https://jsfiddle.net/
CodePen: https://codepen.io/

Herramientas del navegador:

Chrome DevTools (F12): Depura y prueba código JavaScript en el navegador.
O las DevTools del resto de navegadores

Integrar JavaScript en HTML

Inline (no recomendado):

```
<body>
 <script>
  console.log("Hola Mundo!");
 </script>
</body>
```

Archivo externo (recomendado):

```
<body>
  <script src="js/app.js"></script>
</body>
```

Archivo app.js

```
console.log("Hola Mundo!");
```

Funciones de E/S

Funciones de salida:

console.log(mensaje) y **console.error(texto)**

Muestra información en la consola del navegador.

```
console.log("Esto es un mensaje en la consola");
```

alert(mensaje)

Muestra un cuadro de diálogo con un mensaje al usuario.

```
alert("Bienvenido a nuestra página web");
```

document.write(mensaje)

Muestra un cuadro de diálogo con un mensaje al usuario.

```
document.write("<h1>Hola Mundo</h1>");
```

Funciones de E/S

Funciones de entrada:

prompt(mensaje):

- Muestra un cuadro de diálogo que permite al usuario introducir datos.
- Devuelve siempre un valor de tipo `string`.

```
let nombre = prompt("¿Cuál es tu nombre?");
console.log("Hola, " + nombre);
```

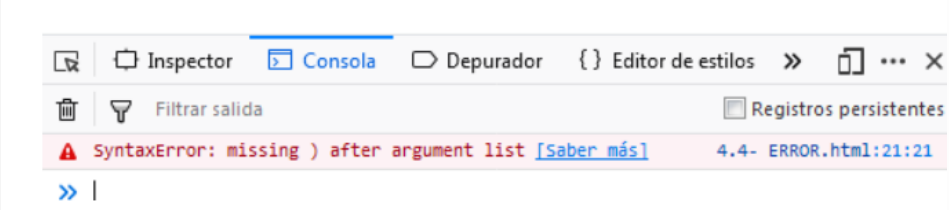
confirm()

La función confirm() en JavaScript se utiliza para mostrar un cuadro de diálogo con un mensaje, botones "Aceptar" y "Cancelar".

```
const respuesta = confirm("¿Estás seguro de que quieres continuar?");
if (respuesta) {
  console.log("El usuario hizo clic en 'Aceptar'.");
} else {
  console.log("El usuario hizo clic en 'Cancelar'.");
}
```

Errores de sintaxis

- En la solapa consola de las herramientas de desarrollador nos aparecerán los errores sintácticos que cometamos.
- El error aparecerá cuando el interprete trate de ejecutar la instrucción que contenga el error
- En el error nos indicará el fichero y la línea donde se produzca el error



Variables y tipos de datos en JS

Tipos de declaración:

- `var` (obsoleto): Alcance de función.
- `let`: Alcance de bloque, recomendado.
- `const`: Constante, no se puede reasignar.

```
let nombre = "Juan";
const edad = 25;
var saludo = "Hola";
```

Tipos de datos

Primitivos:

- `string` (cadena de texto)
- `number` (números)
- `boolean` (true/false)
- `undefined` (declarada pero sin valor asignado)
- `null` (valor intencionadamente vacío)

Objeto:

Arrays, funciones, objetos, etc.

```
let cadena = "Hola"; // string
let numero = 42; // number
let booleano = true; // boolean
let sinDefinir; // undefined
let vacio = null; // null
```

Operadores básicos

Aritméticos:

+, -, *, /, % (módulo)

Relacionales:

>, <, >=, <=, ==, !=, ===, !==

```
let a = 10, b = 20;
console.log(a + b); // 30
console.log(a > b); // false
console.log(a == "10"); // true (igualdad débil)
console.log(a === "10"); // false (igualdad estricta)
```

Variables en JS

Reglas básicas:
El nombre de una variable puede contener:

- Letras (a-z, A-Z)
- Números (0-9)
- El carácter _ (guion bajo)
- El carácter $

Importante: El primer carácter debe ser una **letra**, _, o $, **pero nunca un número.**

JavaScript es case sensitive (Sensibilidad a Mayúsculas y Minúsculas**)**

Se distingue entre mayúsculas y minúsculas en los nombres de las variables.

```
let nombre = "Juan";
let Nombre = "Pedro";

console.log(nombre); // Muestra: "Juan"
console.log(Nombre); // Muestra: "Pedro"
```

Variables en JS

Tipado Débil en JavaScript

El tipo de las variables depende del **valor asignado.**

No es necesario declarar el tipo explícitamente como en otros lenguajes.

```
let variable = 42; // Número
variable = "Hola"; // Cambia a cadena
variable = true; // Ahora es booleano
```

Nota: El tipo puede cambiar dinámicamente según el valor.

Cambio de Tipo Dinámico

JavaScript permite cambiar el tipo de una variable.

Esto puede provocar comportamientos inesperados si no se maneja correctamente.

