

UT 2

JAVASCRIPT

Componentes de una web

HTML



Contenido e
información

CSS



Estilos e interfaz
agradable

JS



Funcionalidad

¿Qué es JavaScript?

Javascript es un **lenguaje de programación**, o lo que es lo mismo, un mecanismo con el que podemos decirle a nuestro navegador que tareas debe realizar, en qué orden y cuantas veces

¿Cuántas líneas ocupa un código para mostrar los números del 1 al 500 en...

...HTML?

500 líneas

...JavaScript?

<10 líneas

Diferencias entre Java y JavaScript

Java	JavaScript
Lenguaje de programación orientado a objetos	Lenguaje de scripting orientado a objetos basados en prototipos
Usado en aplicaciones de escritorio, servidores, aplicaciones móviles	Desarrollo web en el navegador
Tipado fuerte	Tipado dinámico
Ejecutado en la JVM	Ejecutado en navegadores web

¿Por qué ese nombre tan parecido?

1995 Brendan Eich crea un lenguaje de script en Netscape. Se llamaba Mocha

1995 Para aprovechar la fama de Java, se renombra a JavaScript

Se incluyen mejoras bajo el estándar ECMAScript hasta la actualidad



1995 Se renombra a LiveScript

1997 Se estandariza bajo el nombre de ECMAScript

LA CONSOLA

La consola de JavaScript

La consola Javascript es una zona del navegador ubicada en las DevTools donde podemos escribir pequeños fragmentos de código y observar los resultados, así como revisar mensajes de información, error u otros detalles similares.



DevTools - Herramientas
de depuración de código



Consola - Ejecución de
código en tiempo real

El método `console.log`

Para mostrar un texto en la consola

```
console.log("Hola Mundo");
console.log(2 + 2);
VM323:1 Hola Mundo
VM323:2 4
```

El método `console.log()`

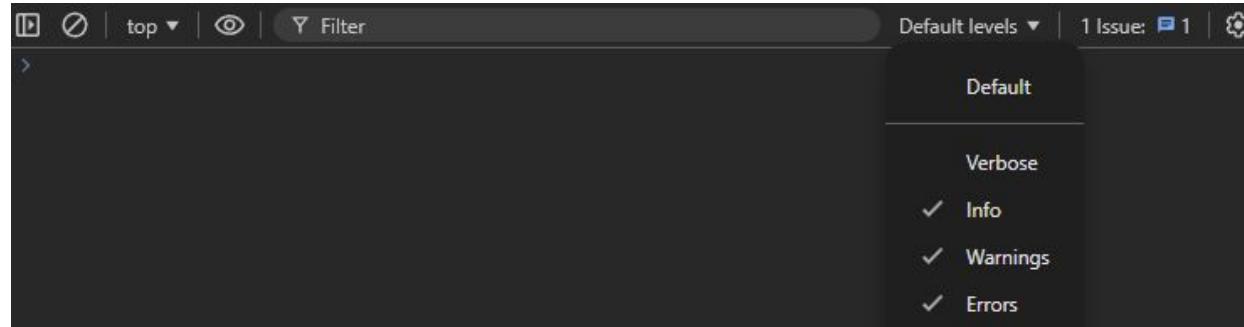
También funciona con varios parámetros



```
console.log("¡Hola a todos! Observen este número: ", 5 + 18);  
VM345:1 ¡Hola a todos! Observen este número: 23
```

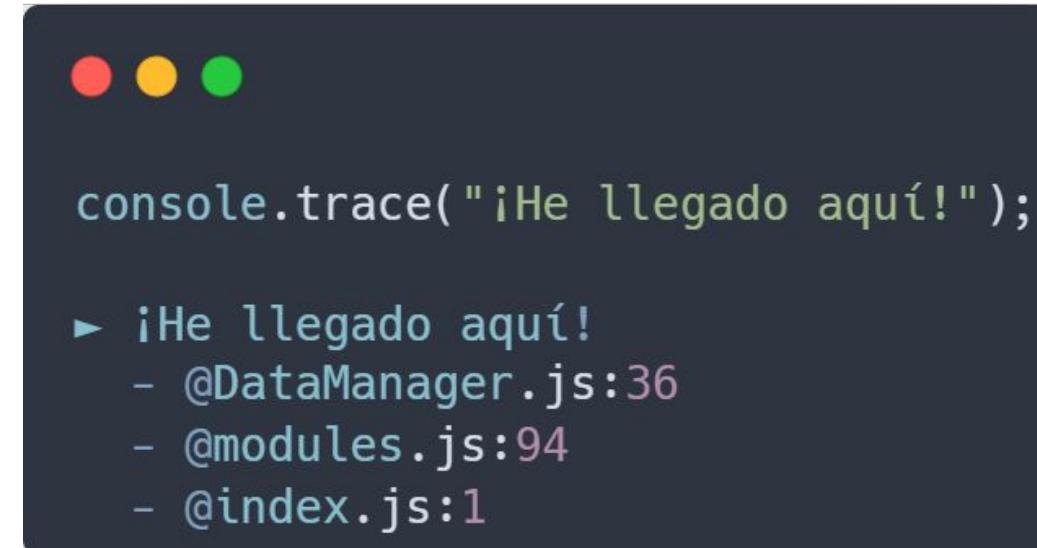
Mensajes de error

Función	Descripción
<code>console.debug()</code>	Muestra información con todo detalle
<code>console.info()</code>	Muestra mensajes informativos
<code>console.warn()</code>	Muestra advertencias. Se destaca en amarillo
<code>console.error()</code>	Muestra errores. Se destaca en rojo



Mensaje con traza del error

Permite mostrar el camino que ha seguido el programa para llegar a un punto o error concreto.



A screenshot of a terminal window with a dark background. At the top left, there are three colored dots: red, yellow, and green. Below them, the following text is displayed:

```
console.trace("¡He llegado aquí!");
```

▶ ¡He llegado aquí!

- @DataManager.js:36
- @modules.js:94
- @index.js:1

Limpiar la consola

Para borrar el texto que haya en la consola se puede usar la función:



O el atajo de teclado:



Agrupar mensajes

Función	Descripción
<code>console.group()</code>	Inicia una sección con el texto indicado por parámetro
<code>console.groupCollapsed()</code>	Como el anterior, pero está colapsado
<code>console.groupEnd()</code>	Finaliza la sección de elementos agrupados

```
> console.group("Información a mostrar");
  console.log("UA: ", navigator.userAgent);
  console.log("Lang: ", navigator.language);
  console.groupEnd();

  ▼ Información a mostrar VM462:1
    UA: Mozilla/5.0 (Windows NT 10.0; Win64; VM462:2
      x64) AppleWebKit/537.36 (KHTML, like Gecko)
      Chrome/131.0.0.0 Safari/537.36

    Lang: es-ES VM462:3
```

El método `console.table()`

En ocasiones, cuando queremos mostrar información más compleja, como en el caso de objetos o arrays, se puede utilizar el método `console.table()`, indicando por parámetro el objeto o array a dibujar

```
> const comidas = [
    { name: "tortilla", ingrediente: "patata" },
    { name: "tortilla francesa", ingrediente: "huevo" },
    { name: "cocido", ingrediente: "garbanzo" }
];

console.table(comidas);
```

VM193:7

(index)	name	ingrediente
0	'tortilla'	'patata'
1	'tortilla francesa'	'huevo'
2	'cocido'	'garbanzo'

▶ Array(3)

ORGANIZACIÓN

¿Dónde se escribe el código JS?

Script en Línea

- En el mismo archivo html
- No recomendado

Script Externo

- En un fichero .js
- Recomendado

Script en línea

```
<html>
  <head>
    <title>Título de la página</title>
    <script>
      console.log("¡Hola!");
    </script>
  </head>
  <body>
    <p>Ejemplo de texto.</p>
  </body>
</html>
```

Script externo

```
<html>
  <head>
    <title>Título de la página</title>
    <script src="js/index.js"></script>
  </head>
  <body>
    <p>Ejemplo de texto.</p>
  </body>
</html>
```

¿Dónde incluimos etiquetas de script?

Dónde	Qué ocurre
En la etiqueta <head>	Actúa antes de que la página comience a dibujarse
En la etiqueta <body>	Se ejecuta durante el dibujado de la página
Antes de </body>	Se ejecuta cuando la página ya está dibujada

VARIABLES

Formas de declarar variables

var

Declara una variable, opcionalmente la inicia a un valor.

let

Declara una variable local con ámbito de bloque, opcionalmente la inicia a un valor.

const

Declara un nombre de constante de solo lectura y ámbito de bloque.