Ejemplo:

```
let dato = "123"; // Tipo: string
dato = dato * 2; // Automáticamente se convierte a número
console.log(dato); // Resultado: 246
```

Cuidado con las operaciones dinámicas:

```
let x = "5";
let y = 3;

console.log(x + y); // Resultado: "53" (concatenación)
console.log(x * y); // Resultado: 15 (conversión automática a número)
```

Variables en JS

Buenas Prácticas

Usa nombres descriptivos para las variables:

```
let edad = 25; // Claro
let e = 25; // Poco claro
```

Utiliza camelCase para variables compuestas:

```
let nombreCompleto = "Juan Pérez";
let precioProducto = 19.99;
```

Declara variables con `let` o `const` en lugar de `var`:

```
const PI = 3.14159; // Valor constante
let total = 100; // Variable mutable
```

Evita usar nombres reservados de JavaScript:

```
let class = 10; // Inválido (class es una palabra reservada)
```

Hoisting en JS

El **hoisting** es un comportamiento de JavaScript en el que las declaraciones de variables, funciones o clases se "mueven" al principio de su contexto (función o script) durante la fase de compilación.

```
console.log(miVariable); // Resultado: undefined
var miVariable = 10;
console.log(miVariable); // Resultado: 10
```

Hoisting con **let y const:**

- Las variables declaradas con let y const también se "elevan", pero **no se inicializan automáticamente.**
- Intentar acceder a ellas antes de la declaración genera un **Error de referencia.**

```
console.log(miLet); // ReferenceError
let miLet = 20;
```

Hoisting también se da en Funciones, clases, expresiones

Tipos de variables

typeof: Indica el tipo de dato que en ese momento tiene la variable.

```
let v1 = "Hola Mundo!";
console.log(typeof v1); // Imprime -> string

v1 = 123;
console.log(typeof v1); // Imprime -> number
```

Hasta que le asignemos un valor, las variables tendrán un tipo especial conocido como **undefined**.

Este valor es diferente de **null** (que si se considera como un valor).

```
let v1;
console.log(typeof v1); // Imprime -> undefined
if (v1 === undefned) { // (!v1) or (typeof v1 === "undefned") también funciona
console.log("Has olvidado darle valor a v1");
}
```

¿Preguntas?

Lenguaje de Marcas: Javascript

Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

Numeros

```
let age = 35
const GRAVITY = 9.81
let mass = 72
const PI = 3.14
```

Class Math

```
// Redondear al número más cercano
console.log(Math.PI); // 3.141592653589793
console.log(Math.round(Math.PI)); // 3
console.log(Math.round(9.81)); // 10
// Redondeo hacia abajo
console.log(Math.floor(PI)); // 3
// Redondeo hacia arriba
console.log(Math.ceil(PI)); // 4
// Mínimo valor en una lista de números
console.log(Math.min(-5, 3, 20, 4, 5, 10)); // -5
// Máximo valor en una lista de números
console.log(Math.max(-5, 3, 20, 4, 5, 10)); // 20
```

Numeros

Class Math

```
// Crear un número aleatorio entre 0 y 0.999999
const randNum = Math.random();
console.log(randNum);
// Crear un número aleatorio entre 0 y 10
const num = Math.floor(Math.random() * 11);
console.log(num);
// Valor absoluto
console.log(Math.abs(-10)); // 10
// Raíz cuadrada
console.log(Math.sqrt(25)); // 5
console.log(Math.sqrt(2)); // 1.4142135623730951
// Potencia
console.log(Math.pow(3, 2)); // 9
```

Strings

Textos definidos entre comillas.

```
let oneSpace = ' ' // string vacio
let city = 'Benidorm'
let cita= "The saying,'Seeing is Believing' is not correct in 2021."
let citaPlantilla= `The saying,'Seeing is Believing' is not correct in 2021.`
```

Concatenación de strings

```
let pais = 'España';
let edad = 25;
let informacionPersona = 'Tengo ' + edad + ' años y vivo en ' + pais;
console.log(informacionPersona); // Tengo 25 años y vivo en España
```

Strings multilinea, utlizamos (\)

```
const parrafo = "Me llamo John Doe. Vivo en Benidorm, España.\
Soy profesor y me encanta enseñar. Enseño HTML, CSS, JavaScript, \
a cualquiera que esté interesado en aprender. \
Espero que tú también lo estés disfrutando.";
```

Strings

Secuencias de escape en cadenas de texto

```
console.log('Espero que todos lo estén disfrutando.\n¿Y tú?'); // Salto de línea
console.log('Días\tTemas\tEjercicios'); // Tabulación (equivale a 8 espacios)
console.log('Día 1\t3\t5'); // Tabulación
console.log('Este es un símbolo de barra invertida (\\)'); // Para escribir una barra invertida
console.log('En todos los lenguajes de programación comienza con \"¡Hola, Mundo!\"'); // Comillas dobles escapadas
console.log("En todos los lenguajes de programación comienza con \'¡Hola, Mundo!\'"); // Comillas simples escapadas
```

```
Espero que todos lo estén disfrutando.
¿Y tú?
Días Temas Ejercicios
Día 1 3 5
Este es un símbolo de barra invertida (\)
En todos los lenguajes de programación comienza con "¡Hola, Mundo!"
En todos los lenguajes de programación comienza con '¡Hola, Mundo!'
```

Strings

Métodos Comunes de Cadenas en JavaScript

Todos estos métodos no modifican el valor de la variable a menos que la reasignes.

```
let s1 = "Esto es un string";
// Obtener la longitud del string
console.log(s1.length); // Imprime 17
// Obtener el carácter de una cierta posición del string (Empieza en 0)
console.log(s1.charAt(0)); // Imprime "E"
// Obtiene el índice de la primera ocurrencia
console.log(s1.indexOf("s")); // Imprime 1
// Obtiene el índice de su última ocurrencia
console.log(s1.lastIndexOf("s")); // Imprime 11
// Devuelve un array con todas las coincidencias en de una expresión regular
console.log(s1.match(/.s/g)); // Imprime ["Es", "es", " s"]
// Obtiene la posición de la primera ocurrencia de una expresión regular
console.log(s1.search(/[aeiou]/)); // Imprime 3
// Reemplaza la coincidencia de una expresión regular (o string) con un string (/g opcionalmente reemplaza todas)
```

Strings

```
let s1 = "Esto es un string";
console.log(s1.replace(/i/g, "e")); // Imprime "Esto es un streng"
// Devuelve un substring (posición inicial: incluida, posición final: no incluida)
console.log(s1.slice(5, 7)); // Imprime "es"
// Igual que slice
console.log(s1.substring(5, 7)); // Imprime "es"
// Como substring pero con una diferencia (posición inicial, número de caracteres desde la posición inicial)
console.log(s1.substr(5, 7)); // Imprime "es un s"
// Transforma en minúsculas, toLowerCase no funciona con caracteres especiales (ñ, á, é, ...)
console.log(s1.toLocaleLowerCase()); // Imprime "esto es un string"
// Transforma a mayúsculas
console.log(s1.toLocaleUpperCase()); // Imprime "ESTO ES UN STRING"
// Devuelve un string eliminando espacios, tabulaciones y saltos de línea del principio y final
console.log(" String con espacios ".trim()); // Imprime "String con espacios"
```

Strings

Conversión de tipo explicita:

Puedes convertir un dato en number usando la función **Number(value).**
- Si el valor no es un número la función devolverá NaN
- Puedes también añadir el prefijo '+' antes de la variable para conseguir el mismo resultado.

```
let s1 = "32";
let s2 = "14";
console.log(Number(s1) + Number(s2)); // Imprime 46
console.log(+s1 + +s2); // Imprime 46
```

La conversión de un dato a booleano se hace usando la función **Boolean(value).**
- **Puedes añadir !! (doble negación), antes del valor para forza la conversión.**
- Estos valores equivalen a false:
  - string vacio (""), null, undefined, 0.
- Cualquier otro valor debería devolver true.

```
let v = null;
let s = "Hello";
console.log(Boolean(v)); // Imprime false
console.log(!!s); // Imprime true
```

Strings

Operadores. Suma ‘+’

- Este operador puede usarse para sumar números o concatenar cadenas.
- Pero, ¿Qué ocurre si intentamos sumar un número con un string, o algo que no sea un número o string?
- Veamos los ejemplos:

```
console.log(4 + 6); // Imprime 10
console.log("Hello " + "world!"); // Imprime "Hello world!"
console.log("23" + 12); // Imprime "2312"
console.log("42" + true); // Imprime "42true"
console.log("42" + undefined); // Imprime "42undefined"
console.log("42" + null); // Imprime "42null"
console.log(42 + "hello"); // Imprime "42hello"
console.log(42 + true); // Imprime 43 (true => 1)
console.log(42 + false); // Imprime 42 (false => 0)
console.log(42 + undefined); // Imprime NaN (undefined no puede ser convertido a number)
console.log(42 + null); // Imprime 42 (null => 0)
console.log(13 + 10 + "12"); // Imprime "2312" (13 + 10 = 23, 23 + "12" = "2312")
```

Strings

Conversiones implícitas en la suma

- Cuando hay un string, siempre se realizará una concatenación, por tanto, si el otro valor no es un string se intentará transformar en un string.
- Si no hay strings, y algún valor no es un número, lo intentará convertir a número e intentará hacer una suma.
- Si la conversión del valor a número falla, devolverá NaN (Not a Number).

Operadores aritméticos

Operadores aritméticos son: resta (-), multiplicación (*), división (/) y módulo (%).

Estos operadores operan siempre con números, por tanto, cualquier operando que no sea un número debe ser convertido a número.

```
console.log(4 * 6); // Imprime 24
console.log("Hello " * "world!"); // Imprime NaN
console.log("24" / 12); // Imprime 2 (24 / 12)
console.log("42" * true); // Imprime 42 (42 * 1)
console.log("42" * false); // Imprime 0 (42 * 0)
console.log("42" * undefined); // Imprime NaN
console.log("42" - null); // Imprime 42 (42 - 0)
console.log(12 * "hello"); // Imprime NaN
```

Operadores incremento y decremento

En JavaScript podemos preincrementar (++variable), postincrementar (variable++), predecrementar (--variable) y postdecrementar (variable--).

```
let a = 1;
let b = 5;
console.log(a++); // Imprime 1 y incrementa a (2)
console.log(++a); // Incrementa a (3), e imprime 3
console.log(++a + ++b); // Incrementa a (4) y b (6). Suma (4+6), e imprime 10
console.log(a-- + --b); // Decrementa b (5). Suma (4+5). Imprime 9. Decrementa a (3)
```

Operador cambio de signo

Podemos usar los signos – y + delante de un número para cambiar o mantener su signo.

Si aplicamos estos operadores con un dato que no es un número, este será convertido a número primero.

Por eso, es una buena opción usar **+value para convertir a número,** lo cual equivale a usar Number(value).

```
let a = "12";
let b = "13";
let c = true;
console.log(a + b); // Imprime "1213"
console.log(+a + +b); // Imprime 25 (12 + 13)
console.log(+b + +c); // Imprime 14 (13 + 1). True -> 1
```

Operadores relacionales

- El operador de comparación, compara dos valores y devuelve un booleano (true o false)
- Estos operadores son prácticamente los mismos que en la mayoría de lenguajes de programación, a excepción de algunos, que veremos a continuación.
- Podemos usar == o === para comparar la igualdad (o lo contrario !=, !==).
- La principal diferencia es que el primero, no tiene en cuenta los tipos de datos que están siendo comparados, compara si los valores son equivalentes.
- Cuando usamos ===, los valores además deben ser del mismo tipo.
- Si el tipo de dato o el valor son diferentes devolverá falso.
- Devolverá true cuando ambos valores son idénticos y del mismo tipo.