Tipos de datos en JS

Tipos de datos primitivos

- Numéricos
- Texto
- Booleanos

Tipos de datos no primitivos

- Objetos
- Funciones

Tipos de datos primitivos

Tipo de dato	Descripción
number	Valor numérico
bigint	Valor numérico muy grande
string	Valor de texto (cadenas, caracteres)
boolean	Verdadero o falso
symbol	Símbolo (valor único)
undefined	Valor sin definir (variable sin inicializar)

Tipos de datos primitivos

```
// Un texto, letra o carácter  
const text = "Hola mundo";  
  
// Un número (entero o decimal)  
const number = 42;  
  
// Un número muy grande (se añade n al final)  
const bignumber = 12345678901234567890n;  
  
// Un valor de verdadero o falso  
const boolean = true;
```

Valores undefined y null

Existe un tipo de dato especial denominado **undefined** (sin definir). Este es el valor que tienen las variables a las que no se les ha dado ningún valor específico

El valor especial **null** indica la ausencia intencional de información. A diferencia de undefined null indica que el valor ha sido definido explícitamente, pero representa una ausencia de valor.

El operador `typeof`

La función `typeof` nos devuelve el tipo de dato primitivo de la variable que le pasemos por parámetro.

```
const text = "Hola mundo!";
typeof text;      // Devuelve "String"

const number = 42;
typeof number;    // Devuelve "Number"

const boolean = true;
typeof boolean;   // Devuelve "Boolean"

let notDefined;
typeof notDefined; // Devuelve undefined
```

El método `prompt`

Este método se utiliza para pedir información por teclado. Puede incluir parámetros de manera opcional. El primero corresponde con el texto que aparecerá en la ventana mientras que el segundo será un placeholder.



```
let persona = prompt("Introduce tu nombre", "Nombre");
```

Conversión de datos a text

La mayoría de las veces, una conversión a cadena de texto se hace de manera implícita, sin embargo, también podemos llamar a la función `String(valor)` para convertir un valor a string:



```
let value = true;
alert(typeof value); // boolean

value = String(value); // ahora value es el string "true"
alert(typeof value); // string
```

Conversión implícita a number

De manera implícita, se puede realizar la conversión al incluirlos en una expresión matemática, aunque hay que tener cuidado con la operación de concatenación.

```
console.log("5" - 3); // 2 ("5" se convierte en 5)
console.log("10" * 2); // 20 ("10" se convierte en 10)
console.log("20" / 4); // 5 ("20" se convierte en 20)
console.log("5" + 3); // "53" (concatenación)
console.log(true + 2); // 3
console.log(false * 5); // 0
console.log(null + 5); // 5
console.log(undefined + 1); // NaN
```

Conversión explícita a number usando `Number()`

De manera explícita, pueden usarse algunas funciones para minimizar errores inesperados.

```
● ● ●

// Convertir cadenas a números
console.log(Number("123"));          // 123
console.log(Number("123.45"));        // 123.45
console.log(Number(" 123  ")); // 123 (ignora espacios al inicio y al final)

// Cadenas no numéricas
console.log(Number("abc"));           // NaN
console.log(Number("123abc"));         // NaN

// Valores booleanos
console.log(Number(true));            // 1
console.log(Number(false));           // 0
```

Conversión a number con `Number.parseInt()`

`Number.parseInt()` convierte cadenas a enteros, deteniéndose en el primer carácter no numérico.



```
// Cadenas numéricas
console.log(Number.parseInt("123"));          // 123
console.log(Number.parseInt("123.45"));        // 123 (ignora la parte decimal)
console.log(Number.parseInt(" 123  "));         // 123

// Cadenas con texto adicional
console.log(Number.parseInt("123abc"));        // 123
console.log(Number.parseInt("abc123"));         // NaN (empieza con texto no numérico)
```

Conversión a number con `Number.parseFloat()`

`Number.parseFloat()` convierte cadenas a números de punto flotante, permitiendo decimales.

```
// Cadenas numéricas
console.log(Number.parseFloat("123.45"));           // 123.45
console.log(Number.parseFloat("123.45abc"));        // 123.45 (ignora texto después del
número)
console.log(Number.parseFloat(" 123.45  ")); // 123.45

// Valores enteros
console.log(Number.parseFloat("123"));             // 123 (convierte a float aunque sea
un entero)

// Cadenas no numéricas
console.log(Number.parseFloat("abc123.45"));       // NaN
console.log(Number.parseFloat "");                  // NaN

// Notación científica
console.log(Number.parseFloat("1e3"));            // 1000 (1 * 103)
console.log(Number.parseFloat("-1.5e-2"));         // -0.015 (-1.5 * 10-2)
```

Resumen de comportamiento en la conversión

Valor	<code>Number()</code>	<code>Number.parseInt()</code>	<code>Number.parseFloat()</code>
“123”	123	123	123
“123abc”	Nan	123	123
“123.45”	123.45	123	123.45
“abc123”	Nan	Nan	Nan
true	1	Nan	Nan
null	0	Nan	Nan
undefined	Nan	Nan	Nan
“1e3”	1000	1	1000

Conversión de datos a boolean

La conversión a boolean es la más simple.

Las reglas de conversión:

- Cualquier valor `false`, obviamente, se considera `false`.
- Cualquier valor que sea `0`, incluyendo `0.0` o `0n` (`bigint`).
- Cualquier valor que sea una cadena vacía ("").
- Los valores especiales `null`, `undefined` y `NaN` también se consideran `false`.
- El resto de valores, se consideran `truthy`.

Valores falsy



```
console.log(Boolean(false));          // false
console.log(Boolean(0));             // false
console.log(Boolean(-0));            // false
console.log(Boolean(0n));            // false
console.log(Boolean(""));            // false
console.log(Boolean(null));          // false
console.log(Boolean(undefined));      // false
console.log(Boolean(NaN));           // false
```

Valores truthy



```
console.log(Boolean(true));          // true
console.log(Boolean(1));            // true
console.log(Boolean(-1));           // true
console.log(Boolean(123));          // true
console.log(Boolean("hello"));       // true (cadena no vacía)
console.log(Boolean("false"));       // true (cadena no vacía)
console.log(Boolean([]));           // true (arreglo vacío)
console.log(Boolean({}));           // true (objeto vacío)
console.log(Boolean(function(){})); // true (función)
```

STRING

Creación de un string

Los strings son uno de los tipos de datos básicos (primitivos), y como tal, es más sencillo utilizar los literales que la notación que utiliza la palabra clave new. Para englobar los textos, se pueden utilizar tres tipos de comillas:

- Comillas simples: '
- Comillas dobles: "
- Backticks: `



```
// Notación literal (preferida)
const text = `¡Hola a todos!`;
const message = "Otro mensaje de texto";

// Notación mediante objeto
const text = new String("¡Hola a todos!");
const message = new String("Otro mensaje de texto");
```

Tamaño de un string

.length, que se encarga de devolver el tamaño total de una cadena de texto.



```
"Hola".length;      // 4
"Adiós".length;   // 5
"".length;         // 0
"¡Yeah!".length;  // 6
```

Acceso a los caracteres

Si queremos acceder a cada uno de los caracteres de un string, podemos utilizar el operador [] indicando la posición como si se tratase de un array.

```
const text = "Hola";  
  
text[0];      // "H"  
text[1];      // "o"  
text[2];      // "l"  
text[4];      // undefined
```

Interpolación de variables

Las backticks (comillas hacia atrás), que nos permitirán interpolar el valor de las variables sin tener que cerrar, concatenar y abrir la cadena de texto continuamente.



```
const firstWord = "frase";
const secondWord = "concatenada";

"Una " + firstWord + " bien " + secondWord;      // 'Una frase bien concatenada'

`Una ${firstWord} mejor ${secondWord}`;           // 'Una frase mejor concatenada'
```

Algunos métodos de string

Método	Descripción
charAt(index)	Devuelve el carácter en la posición indicada.
includes(substr)	Verifica si el string contiene el substring especificado.
indexOf(substr)	Devuelve el índice de la primera aparición del substring, o -1.
replace(old, new)	Reemplaza el substring por uno nuevo
slice(start, end)	Extrae una parte del string desde el índice start hasta end.
split(separator)	Divide el string en un array de subcadenas usando un separador.
match(regex)	Devuelve las coincidencias del string con la expresión regular.
trim()	Elimina los espacios en blanco al principio y al final.

OPERADORES

Operadores aritméticos

Son los operadores que utilizamos para realizar operaciones matemáticas básicas.

Nombre	Operador	Descripción
Suma	$a+b$	Suma el valor de a al valor de b
Resta	$a-b$	Resta el valor de b al valor de a
Multiplicación	$a*b$	Multiplica el valor de a por el valor de b
División	a/b	Divide el valor de a entre el valor de b
Módulo	$a \% b$	Devuelve el resto de la división a entre b
Exponente	$a^{**}b$	Eleva a a la potencia de b , es decir a^b

Operadores de asignación

Permiten asignar información a diferentes variables a través del símbolo =.