Operadores relacionales

```
console.log(3 == "3"); // true
console.log(3 === "3"); // false
console.log(3 != "3"); // false
console.log(3 !== "3"); // true
// Equivalente a falso (todo lo demás es equivalente a cierto)
console.log("" == false); // true
console.log(false == null); // false (null no es equivalente a cualquier boolean).
console.log(false == undefined); // false (undefined no es equivalente a
cualquier boolean).
console.log(null == undefined); // true (regla especial de JavaScript)
console.log(0 == false); // true
console.log({} >= false); // Object vacío -> false
console.log([] >= false); // Array vacío -> true
```

Otros operadores relacionales

Otros operadores relaciones para números o strings son: menor que (<), mayor que (>), menor o igual que (<=) y mayor o igual que (>=)

Cuando comparamos un string con estos operadores, se va comparando carácter a carácter y se compara su posición en la codificación Unicode para determinar si es menor (situado antes) o mayor (situado después).

A diferencia del operador de suma (+), cuando uno de los dos operandos es un número, el otro será transformado en número para comparar.

Para poder comparar como string, ambos operandos deben ser string.

```
console.log(6 >= 6); // true
console.log(3 < "5"); // true ("5" → 5)
console.log("adiós" < "bye"); // true
console.log("Bye" > "Adiós"); // true
console.log("Bye" > "adiós"); // false. Las letras mayúsculas van siempre antes
console.log("ad" < "adiós"); // true
```

Operadores booleanos

Los operadores booleanos son negación (!), y (&&), o (||).

Estos operadores, normalmente, son usados de forma combinada con los operadores relacionales formando una condición más compleja, la cual devuelve true o false.

```
console.log(!true); // Imprime false
console.log(!(5 < 3)); // Imprime true (!false)
console.log(4 < 5 && 4 < 2); // Imprime false (ambas condiciones deben ser ciertas)
console.log(4 < 5 || 4 < 2); // Imprime true (en cuanto una condición sea cierta, devuelve cierta y deja de comparar)
```

¿Preguntas?

Lenguaje de Marcas: Javascript

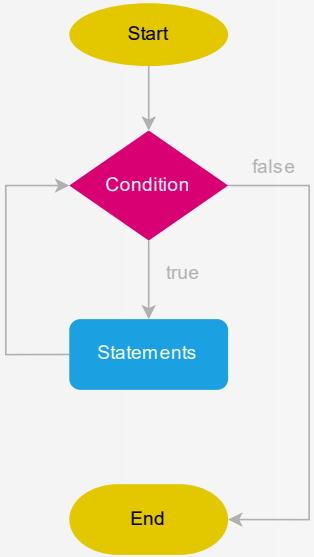
Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

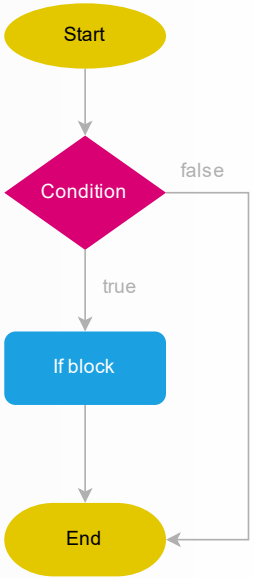
¿Qué son las estructuras de control?

•Son bloques de código que permiten tomar decisiones o repetir acciones según ciertas condiciones o reglas. En JavaScript, se dividen en:

• Estructuras de decisión (condicionales).



• Estructuras de repetición (bucles).



Estructuras de control

If:

Se usa para ejecutar código solo si una condición es verdadera.

```
let edad = 18;

if (edad >= 18) {
console.log("Eres mayor de edad.");
}
```

If...else

Se usa cuando necesitas ejecutar un bloque de código si la condición es verdadera y otro si es falsa.

```
let hora = 20;

if (hora < 12) {
console.log("Buenos días.");
} else {
console.log("Buenas tardes.");
}
```

if...else if...else

```
let nota = 85;

if (nota >= 90) {
console.log("Sobresaliente.");
} else if (nota >= 70) {
console.log("Aprobado.");
} else {
console.log("Suspendido.");
}
```

Estructuras de control

switch:

Ideal cuando hay varias condiciones posibles para una misma variable.

```
switch (expresión) {
case valor1:
// Código para el caso valor1
break;
case valor2:
// Código para el caso valor2
break;
default:
// Código si no se cumple ningún caso
}
```

```
let dia = 3;

switch (dia) {
case 1:
console.log("Lunes");
break;
case 2:
console.log("Martes");
break;
case 3:
console.log("Miércoles");
break;
default:
console.log("Día no válido.");
}
```

Estructuras de control

Estructuras de repetición (bucles)
Bucle for:
El bucle `for` es una estructura de control que se utiliza para repetir un bloque de código un número específico de veces.