Nombre	Operador	Equivalencia
Asignación	$a=b$	Asigna el valor de b en la variable a
Suma y asignación	$a+=b$	$a=a+b$
Resta y asignación	$a-=b$	$a=a-b$
Multiplicación y asignación	$a*=b$	$a=a*b$
División y asignación	$a/=b$	$a=a/b$
Módulo y asignación	$a\%=b$	$a=a\%b$
Exponente y asignación	$a**=b$	$a=a**b$

Operadores unarios

Son aquellos que en lugar de tener dos operandos, como los mencionados anteriormente, sólo tienen uno.

Nombre	Operador	Descripción
Incremento	a++	Postincremento. Usa a y lo incrementa
Decremento	a--	Postdecremento. Usa a y lo decrementa
Preincremento	++a	Incrementa a y lo usa
Predecremento	--a	Decrementa a y lo usa
Negación	-a	Cambia el signo de a

Preincremento y postincremento



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

// Postincremento
let post = a++; // Asigna el valor actual de `a` a `post`, luego incrementa `a`

// Preincremento
let pre = ++b; // Incrementa el valor de `b`, luego asigna el valor incrementado a `pre`

console.log(post); // 5 (el valor actual de `a` antes de incrementar)
console.log(a);    // 6 (incremento aplicado después)

console.log(pre); // 6 (el valor de `b` después del incremento)
console.log(b);    // 6 (incremento aplicado antes)
```

Operadores de comparación

Nombre	Operador	Descripción
Igualdad	<code>a==b</code>	Comprueba si el valor de a es igual al valor de b
Desigualdad	<code>a!=b</code>	Comprueba si el valor de a es distinto al de b
Mayor que	<code>a>b</code>	Comprueba si el valor de a es mayor que el de b
Menor o igual	<code>a>=b</code>	Comprueba si el valor de a es mayor o igual que b
Menor que	<code>a<b</code>	Comprueba si el valor de a es menor que el de b
Menor o igual	<code>a<=b</code>	Comprueba si el valor de a es menor o igual que b
Identidad	<code>a====b</code>	Comprueba si el valor de a y su tipo de dato es igual a b
No identidad	<code>a!==b</code>	Comprueba si el valor de a y su tipo de dato es distinto a b

Igualdad e identidad

En Javascript no es lo mismo utilizar == (igualdad) que === (identidad). Mientras que el primero sólo comprueba el valor de la comparación, el segundo, el operador de identidad comprueba el valor y el tipo de dato de la comparación.



```
5 == 5      // true    (ambos son iguales, coincide su valor: 5)
"5" == 5    // true    (ambos son iguales, coincide su valor: 5)
5 === 5     // true    (ambos son idénticos, coincide valor y tipo de dato)
"5" === 5   // false   (no son idénticos, coincide valor pero no tipo de dato)
```

Operadores lógicos

Se utilizan para realizar operaciones lógicas

Operador	Símbolo	Descripción
AND	&&	Devuelve true si ambos valores son true
OR		Devuelve true si algún valor es true
NOT	!	Devuelve el valor opuesto

Operador AND



```
false && false      // false (si ninguno de los dos es true, false)
true && false       // false (idem)
false && true        // false (idem)
true && true         // true (si ambos son true, true)
```

Operador AND con otros tipos de datos

El operador lógico AND funciona de una forma que se llama cortocircuito lógico, donde:

- ① Si el primer valor se evalúa como verdadero, devuelve el segundo valor.
- ② Si el primer valor se evalúa como falso, devuelve ese primer valor.
- ✓ O lo que es lo mismo: Si A es verdadero, entonces B.



```
0 && undefined      // 0          (-> false && false, devuelve el primero)
undefined && 0        // undefined (-> false && false, devuelve el primero)
55 && null           // null       (-> true && false, devuelve el segundo)
null && 55             // null       (-> false && true,  devuelve el primero)
44 && 20              // 20         (-> true && true,  devuelve el segundo)
```

Operador AND ejemplo de uso



```
45 && "OK"          // "OK"
false && "OK"         // false

const doTask = () => "OK!";    // Creamos función que devuelve "OK!"
isCorrect && doTask()        // Si isCorrect es true, ejecuta doTask()
isCorrect && (a = 55);      // Si isCorrect es true, asigna a la variable "a" el 5
```

Operador OR



```
true || false;      // true
false || true;      // true
true || true;       // true
false || false;     // false
```

Operador OR con otros tipos de datos

Por otro lado, el operador lógico OR funciona de la siguiente manera:

- ① Si el primer valor se evalúa como verdadero, devuelve el primer valor.
- ② Si el primer valor se evalúa como falso, devuelve el segundo valor.
- ✓ O lo que es lo mismo: Si A es verdadero, A, sino B.



```
0 || false          // false (se evalua como false || false, devuelve el segundo)
0 || null           // null (se evalua como false || false, devuelve el segundo)
44 || undefined    // 44 (se evalua como true || false, devuelve el primero)
0 || 17              // 17 (se evalua como false || true, devuelve el segundo)
4 || 10              // 4 (se evalua como true || true, devuelve el primero)
```

Operador OR ejemplo de uso



```
const userName = name || "Unknown name";  
  
"José" || "Unknown name"      // "José"  
null || "Unknown name"        // "Unknown name"  
false || "Unknown name"       // "Unknown name"  
undefined || "Unknown name"   // "Unknown name"  
0 || "Unknown name"          // "Unknown name" (el 0 se evalua como falso)
```

Asignación lógica



```
let userName = "";

if (!userName) {
  userName = "José";
}

let userName = "";          // o null, false o undefined

userName ||= "José"        // Ahora es "José"
userName ||= "Paco"         // Sigue siendo "José"
userName ||= false          // Sigue siendo "José"
```

Operador NOT

Por último, el operador lógico NOT es un operador unario muy frecuente y útil, que se utiliza para negar un valor, es decir, para invertir su valor booleano.

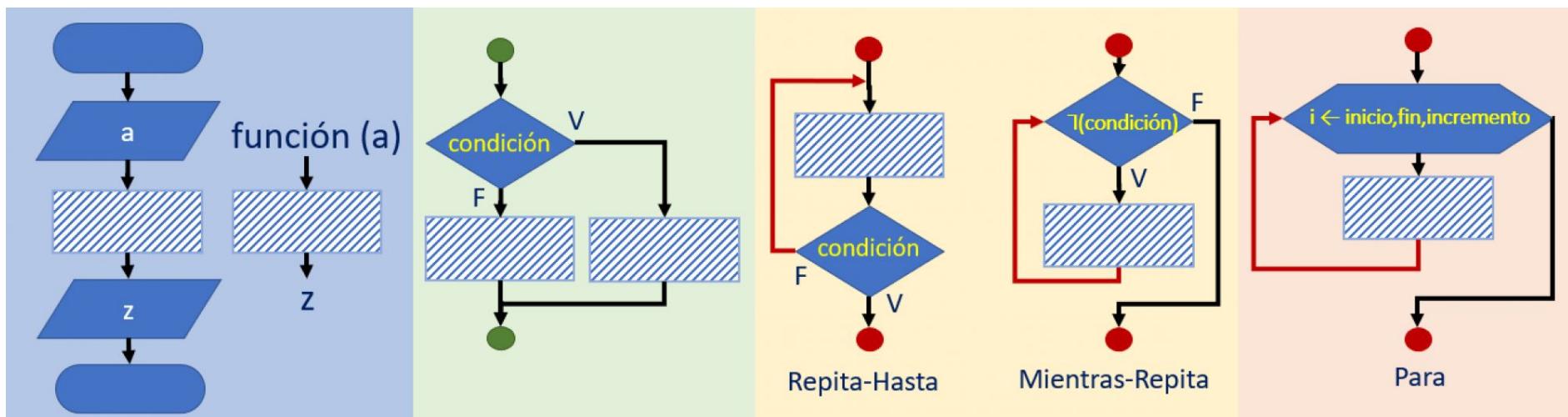


```
!true      // false
!false     // true
!!true     // true
!!!true    // false
!44        // false (se evalua como !true)
!0         // true (se evalua como !false)
!""
!(10 || 23) // false (se evalua como !10, que es !true)
```

ESTRUCTURAS DE CONTROL

Estructuras de control

Para modificar el flujo de control del programa



Estructura condicional if



```
let numero = 5;

if (numero > 0) {
    console.log("El número es positivo");
}
```

Estructura condicional else



```
let numero = -1;

if (numero > 0) {
    console.log("El número es positivo");
} else {
    console.log("El número no es positivo");
}
```

Estructura condicional else if

```
let numero = 0;

if (numero > 0) {
    console.log("El número es positivo");
} else if (numero < 0) {
    console.log("El número es negativo");
} else {
    console.log("El número es cero");
}
```

Estructura iterativa for

- Inicialización: Se crea y asigna un valor inicial a la variable de control. Por lo general, esta variable es un número que actúa como índice o contador.
- Condición: Si es verdadera, el bloque de código se ejecuta. Si es falsa, el bucle termina y el flujo de control del programa continúa después del bucle.
- Actualización: Actualiza el valor de la variable de control después de cada iteración.