- **Inicialización**: Se ejecuta una sola vez al principio del bucle y se utiliza para definir una variable de control, generalmente un contador.
- **Condición**: Se evalúa antes de cada iteración. Si la condición es verdadera, el bucle continúa. Si es falsa, el bucle se detiene.
- **Incremento/Decremento**: Se ejecuta después de cada iteración y se utiliza para modificar el valor del contador.

```
for (inicialización; condición; incremento/decremento) {
// Código a ejecutar
}

for (let i = 1; i <= 5; i++) {
console.log("Número:", i);
}
```

Estructuras de control

While:

Se usa cuando no sabes exactamente cuántas iteraciones necesitas, pero tienes una condición.

```
while (condición) {
// Código a ejecutar mientras la condición sea verdadera
}

let contador = 1;

while (contador <= 3) {
    console.log("Contador:", contador);
    contador++;
}
```

Estructuras de control

Do...while:

Es similar a while, pero garantiza que el bloque de código se **ejecute al menos una vez.**

```
let numero = 0;

do {
    console.log("Número:", numero);
    numero++;
} while (numero < 3);
```

Estructuras de control

Bucles con for...of y for...in

Es similar a while, pero garantiza que el bloque de código se **ejecute al menos una vez.**

for...of: Itera sobre elementos de arrays u objetos iterables.

```
let colores = ["rojo", "verde", "azul"];

for (let color of colores) {
    console.log(color);
}
```

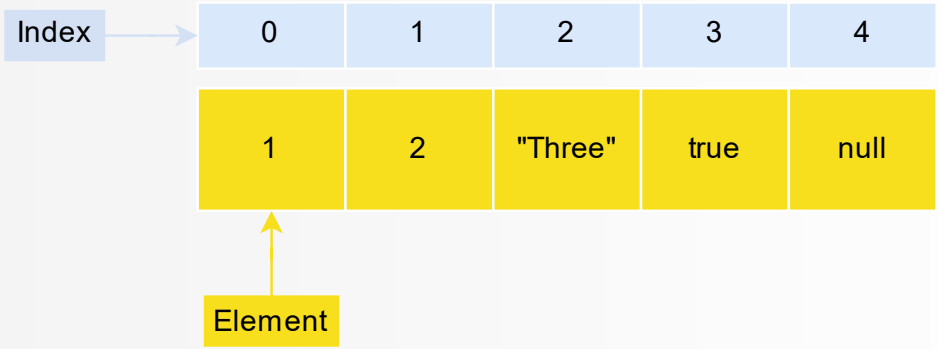
`for...in`: Itera sobre las propiedades de un objeto.

```
let persona = { nombre: "Juan", edad: 30 };

for (let propiedad in persona) {
    console.log(propiedad, ":",         persona[propiedad]);
}
```

Arrays en JS

- Un **array** es una colección ordenada de elementos.
- Los arrays pueden almacenar cualquier tipo de dato: **números**, **cadenas**, **objetos**, incluso otros arrays.




```
let frutas = ['Manzana', 'Banana', 'Naranja'];
console.log(frutas); // ["Manzana", "Banana", "Naranja"]
```

Arrays

Array literal (más común):

```
let frutas = ['Manzana', 'Banana', 'Naranja'];
console.log(frutas); // ["Manzana", "Banana", "Naranja"]
```

Constructor `Array`:

```
let numeros = new Array(1, 2, 3, 4);
console.log(numeros); // [1, 2, 3, 4]
```

Array vacío y luego agregar elementos:

```
let vacio = [];
vacio.push('Elemento1');
console.log(vacio); // ["Elemento1"]
```

Arrays

Arrays normales y asociativos

```
let numeros = [10, 20, 30];
console.log(numeros[0]); // 10
```

Array asociativo

```
const persona = {
nombre: "Juan",
edad: 30,
profesion: "Desarrollador",
pais: "España"
};

// Acceder a los valores
console.log(persona.nombre); // Output: Juan
console.log(persona.edad); // Output: 30
```

Arrays

Acceso y modificación de elementos

```
let frutas = ['Manzana', 'Banana', 'Naranja'];
console.log(frutas[0]); // "Manzana"

// Modificar un elemento
frutas[1] = 'Fresa';
console.log(frutas); // ["Manzana", "Fresa", "Naranja"]
```

Propiedad importante: `length`

```
console.log(frutas.length); // 3
```

Arrays

Recorrido de arrays :

Bucle for:
Se utiliza generalmente con **arrays** y otras estructuras de datos que tienen índices numéricos.

```
const numeros = [10, 20, 30, 40, 50];

for (let i = 0; i < numeros.length; i++) {
console.log(numeros[i]);
}

// Output:
// 10
// 20
// 30
// 40
// 50
```

Arrays

Recorrido de arrays :

forEach:

```
numeros.forEach((numero) => {
        console.log(numero);
});
```

Arrays - Métodos básicos de arrays

Agregar elementos:

`push` (al final):

```
let numeros = [1, 2, 3, 4, 5];
numeros.push(6);
console.log(numeros); // [1, 2, 3, 4, 5, 6]
```

`unshift` (al inicio):

```
let numeros = [1, 2, 3, 4, 5,6];
numeros.unshift(0);
console.log(numeros); // [0, 1, 2, 3, 4, 5, 6]
```

Eliminar elementos:

`pop` (elimina el último):

```
numeros.pop();
console.log(numeros); // [0, 1, 2, 3, 4, 5]
```

`shift` (elimina el primero):

```
numeros.shift();
console.log(numeros); // [1, 2, 3, 4, 5]
```

Arrays - Métodos básicos de arrays

Concatenar arrays:

```
let numeros = [1, 2, 3, 4, 5];
let masNumeros = [6, 7, 8];
let combinados = numeros.concat(masNumeros);
console.log(combinados); // [1, 2, 3, 4, 5, 6, 7, 8]
```

Verificar elementos:

```
let numeros = [1, 2, 3, 4, 5,6];
console.log(numeros.includes(3)); // true
console.log(numeros.includes(10)); // false
```

Arrays - Métodos avanzados

`map`: Crear un nuevo array aplicando una función

```
let numeros = [1, 2, 3, 4, 5];
let cuadrados = numeros.map((numero) => numero * numero);
console.log(cuadrados); // [1, 4, 9, 16, 25]
```

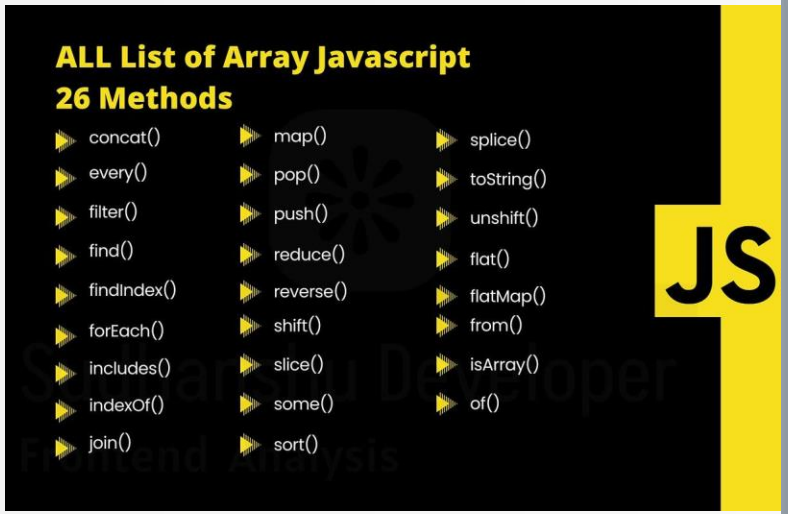
`filter`: Filtrar elementos según una condición

```
let numeros = [1, 2, 3, 4, 5,6];
let mayoresA3 = numeros.filter((numero) => numero > 3);
console.log(mayoresA3); // [4, 5]
```

`find`: Encuentra el primer elemento que cumple una condición

```
let encontrado = numeros.find((numero) => numero > 3);
console.log(encontrado); // 4
```

Arrays - Métodos avanzados




```
/  JavaScript Array Methods

[3, 4, 5, 6].at(1); // 4
[3, 4, 5, 6].pop(); // [3, 4, 5]
[3, 4, 5, 6].push(7); // [3, 4, 5, 6, 7]
[3, 4, 5, 6].fill(1); // [1, 1, 1, 1]
[3, 4, 5, 6].join("-"); // '3-4-5-6'
[3, 4, 5, 6].shift(); // [4, 5, 6]
[3, 4, 5, 6].reverse(); // [6, 5, 4, 3]
[3, 4, 5, 6].unshift(1); // [1, 3, 4, 5, 6]
[3, 4, 5, 6].includes(5); // true
[3, 4, 5, 6].map((num) => num + 6); // [9, 10, 11, 12]
[3, 4, 5, 6].find((num) => num > 4); // 5
[3, 4, 5, 6].filter((num) => num > 4); // [5, 6]
[3, 4, 5, 6].every((num) => num > 5); // false
[3, 4, 5, 6].findIndex((num) => num > 4); // 2
[3, 4, 5, 6].reduce((acc, num) => acc + num, 0); // 18
```

Arrays - Métodos avanzados

Arrays multidimensionales
Un array dentro de otro array.

```
let matriz = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
];
console.log(matriz[1][2]); // 6
```

Recorrido de una matriz:

```
matriz.forEach((fila) => {
    fila.forEach((columna) => {
        console.log(columna);
    });
});
```

¿Preguntas?

Lenguaje de Marcas: Javascript

Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

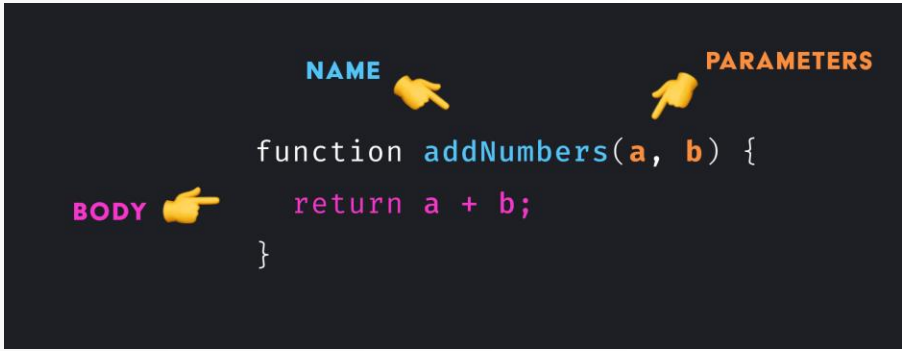
¿Qué es una función?

Una **función** es un bloque de código reutilizable diseñado para realizar una tarea específica.

Las funciones nos ayudan a **organizar, reutilizar y simplificar el código**.