```
for (inicialización; condición; actualización) {  
    // bloque de código a ejecutar  
}
```

Estructura iterativa for



```
let suma = 0;  
  
for (let i = 1; i <= 10; i++) {  
    suma += i;  
}  
  
console.log("La suma de los números del 1 al 10 es:", suma)
```

Estructura iterativa while



```
let numero = 5;
let factorial = 1;

while (numero > 1) {
    factorial *= numero;
    numero--;
}

console.log(factorial);
```

Estructura iterativa do-while

```
let numero;

do {
    numero = parseInt(prompt("Ingrese
    un número mayor que 10:"));
} while (numero <= 10);

console.log("El número ingresado
es:", numero);
```

ÁMBITO

¿Qué es el ámbito en JavaScript?

El ámbito determina dónde una variable es accesible en tu código.

Tipos de ámbito:

- ① Global: Disponible en todo el programa.
- ② Local: Disponible sólo dentro de una función o bloque.

Ámbito de Bloque (let y const)

Las variables declaradas con **let** y **const** están limitadas al bloque {} donde se declaran.

```
{  
  let x = 10;  
  console.log(x); // 10  
}  
console.log(x); // Error: x no está definida
```

Ámbito de Función (var)

Variables declaradas con **var** tienen ámbito de función, ignorando bloques.

```
function test() {  
    var y = 20;  
    if (true) {  
        var y = 30; // Mismo ámbito  
        console.log(y); // 30  
    }  
    console.log(y); // 30  
}  
test();
```

Hoisting con var vs let

Hoisting es un comportamiento de JS en el que las declaraciones de variables, funciones y clases son "elevadas" al inicio de su ámbito durante la fase de compilación. Esto significa que puedes usarlas antes de que se declaren en el código, aunque el comportamiento puede variar según cómo se declaren.

```
● ● ●

console.log(a); // undefined
var a = 10;

console.log(b); // Error: b no está definida
let b = 20;
```

FUNCIONES

Funciones

Las funciones nos permiten agrupar líneas de código en tareas con un nombre, para que, posteriormente, podamos hacer referencia a ese nombre para realizar todo lo que se agrupe en dicha tarea.

En JavaScript, las funciones son un tipo de dato.



```
typeof function () {}; // 'function'
```

Formas de crear funciones - por declaración

Una función en JS puede declararse como en cualquier otro lenguaje de programación.

```
function saludar() {  
    return "Hola";  
}  
  
saludar();      // 'Hola'  
typeof saludar; // 'function'
```

Formas de crear funciones - por expresión

Sin embargo, es común guardar funciones dentro de variables, para luego ejecutar dichas variables.

Observa como la llamada a la función se realiza mediante la variable, pues el nombre anterior desaparece.

```
const saludo = function saludar() {  
    return "Hola";  
};  
  
saludo(); // 'Hola'
```

Funciones anónimas

Son un tipo de funciones que se declaran sin definir un nombre de función, alojándolas en el interior de una variable y haciendo referencia a dicha variable cada vez que queramos utilizarla.

```
// Función anónima "saludo"
const saludo = function () {
    return "Hola";
};

saludo(); // 'Hola'
saludo; // f () { return 'Hola'; }
```

Callback

Un callback (llamada hacia atrás) es pasar una función por parámetro a otra función, de modo que esta última función puede ejecutar la función pasada por parámetro de forma genérica desde su propio código, y nosotros podemos definirlas desde fuera de dicha función.

```
const action = function () {
    console.log("Acción ejecutada.");
};

const mainFunction = function (callback) {
    callback();
};

mainFunction(action);
```

Función de orden superior (HOF)

Se trata de funciones que aceptan funciones por parámetro y/o devuelven funciones en el return.

```
function crearMultiplicador(factor) {  
    return function (numero) {  
        return numero * factor;  
    };  
}  
  
const duplicar = crearMultiplicador(2); // HOF  
devuelve una nueva función  
console.log(duplicar(5)); // 10
```

Función de orden superior (HOF)

Podemos ejecutar la función doTask(), que es nuestra HOF, cambiando los callbacks según nos interese, sin necesidad de repetir código.

```
● ● ●

const action = function () {
  console.log("Acción ejecutada.");
};

const error = function () {
  console.error("Ha ocurrido un error");
};

const doTask = function (callback, callbackError) {
  const isError = Math.random() < 0.5;

  if (!isError) callback();
  else callbackError();
};

doTask(action, error);
```

Función de orden superior (HOF)

Con callbacks pequeños, es común utilizar funciones anónimas.

```
const doTask = function (callback, callbackError) {
    const isError = Math.random() < 0.5;

    if (!isError) callback();
    else callbackError();
};

doTask(function () {
    console.log("Acción ejecutada.");
},
function () {
    console.error("Ha ocurrido un error");
}
);
```

Funciones flecha

Las Arrow functions, funciones flecha o «fat arrow» son una forma corta y compacta de escribir las funciones tradicionales de Javascript. A grandes rasgos, se trata de eliminar la palabra function y añadir el texto => antes de abrir las llaves:

```
const func = function () {
    return "Función tradicional.";
};

const func = () => {
    return "Función flecha.";
};
```

Atajos con las funciones flecha

- 1 Si el cuerpo de la función sólo tiene una línea, podemos omitir las llaves ({}).
- 2 En ese caso, se hace un return implícito, por lo que podemos omitir también el return.
- 3 Si la función no tiene parámetros, se indica como en el ejemplo anterior: () =>.
- 4 Si la función tiene un solo parámetro, opcionalmente te puedes ahorrar los paréntesis: e =>.
- 5 Si la función tiene 2 ó más parámetros, se indican entre paréntesis: (a, b) =>.

```
// 0 parámetros: Devuelve "Función flecha"
const func = () => "Función flecha.";

// 1 parámetro: Devuelve el valor de e + 1
const func = (e) => e + 1;

// 2 parámetros: Devuelve el valor de a + b
const func = (a, b) => a + b;
```

Funciones autoejecutables (IIFE)

Se trata de funciones que se declaran y se ejecutan inmediatamente. Para ello se envuelve la función entre paréntesis y se añade un par de paréntesis detrás para los parámetros.

```
(function () {
    console.log("Hola!!");
})();
```

Funciones autoejecutables (IIFE)

Los parámetros se pueden añadir en el segundo par de paréntesis.



```
(function (name) {  
    console.log(`¡Hola, ${name}!`);  
})( "Julia" );
```

Funciones autoejecutables (IIFE)

Si hacemos una asignación de una función autoejecutable, no almacenamos la función, si no el valor del return.

```
const value = (function (name) {  
    return `¡Hola, ${name}!`;  
})( "Julia" );  
  
value; // '¡Hola, Julia!'  
typeof value; // 'string'
```

Clausuras

Una clausura o cierre se define como una función que «encierra» variables en su propio ámbito (y que continúan existiendo aún habiendo terminado de ejecutar la función).

```
const incr = (function () {
  let num = 0;
  return function () {
    num++;
    return num;
  };
})();
typeof incr; // 'function'
incr(); // 1
incr(); // 2
incr(); // 3
```

Clausuras

Otro ejemplo de clausura donde guardamos la variable `i` del bucle en un ámbito privado para aprovecharla en el callback de `setTimeout`.

```
function temporizador() {
    for (let i = 1; i <= 3; i++) {
        setTimeout(function () {
            console.log(`Tiempo: ${i} segundos`);
        }, i * 1000);
    }
}

temporizador();
// Salida (con 1 segundo de diferencia):
// Tiempo: 1 segundos
// Tiempo: 2 segundos
// Tiempo: 3 segundos
```

OBJETOS

Qué son los objetos

Un objeto es una variable especial que puede contener más variables en su interior. De esta forma, tenemos la posibilidad de organizar múltiples variables de la misma temática en el interior de un objeto.

Pueden declararse con new o con un par de llaves (preferido).

```
● ● ●  
const objeto = {};  
const jugador = {  
    name: "hero",  
    life: 99,  
    power: 10,  
};
```

Propiedades de un objeto

Una vez tengamos un objeto, podemos acceder a sus propiedades de dos formas diferentes: a través de la notación con puntos o a través de la notación con corchetes.