Ventajas:

- Evita repetir código.
- Facilita el mantenimiento.
- Mejora la legibilidad.




```
function saludar() {
	console.log("Esta es una función básica en JavaScript");
}

//para ejecutarla simplemente la invocamos por su nombre:
saludar();
```

Funciones con Parámetros y Argumentos

Podemos **pasar valores a una función** a través de **parámetros**. Esto nos permite trabajar con datos dinámicos.

```
function saludar(nombre) {
    console.log("Hola, " + nombre + "!");
}

saludar("Carlos"); // Salida: Hola, Carlos!
saludar("Ana"); // Salida: Hola, Ana!
```

Funciones que Devuelven Valores

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

let resultado = sumar(5, 3);

console.log("Resultado:", resultado);
```

Funciones Expresadas (Funciones Anónimas)

Una **función expresada** es aquella que se almacena en una variable. A menudo, estas funciones no tienen nombre y se denominan **funciones anónimas**.

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

console.log(multiplicar(4, 5));
```

La función **no tiene nombre** dentro de function(...) { ... }.
Se almacena en la variable sumar, que ahora **funciona como el nombre de la función**.
Se llama usando sumar(3, 5), como si fuera una función con nombre.

Ventajas de las funciones anónimas

- Se pueden asignar a variables.
- Son útiles para operaciones **cortas y específicas**.

Arrow Functions (Funciones de Flecha)

Una **función expresada** es aquella que se almacena en una variable. A menudo, estas funciones no tienen nombre y se denominan **funciones anónimas**.

```
const multiplicar = (a, b) => a * b;

console.log(multiplicar(3, 5));
```

- **Se eliminan `{}` y `return`** porque la función **solo tiene una línea**.
- La flecha => indica el retorno automático del resultado.

Ventajas de las funciones anónimas

- Se pueden asignar a variables.
- Son útiles para operaciones **cortas y específicas**.
- Si solo tenemos un argumento, podemos obviar el paréntesis:

Funciones

Función anónima normal

```
const cuadrado = function(num) {
return num * num;
};
console.log(cuadrado(4));
```

Función Arrow

```
const cuadrado = num => num * num;
console.log(cuadrado(4));
```

Importante:

- **Funciones anónimas** son útiles en **callbacks** y cuando **no necesitas reutilizarlas**.
- **Funciones de flecha** son una alternativa más **concisa y moderna**.
- **Las funciones de flecha no tienen this**, por lo que no siempre son ideales.
- **Usa funciones normales** cuando necesites acceder a this en objetos o eventos.

¿Preguntas?

Lenguaje de Marcas: Javascript

Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

Scope en JavaScript

¿Qué es el Scope?
El **Scope** o **alcance** en JavaScript define dónde pueden accederse las variables dentro del código.

JavaScript tiene tres tipos principales de scope:

- **Global**
- **Local o de función**
- **De bloque**

Scope Global
Variables declaradas fuera de cualquier función o bloque pueden ser accedidas desde cualquier parte del programa.

```
var globalVar = "Soy una variable global";

function mostrarGlobal() {
        console.log(globalVar);
}
mostrarGlobal();
```

Scope en JavaScript

Scope de Función

Variables declaradas dentro de una función solo existen dentro de esa función.

```
function funcionScope() {
    let localVar = "Solo existo aquí";
    console.log(localVar);
}
funcionScope(); ❌ Error
```

Scope de Bloque

- Variables declaradas con `let` o `const` dentro de un bloque ({}) solo existen dentro de ese bloque.

```
if (true) {
    let blockVar = "Solo existo en este bloque";
    console.log(blockVar);
}
console.log(blockVar); ❌ Error
```

Objetos en JavaScript

¿Qué son los Objetos?

- Son estructuras de datos que agrupan **propiedades** y **métodos** en **una sola entidad.**
- Se representan con llaves **{}** y contienen pares **clave-valor**.

```
const persona = {
    nombre: "Juan",
    edad: 30,
    saludar: function() {
        console.log("Hola, soy " + this.nombre);
    }
};

console.log(persona.nombre); // "Juan"
persona.saludar(); // "Hola, soy Juan"
```

- Se acceden con **objeto.propiedad** o **objeto["propiedad"]**
- `this` dentro de un método **hace referencia al propio objeto.**

Desde **ES6**, podemos escribir métodos en un objeto de forma más corta:

```
const persona = {
    nombre: "Juan",
    edad: 30,
    saludar() { // Forma simplificada
        console.log(`Hola, soy ${this.nombre}`);
    }
};
```

Manipulación de Objetos

Agregar y Modificar Propiedades

```
persona.apellido = "Pérez";
persona.edad = 31;
console.log(persona);
```

Eliminar Propiedades

```
delete persona.edad;
console.log(persona);
```

Podemos modificar, agregar o eliminar propiedades dinámicamente.

Recorrer un Objeto

```
for (let clave in persona) {
    console.log(`${clave}: ${persona[clave]}`);
}
```

Manipulación de Objetos

Arrays de Objetos

```
const estudiantes = [
        { nombre: "Ana", edad: 20 },
        { nombre: "Luis", edad: 22 },
        { nombre: "Marta", edad: 19 }
];
```

map() - Transformar datos

```
const nombres = estudiantes.map(est => est.nombre);
console.log(nombres); // ["Ana", "Luis", "Marta"]
```

.filter() - Filtrar datos

```
const mayoresDe20 = estudiantes.filter(est => est.edad > 20);
console.log(mayoresDe20);
```

Manipulación de Objetos

```
const estudiantes = [
        { nombre: "Ana", edad: 20 },
        { nombre: "Luis", edad: 22 },
        { nombre: "Marta", edad: 19 }
];
```

reduce() - Acumular valores

```
const sumaEdades = estudiantes.reduce((acc, est) => acc + est.edad, 0);
console.log(sumaEdades); // 61
```

forEach() - Iterar sin retornar valor

```
estudiantes.forEach(est => console.log(est.nombre));
```

Uso del Spread Operator (. . .)