```
// Notación con puntos (preferida)
console.log(player.name);      // Muestra "Hero"
console.log(player.life);      // Muestra 99

// Notación con corchetes
console.log(player["name"]);   // Muestra "Hero"
console.log(player["life"]);   // Muestra 99
```

Añadir propiedades

También podemos añadir propiedades al objeto después de haberlo creado, y no sólo en el momento de crear el objeto.

```
// FORMA 1: A través de notación con puntos
const player = {};

player.name = "Hero";
player.life = 99;
player.power = 10;

// FORMA 2: A través de notación con corchetes
const player = {};

player["name"] = "Hero";
player["life"] = 99;
player["power"] = 10;
```

Métodos de un objeto

Si dentro de una variable del objeto metemos una función (o una variable que contiene una función), tendríamos lo que se denomina un método de un objeto:

```
const user = {  
  name: "Manz",  
  talk: function() { return "Hola"; }  
};  
  
user.name;      // Es una variable (propiedad), devuelve "Manz"  
user.talk();    // Es una función (método), se ejecuta y devuelve "Hola"
```

Desestructuración de objetos

La desestructuración de objetos es una de las estrategias más utilizadas al trabajar en JS debido a la enorme utilización de objetos, donde conviene simplificar al máximo.

Consiste en separar en variables las propiedades de un objeto.

```
● ● ●

const character = {
  name: "Hero",
  role: "Wizard",
  life: 99
}
const { name, role, life } = character;
console.log(name, role, life);
```

Desestructuración de objetos

También podemos renombrar las propiedades e indicar valores por defecto para los casos en las que alguna de ellas no exista.

```
const { name, role: type, life } = user;
console.log({ name, type, life });

const { name, role = "normal user", life = 100 } = user;
console.log({ name, role, life });
```

Copias de objetos

En Javascript, así como en muchos otros lenguajes, necesitaremos en ocasiones copiar o clonar elementos de nuestro código, de forma que podamos cambiar uno y dejar intacto el original. Para ello, Javascript (al igual que en otros lenguajes) tiene dos mecanismos para copiar elementos:

- Copia por valor (Duplica el contenido)
- Copia por referencia (Hace referencia a dónde está el contenido)

Copia por valor

La copia por valor, se realiza con los tipos de datos más básicos, es decir, los tipos de datos primitivos.

```
let originalValue = 42;

// Creamos una copia del valor de
originalValue
let copy = originalValue;

originalValue;    // 42
copy;            // 42

// Alteramos el valor de copy
copy = 55;

originalValue;    // 42
copy;            // 55
```

Copia por referencia

Con estructuras de datos complejas como arrays y objetos, la información no se copia por valor, si no que se guarda una referencia a la misma dirección de memoria

```
let originalValue = { name: "Mariano" };

// Creamos una supuesta copia del valor de originalValue
let copy = originalValue;

originalValue;    // { name: "Mariano" }
copy;            // { name: "Mariano" }

// Alteramos el valor de copy
copy.name = "Emilio";

originalValue;    // { name: "Emilio" }
copy;            // { name: "Emilio" }
```

Clonación de objetos

Dos conceptos:

- Clonación superficial: Se copia su primer nivel, mientras que segundo y niveles más profundos, se crean referencias.
- Clonación profunda: Clonación de una estructura de datos a todos sus niveles.



```
const data = {  
    name: "Programador",      // Se clona en superficial y en profundidad  
    tired: false,             // Se clona en superficial y en profundidad  
    likes: ["css", "javascript", "html", "vue"], // Sólo en profundidad  
    numbers: [4, 8, 15, 16, 23, 42]           // Sólo en profundidad  
}
```

Métodos de clonación

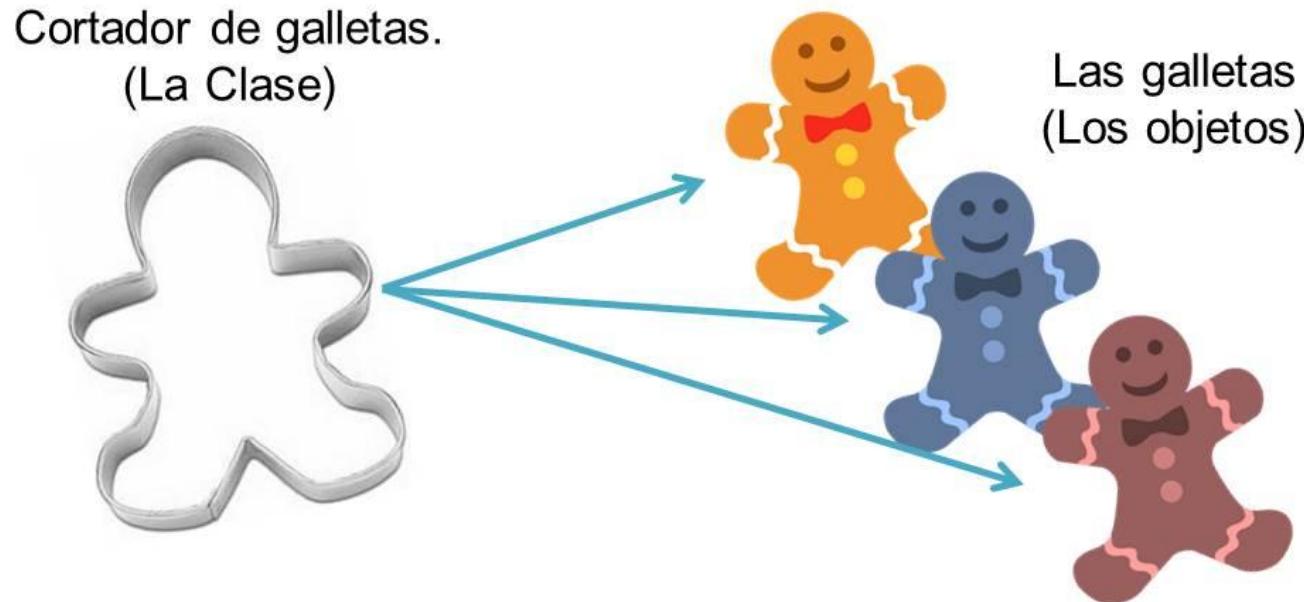
Método	Clonación superficial	Clonación profunda	Tipos avanzados
Operador =	No	No	No
Object.assign()	Si	No	No
structuredClone()	Si	Si	Salvo funciones

Métodos de clonación

```
const data = {  
    name: "Numeros",  
    numbers: [4, 8, 15, 16, 23, 42]  
}  
const copy = data;  
const copy2={};  
Object.assign(copy2, data);  
const copy3 = structuredClone(data);  
data.name="juan";  
data.numbers[3]=35;  
console.log(data,copy,copy2,copy3);  
  
//{name: 'juan', [4, 8, 15, 35, 23, 42]}  
//{name: 'juan', [4, 8, 15, 35, 23, 42]}  
//{name: 'Numeros', [4, 8, 15, 35, 23, 42]}  
//{name: 'Numeros', [4, 8, 15, 16, 23, 42]}
```

Orientación a objetos

La clase es el concepto abstracto de un objeto, mientras que el objeto es el elemento final que se basa en la clase. La clase es una “plantilla” para crear objetos



Organización del código

Cuando trabajamos con clases en JavaScript, es una buena práctica organizar el código en módulos para que cada clase esté en su propio archivo. Un esquema común sería organizar un archivo por cada clase, con nombres descriptivos. Por ejemplo:

```
/project
    └── index.html          (Archivo HTML principal)
    └── main.js              (Archivo principal que usa las clases)
    └── clases/
        └── Persona.js      (Clase Persona)
        └── Estudiante.js   (Clase Estudiante, que hereda de Persona)
```

Importación de archivos

A través de import se pueden cargar clases, funciones o constantes exportadas desde otros módulos,

```
//Exportación
export default class Persona {
    constructor(nombre) {
        this.nombre=nombre;
    }
}
```

```
// Importación
import { PI, suma, Persona } from './archivo.js';
import Persona from './archivo.js';
import Persona, { PI, suma } from './archivo.js';
import * as utils from './archivo.js';
console.log(utils.PI); // Acceder a PI
```

```
<script type="module" src="./archivo.js"></script>
```

Instanciar una clase

Se le llama instanciar una clase a la acción de crear un nuevo objeto basado en una clase particular. Esta acción la realizamos a través de la palabra clave new, seguida del nombre de la clase.



```
// Declaración de una clase (de momento, vacía)
class Animal {}

// Crear (instanciar) un objeto basada en una clase
const pato = new Animal();
```

Miembros de una clase

Los atributos: Variables dentro de clases

Los métodos: Funciones dentro de clases

```
● ● ●

class Animal {
    // Atributos
    name = "Garfield";
    type = "cat";

    // Métodos
    hablar() {
        return "Odio los lunes."
    }
}
```

Atributos

Las clases, siendo estructuras para guardar y almacenar información, tienen unas variables que viven dentro de la clase. Estos elementos se llaman atributos.