El Spread Operator (. . .) facilita copias y fusiones de arrays y objetos.

Copiar Arrays

```
const numeros = [1, 2, 3];
const copiaNumeros = [...numeros];
console.log(copiaNumeros); // [1, 2, 3]
```

Fusionar Objetos

```
const usuario = { nombre: "Ana", edad: 25 };
const datosAdicionales = { ciudad: "Madrid", ocupacion: "Desarrolladora" };
const usuarioCompleto = { ...usuario, ...datosAdicionales };
console.log(usuarioCompleto);
```

¿Preguntas?

Lenguaje de Marcas: Javascript

Iván Nieto Ruiz

il.nietoruiz@edu.gva.es

Referencia y copia en JavaScript

Tipos primitivos (copia por valor):

- Incluyen: `number`, `string`, `boolean`, `null`, `undefined`, `symbol`, `bigint`.
- Cuando se asigna una variable con un valor primitivo a otra, se hace una **copia del valor**.

Tipos de referencia (copia por referencia):

- Incluyen: `object`, `array`, `function`.
- Cuando se asigna una variable con un objeto a otra, **ambas apuntan al mismo objeto en memoria**.

Ejemplo de **Copia por Valor** (Primitivos)

```
let a = 10;
let b = a; // Se copia el valor de 'a' en 'b'

b = 20; // Modificamos 'b', pero 'a' no cambia

console.log(a); // 10
console.log(b); // 20
```

Referencia y copia en JavaScript

Ejemplo de **Copia por Referencia** (Objetos y Arrays) - **comparten la misma referencia** en memoria

```
let obj1 = { name: "Juan" };
let obj2 = obj1; // Ambas variables apuntan al mismo objeto en memoria

obj2.name = "Carlos"; // Modificamos 'obj2'

console.log(obj1.name); // "Carlos" (también cambió en 'obj1')
console.log(obj2.name); // "Carlos"
```

Copiar objetos sin modificar el original

Copia superficial (shallow copy)

```
let obj1 = { name: "Juan", age: 30 };

// Usando spread operator (...):
let obj2 = { ...obj1 };
obj2.name = "Carlos";

console.log(obj1.name); // "Juan" (no se modifica)
console.log(obj2.name); // "Carlos"

// Usando Object.assign():
let obj2 = Object.assign({}, obj1);
```

Referencia y copia en JavaScript

Problema: Si el objeto tiene **objetos anidados**, estos seguirán compartiendo la referencia.

```
let obj1 = { name: "Juan", address: { city: "Madrid" } };
let obj2 = { ...obj1 };

obj2.address.city = "Barcelona";

console.log(obj1.address.city); // "Barcelona" CUIDADO (¡Se modificó el original!)
```

Si hay objetos anidados, usa **structuredClone()** o **JSON.parse(JSON.stringify(obj))**.

```
let obj1 = { name: "Juan", address: { city: "Madrid" } };
let obj2 = structuredClone(obj1); // ¡Ahora sí es independiente!

obj2.address.city = "Barcelona";

console.log(obj1.address.city); // "Madrid" - OK
console.log(obj2.address.city); // "Barcelona"
```

structuredClone() es nativo en JS moderno, mientras que JSON.parse(JSON.stringify(obj)) funciona pero **pierde funciones y símbolos**.

Referencia y copia en JavaScript

Lo mismo ocurre con los **arrays**

```
let arr1 = [1, 2, 3];
let arr2 = arr1; // Copia por referencia

arr2.push(4);

console.log(arr1); // [1, 2, 3, 4] ❌ (¡Se modificó el original!)
console.log(arr2); // [1, 2, 3, 4]
```

Forma correcta de copiar un array (shallow copy):

```
let arr2 = [...arr1]; // Operador spread
// o
let arr2 = arr1.slice(); // Método slice()
```

Copia profunda de arrays anidados:

```
let arr1 = [[1, 2], [3, 4]];
let arr2 = structuredClone(arr1);

arr2[0][0] = 99;

console.log(arr1[0][0]); // 1 --> OK
console.log(arr2[0][0]); // 99
```

Ejemplos

Ejercicio 1:

```
let a = 5;
let b = a;

b = 10;

console.log(a); // ¿?
console.log(b); // ¿?
```

¿Por qué **a** no cambia cuando modificamos b?

Ejemplos

Ejercicio 1:

```
let persona1 = {
        nombre: "Lucas",
        direccion: { ciudad: "Madrid", pais: "España" }
};

let persona2 = { ...persona1 };

persona2.direccion.ciudad = "Barcelona";

console.log(persona1.direccion.ciudad); // ¿?
console.log(persona2.direccion.ciudad); // ¿?
```

Ejemplos

Ejercicio 3:

```
function deepCopy(obj) {
        return structuredClone(obj);
}

let user1 = { nombre: "Sofía", datos: { edad: 28, ciudad: "Valencia" } };
let user2 = deepCopy(user1);

user2.datos.ciudad = "Sevilla";

console.log(user1.datos.ciudad); // ¿?
console.log(user2.datos.ciudad); // ¿?
```

¿Preguntas?