```
class Personaje {  
    name;          // Propiedad sin definir (undefined)  
    type = "Player"; // Propiedad definida  
    lifes = 5;      // Propiedad definida con 5 vidas restantes  
    energy = 10;    // Propiedad definida con 10 puntos de energía  
}
```

Atributos

Tradicionalmente en Javascript, los atributos acostumbraban a definirse a través del constructor, mediante la palabra clave this, por lo que es muy probable que también te los encuentres declarados de esta forma, sin necesidad de declararlos fuera del constructor:

```
class Personaje {  
    constructor() {  
        this.name;                      // Atributo sin definir (undefined)  
        this.type = "Player";           // Atributo definido  
        this.lifes = 5;                 // Atributo definido con 5 vidas restantes  
        this.energy = 10;                // Atributo definido con 10 puntos de energía  
    }  
}
```

Atributos

A la hora de utilizarlos, simplemente accedemos a ellos haciendo uso de la palabra clave `this`.

```
class Personaje {  
    name;           // Atributo sin definir (undefined)  
    type = "Player"; // Atributo definido  
    lifes = 5;       // Atributo definido con 5 vidas restantes  
    energy = 10;      // Atributo definido con 10 puntos de energía  
  
    constructor(name) {  
        this.name = name; // Modificamos el valor del atributo name  
        console.log(`¡Bienvenido/a, ${this.name}!`); // Accedemos al valor de name  
    }  
  
    const mario = new Personaje("Mario"); // '¡Bienvenido/a, Mario!'
```

Atributos privados

Por defecto, todas los atributos y métodos son públicos por defecto, sin embargo, si añadimos el carácter # justo antes del nombre del atributo, se tratará de un atributo privado:

```
class Personaje {  
    #name;  
    energy = 10;  
  
    constructor(name) {  
        this.#name = name;  
    }  
}
```

Atributos privados



```
class Personaje {  
    #name;  
    energy = 10;  
  
    constructor(name) {  
        this.#name = name;  
    }  
}  
  
const mario = new Personaje("Mario");      // { name: "Mario", energy: 10 }  
  
mario.name; // Es incorrecto, el nombre correcto de la propiedad es #name  
  
// Los dos siguientes dan el mismo error (no se puede acceder a la propiedad privada)  
mario.#name;  
mario.#name = "Evil Mario";  
  
// Lo siguiente funcionará, pero ha creado otra propiedad 'name' que no es la misma que '#name'  
mario.name = "Evil Mario";
```

Encapsulamiento

Los getters y setters son herramientas útiles para encapsular el acceso a propiedades privadas o protegidas. Se utilizan para:

- Controlar cómo se accede y modifica el estado interno de un objeto.
- Validar datos antes de asignarlos.
- Abstraer la implementación interna de una propiedad.

Encapsulamiento

```
class Persona {  
    #edad; // Propiedad privada  
  
    constructor(nombre, edad) {  
        this.nombre = nombre;  
        this.#edad = edad;  
    }  
  
    // Getter para acceder a la propiedad privada  
    get edad() {  
        return this.#edad;  
    }  
  
    // Setter para modificar la propiedad privada con validación  
    set edad(nuevaEdad) {  
        if (nuevaEdad < 0) {  
            console.log('La edad no puede ser negativa');  
        } else {  
            this.#edad = nuevaEdad;  
        }  
    }  
}  
  
const persona = new Persona('Ana', 25);  
console.log(persona.edad); // 25  
persona.edad = -5; // "La edad no puede ser negativa"  
persona.edad = 30;  
console.log(persona.edad); // 30
```

Getters y setters para variables computadas

En otros casos, los getters y setters se usan para definir propiedades cuyo valor depende de cálculos basados en otros datos del objeto.

```
● ● ●

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

    // Getter para calcular el área
    get area() {
        return this.ancho * this.alto;
    }

    // Setter para actualizar dimensiones proporcionalmente
    set area(nuevaArea) {
        this.alto = nuevaArea / this.ancho; // Mantener proporción
    }
}

const rectangulo = new Rectangulo(5, 10);
console.log(rectangulo.area); // 50
rectangulo.area = 100;          // Ajusta alto para mantener el área
console.log(rectangulo.alto); // 20
```

Métodos de clase

Los métodos son funciones que se incluyen dentro de las clases.

```
class Animal {  
    hablar() {  
        return "Cuak";  
    }  
}  
// Creación de una instancia u objeto (pato)  
const pato = new Animal();  
pato.hablar(); // 'Cuak'
```

Métodos estáticos

Los métodos estáticos son funciones de una clase, pero que no requieren crear una instancia para ejecutarlos, sino que se pueden ejecutar directamente haciendo referencia al nombre de la clase. En ellos solo podremos hacer referencia a elementos que también sean estáticos.

```
class Animal {  
    static despedirse() {  
        return "Adiós";  
    }  
  
    hablar() {  
        return "Cuak";  
    }  
}  
  
Animal.despedirse();           // Método estático (no requiere  
                                // instancia): 'Adiós'  
Animal.hablar();               // Uncaught TypeError: Animal.hablar  
                                // is not a function  
  
const pato = new Animal();     // Creamos una instancia  
  
pato.despedirse();            // Uncaught TypeError: pato.despedirse  
                                // is not a function  
pato.hablar();                // Método no estático (requiere  
                                // instancia): 'Cuak'
```

Métodos privados

Añadiendo un # antes del nombre del método conseguimos que sea privado.

```
class Personaje {  
    name = "Mario";  
  
    constructor() {  
        this.#hablar();  
    }  
  
    #hablar() {  
        console.log("It's me, Mario!");  
    }  
}  
  
const mario = new Personaje();      // It's me, Mario! (se ha accedido a #hablar() desde dentro de la clase)  
  
// Da error, no se permite acceder a un método privado desde fuera de la class  
mario.#hablar();  
  
// Da error, el método hablar() no existe, ya que el nombre del método es #hablar()  
mario.hablar()
```

ARRAYS

Inicialización de arrays

Un array es una colección o agrupación de elementos en una misma variable, cada uno de ellos ubicado por la posición que ocupa en el array. En Javascript, se pueden definir de varias formas:

```
// Forma tradicional (no se suele usar en Javascript)
const letters = new Array("a", "b", "c");    // Array con 3 elementos
const letters = new Array(3);                  // Array vacío de tamaño 3

// Mediante literales (notación preferida)
const letters = ["a", "b", "c"];   // Array con 3 elementos
const letters = [];              // Array vacío (0 elementos)
const letters = ["a", 5, true];   // Array mixto (String, Number, Boolean)
```

Acceso a los elementos de un array

La propiedad `.length` nos devolverá el número de elementos existentes en un array. Para acceder al contenido de un elemento específico basta con utilizar el operador `[]`, permitiendo operaciones de lectura y escritura. Cabe destacar que las operaciones de escritura pueden mutar el array añadiendo elementos nuevos

```
● ● ●

const letters = ["a", "b", "c"];

letters.length;    // 3
letters[0];        // 'a'
letters[2];        // 'c'
letters[5];        // undefined
letters[1] = "Z";  // Da "Z" y modifica letters a ["a", "Z", "c"]
letters[3] = "D";  // Da "D" y modifica letters a ["a", "Z", "c", "D"]
letters[5] = "A";  // Da "A" y modifica letters a ["a", "Z", "c", "D", undefined, "A"]
```

Acceso a los elementos de un array con .at()

El método `.at()` permite acceder a los elementos de un array usando números negativos, con lo que se recuperan los valores en orden inverso. Es una operación de solo lectura.

```
const letters = ["a", "b", "c"];

letters.at(0);    // "a"
letters.at(1);    // "b"
letters.at(3);    // undefined
letters.at(-1);   // "c"
letters.at(-2);   // "b"
```

Añadir o eliminar elementos

Métodos que añaden o retiran elementos de un array, cambiando su tamaño.

Método	Descripción
<code>.push(e1, e2, e3)</code>	Añade uno o varios elementos al final del array, devuelve el nuevo tamaño.
<code>.pop()</code>	Elimina el último elemento del array. Devuelve dicho elemento.
<code>.unshift(e1, e2, e3)</code>	Añade uno o varios elementos al inicio del array. Devuelve el nuevo tamaño.
<code>.shift()</code>	Elimina el primer elemento del array. Devuelve dicho elemento.

Añadir o eliminar elementos

```
const elements = ["a", "b", "c"]; // Array inicial

elements.push("d");      // Devuelve 4. Ahora elements = ['a', 'b', 'c', 'd']
elements.pop();          // Devuelve 'd'. Ahora elements = ['a', 'b', 'c']

elements.unshift("Z");   // Devuelve 4. Ahora elements = ['Z', 'a', 'b', 'c']
elements.shift();         // Devuelve 'Z'. Ahora elements = ['a', 'b', 'c']
```

Los tipos de datos dentro de un array pueden ser distintos

```
const miArray = [
    42,                      // Número
    "Hola",                  // Cadena
    true,                    // Booleano
    { clave: "valor" },     // Objeto
    [1, 2, 3],               // Otro array
    function() {             // Función
        console.log("Soy una función dentro de un array");
    }
];

// Accediendo a los elementos del array
console.log(miArray[0]); // 42
console.log(miArray[1]); // "Hola"
console.log(miArray[3].clave); // "valor"
console.log(miArray[4][1]); // 2

// Ejecutando la función dentro del array
miArray[5](); // "Soy una función dentro de un array"
```

El método concat

Nos permite unir los elementos pasados por parámetro en un array a la estructura que estamos manejando. Se podría pensar que los métodos `.push()` y `concat()` funcionan de la misma forma, pero no es exactamente así.



```
const elements = [1, 2, 3];

elements.push(4, 5, 6);      // Devuelve 6. Ahora elements = [1, 2, 3, 4, 5, 6]
elements.push([7, 8, 9]);    // Devuelve 7. Ahora elements = [1, 2, 3, 4, 5, 6, [7, 8, 9]]

const firstPart = [1, 2, 3];
const secondPart = [4, 5, 6];

firstPart.concat(secondPart);           // Devuelve [1, 2, 3, 4, 5, 6]
firstPart.concat(4, 5, 6);             // Devuelve [1, 2, 3, 4, 5, 6]
firstPart.concat(firstPart, 7);        // Devuelve [1, 2, 3, 1, 2, 3, 4, 5, 6, 7]
```

Desestructuración de arrays

Consiste en separar elementos de un array y sacarlo a variables individuales.



```
const elements = [5, 2];
const [first, last] = elements;    // first = 5, last = 2

const elements = [5, 4, 3, 2];
const [first, second] = elements; // first = 5, second = 4, rest = discard

const elements = [5, 4, 3, 2];
const [first, , third] = elements; // first = 5, third = 3, rest = discard

const elements = [4];
const [first, second] = elements; // first = 4, second = undefined
```

Desestructuración para intercambio de variables

Sin desestructuración necesitamos una variable auxiliar y tres asignaciones para realizar el intercambio.
Con desestructuración lo resolvemos con una sola línea.



```
let a = 10;  
let b = 5;
```

```
// Sin desestructuración  
let aux = a;  
a = b;  
b = aux;
```

```
// Con desestructuración  
[a, b] = [b, a];
```

Operador spread (...)

Se utiliza para expandir elementos de un array, objeto o cadena en otro contexto donde se esperan valores separados (como argumentos de funciones o elementos de un array).

```
//Función que separa un array
const debug = (param) => console.log(...param);
const array = [1, 2, 3, 4, 5];
debug(array);           // 1 2 3 4 5

//Combinando dos arrays usando spread
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
//Sin spread se crearía un array de dos elementos array
const combinado1 = [...array1, ...array2];
const combinado2 = [array1, array2];
console.log(combinado1); // [1, 2, 3, 4, 5, 6]
console.log(combinado2) // [[1, 2, 3], [4, 5, 6]]
```

Operador rest (...)

Se utiliza para agrupar los argumentos restantes en una función o elementos sobrantes de una estructura de datos. En esencia, "recoge el resto de los valores" en un array u objeto.

```
//Agrupando parámetros
const debug = (...param) => console.log(param);
debug(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]

//Agrupando el resto de un array en otro array
const [primero, segundo, ...resto] = [10, 20, 30, 40, 50];
console.log(primer); // 10
console.log(segundo); // 20
console.log(resto); // [30, 40, 50]
```

Métodos de búsqueda en un array

Método	Descripción
<code>.includes(element)</code>	Comprueba si element está incluido en el array.
<code>.includes(element, from)</code>	Lo mismo, partiendo desde from.
<code>.indexOf(element)</code>	Devuelve el índice de la primera aparición de element.
<code>.indexOf(element, from)</code>	Lo mismo, partiendo desde from.
<code>.lastIndexOf(element)</code>	Devuelve el índice de la última aparición de element.
<code>.lastIndexOf(element, from)</code>	Lo mismo, partiendo desde from.

Búsqueda en un array

```
● ● ●

const names = [
  { name: "María", age: 20 },
  { name: "Jorge", age: 32 },
  { name: "Pancracio", age: 22 },
  { name: "Andrea", age: 19 },
  //etc
];

// Busca el primer elemento con la edad indicada, sino devuelve -1
const findElement = (array, searchedAge) => {
  for (let i = 0; i < array.length; i++) {
    const element = array[i];
    if (element.age === searchedAge) {
      return element;
    }
  }
  return -1;
}

findElement(names, 19);      // { name: "Andrea", age: 19 }
findElement(names, 32);      // { name: "Jorge", age: 32 }
findElement(names, 33);      // -1
```

Búsqueda en un array usando find()

find en JavaScript se utiliza para buscar el primer elemento en un array que cumple con una condición especificada en un callback. El único parámetro obligatorio del callback es una variable que guarda cada elemento del array.



```
const findElement = (array, searchedAge) => {
  return array.find(element => element.age === searchedAge) ?? -1;
}

findElement(names, 19);      // { name: "Andrea", age: 19 }
findElement(names, 32);      // { name: "Jorge", age: 32 }
findElement(names, 33);      // -1
```

Algunos métodos más para arrays

Método	Descripción
<code>.slice(start, end)</code>	Devuelve un array con los elementos desde start hasta end (excluido).
<code>.fill(element, start, end)</code>	Cambia los elementos del array por element desde start hasta end.
<code>.reverse()</code>	Invierte los elementos del array.
<code>.sort()</code>	Ordena los elementos de un array alfabéticamente.
<code>.sort(criterio)</code>	Ordena los elementos siguiendo las indicaciones de una función.
<code>.forEach(callback)</code>	Ejecuta el callback para cada uno de los elementos del array.
<code>every(f) , some(f)</code>	Comprueba si todos/algún elemento del array cumple la condición de f.
<code>map(f)</code>	Devuelve un array con el resultado de ejecutar f sobre cada elemento.

Ejemplo con forEach()



```
const letters = ["a", "b", "c", "d"];  
  
letters.forEach(element) => console.log(element));  
// Devuelve 'a' / 'b' / 'c' / 'd'
```

Ejemplo con sort(criterio)

El método sort compara dos elementos consecutivos (a y b) usando el callback proporcionado:

- Si el callback devuelve un número negativo, se mantiene el orden actual (a antes que b).
- Si devuelve cero, no cambia el orden.
- Si devuelve un número positivo, intercambia a y b.



```
const personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 32 },
  { nombre: "Maria", edad: 20 },
  { nombre: "Carlos", edad: 28 }
];

// Ordenar de menor a mayor edad
personas.sort((a, b) => a.edad - b.edad);

console.log(personas);
/*
[
  { nombre: "Maria", edad: 20 },
  { nombre: "Ana", edad: 25 },
  { nombre: "Carlos", edad: 28 },
  { nombre: "Luis", edad: 32 }
]
*/
```

Otro ejemplo con sort(criterio)

```
const personas = [
    { nombre: "Ana", edad: 25 },
    { nombre: "Luis", edad: 32 },
    { nombre: "Maria", edad: 20 },
    { nombre: "Carlos", edad: 28 }
];

// Ordenar por nombre de Z a A
personas.sort((a, b) => b.nombre.localeCompare(a.nombre));

console.log(personas);
/*
[
    { nombre: "Maria", edad: 20 },
    { nombre: "Luis", edad: 32 },
    { nombre: "Carlos", edad: 28 },
    { nombre: "Ana", edad: 25 }
]
*/
```

Ejemplo con map(f)

```
● ● ●

const personas = [
  { nombre: "Ana", edad: 25 },
  { nombre: "Luis", edad: 32 },
  { nombre: "Maria", edad: 20 }
];

// Crear un nuevo array solo con los nombres
const nombres = personas.map(persona => persona.nombre);

console.log(nombres); // ["Ana", "Luis", "Maria"]
```

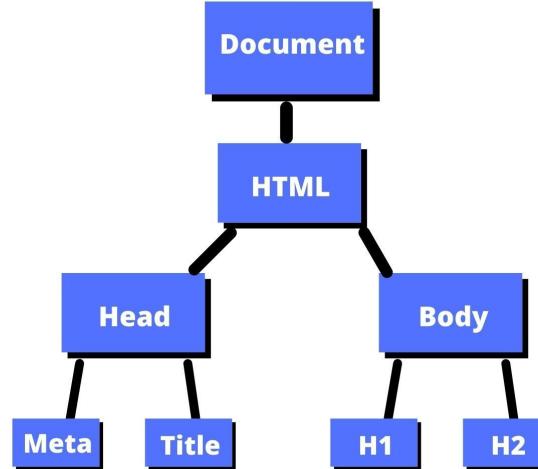
DOM
(DOCUMENT OBJECT MODEL)

¿Qué es el DOM?

Una página HTML está formada por múltiples etiquetas HTML, anidadas una dentro de otra, formando un árbol de etiquetas relacionadas entre sí, que se denomina árbol DOM (o simplemente DOM). En JS podemos acceder a esa estructura para modificarla.



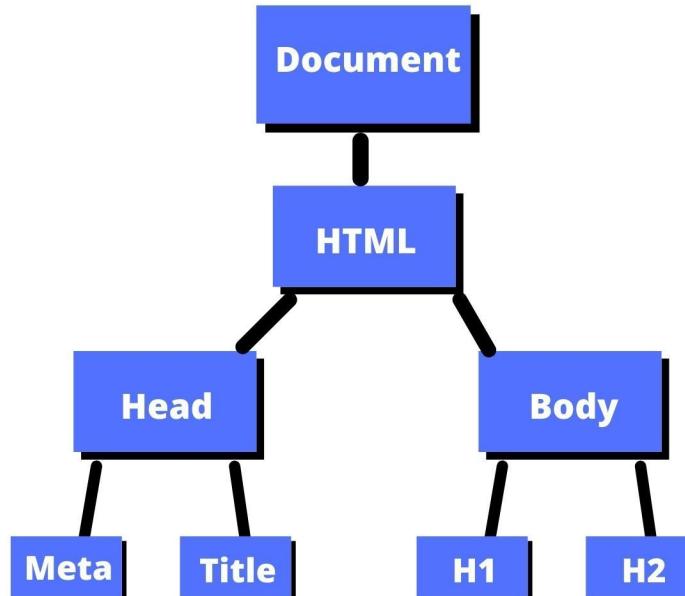
```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi página</title>
</head>
<body>
    <h1>Un título</h1>
    <h2>Un subtítulo</h2>
</body>
</html>
```



El objeto document

Desde el navegador, la forma de acceder al DOM es a través de un objeto Javascript llamado `document`, que representa el árbol DOM de la página.

En JS, existe el objeto `document` para acceder al DOM.



Métodos tradicionales de búsqueda en el DOM

Método	Descripción
<code>.getElementById(id)</code>	Devuelve un elemento HTML con la misma id.
<code>.getElementsByClassName(class)</code>	Devuelve un array de elementos con la misma clase.
<code>.getElementsByName(value)</code>	Devuelve un array de elementos con un atributo name cuyo valor coincide con el parámetro.
<code>.getElementsByTagName(tag)</code>	Devuelve un array de elementos con la misma etiqueta html

getElementById(id)

Busca un elemento HTML con el id especificado. Un documento HTML bien construído no debería tener más de un elemento con el mismo id, por lo tanto, este método devolverá siempre un solo elemento.



```
<div id="page"></div>
```



```
const element = document.getElementById("page");
```

```
console.log(element); // <div id="page"></div>
```

getElementsByClassName()

Permite buscar los elementos que tengan la clase especificada en class. En este caso, al poder existir varios elementos con la misma clase este método devuelve un array.



```
const elements =  
document.getElementsByClassName("item");  
  
console.log(elements);          // [div, div, div]  
console.log(elements[0]);        // Primer item  
encontrado: <div class="item"></div>  
console.log(elements.length);   // 3
```



```
<div class="container">  
  <div class="item">Item 1</div>  
  <div class="item">Item 2</div>  
  <div class="item">Item 3</div>  
</div>
```

En realidad se devuelve un HTMLCollection

Aunque parece que se devuelve un array en la función anterior, lo que en realidad se devuelve es un elemento de tipo HTMLCollection

```
const elements = document.getElementsByTagName("div");

elements instanceof Array;           // false
elements instanceof HTMLCollection; // true
elements.constructor.name;          // 'HTMLCollection'
```

En realidad se devuelve un HTMLCollection

La estructura HTMLCollection es una colección viva de elementos, lo que significa que si modificas elementos del DOM, se actualizan en la estructura, al contrario que si tienes un array.

```
// HTMLCollection
const collection = document.getElementsByTagName("div");
collection.length;          // 63
collection[62].remove();    // Eliminamos del DOM el último elemento
collection.length;          // 62 (Como el elemento no existe en el DOM, se borra)

// Array
const collection = [...document.getElementsByTagName("div")];
collection.length;          // 63
collection[62].remove();    // Eliminamos del DOM el último elemento
collection.length;          // 63 (Sigue teniendo el mismo número de elementos)
```

Métodos modernos de búsqueda en el DOM

Método	Descripción
<code>.querySelector(sel)</code>	Devuelve el primer elemento que coincide con el selector CSS introducido.
<code>.querySelectorAll(sel)</code>	Devuelve un array de elementos que coincidan con el selector CSS introducido

querySelector()

Este método permite suministrar un selector CSS, haciendo que sea mucho más potente que los métodos tradicionales.



```
const page = document.querySelector("#page");           // <div id="page"></div>
const info = document.querySelector(".main .info");    // <div class="info"></div>
```

El selector del primer caso es equivalente a getElementById, ya que usamos una #, el selector CSS para los ids.

El segundo caso no lo podríamos conseguir con los métodos tradicionales. ya que estamos usando un combinador descendente.

querySelectorAll()

El método `.querySelectorAll()` siempre nos devolverá un Array de elementos. En el caso de no encontrar ninguna coincidencia, nos devolverá un array de 0 elementos.



```
// Obtiene todos los elementos con clase "info"
const infos = document.querySelectorAll(".info");

// Obtiene todos los elementos con atributo name="nickname"
const nicknames = document.querySelectorAll('[name="nickname"]');

// Obtiene todos los elementos <div> de la página HTML
const divs = document.querySelectorAll("div");
```

Búsqueda acotada

Podemos realizar una búsqueda con `querySelector()` sobre un objeto devuelto por esta misma función.

```
const menu = document.querySelector("#menu");
const links = menu.querySelectorAll("a");

//Es equivalente a
const links = document.querySelectorAll("#menu a");
```

En realidad se devuelve un NodeList

NodeList también es una colección viva. Al realizar cambios en el DOM se reflejan en la colección.



```
var parent = document.getElementById("parent");
var child_nodes = parent.childNodes;
console.log(child_nodes.length); // asumamos "2"
parent.appendChild(document.createElement("div"));
console.log(child_nodes.length); // debería imprimir
                                "3"
```

Acceso al contenido del DOM

Propiedad	Descripción
nodeName	Devuelve el nombre del nodo (la etiqueta). Solo lectura.
textContent	Devuelve o cambia el contenido de texto del elemento.
innerHTML	Devuelve o cambia el contenido HTML del interior del elemento
outerHTML	Como el anterior, pero incluye también la etiqueta HTML

nodeName

Nos permite obtener el nombre de la etiqueta.



```
// <div class="container"></div>
const element = document.querySelector("div");

element.nodeName;          // DIV
```

textContent

La propiedad recomendada para hacer modificaciones de texto es `.textContent`, que nos devuelve el contenido de texto de un elemento HTML concreto.



```
<div class="container">
  <div class="parent">
    <p>Hola a todos.</p>
    <p class="message">Mi nombre es <strong>Jose</strong>.</p>
  </div>
</div>
```



```
const element = document.querySelector(".message");

element.textContent; // "Mi nombre es Jose."
element.textContent = "Hola a todos"; // Modificamos el contenido de texto
element.textContent; // "Hola a todos"
```

innerHTML

La propiedad `.innerHTML` nos permite acceder al contenido de un elemento, pero en lugar de devolver su contenido de texto como lo hace `.textContent`, esta propiedad nos devuelve su contenido HTML.



```
const element = document.querySelector(".message");

element.innerHTML;    // "Mi nombre es <strong>Jose</strong>."
element.textContent;  // "Mi nombre es Jose."

element.innerHTML = "<strong>Importante</strong>";    // Se lee "Importante" (en negrita)
element.textContent = "<strong>Importante</strong>";    // Se lee "<strong>Importante</strong>"
```

outerHTML

Mientras innerHTML devuelve el código HTML del interior de la etiqueta, outerHTML devuelve el código HTML con la etiqueta del propio elemento.



```
const data = document.querySelector(".data");
data.innerHTML = "<h1>Tema 1</h1>";

data.textContent;      // "Tema 1"
data.innerHTML;       // "<h1>Tema 1</h1>"
data.outerHTML;        // "<div class='data'><h1>Tema 1</h1></div>"
```

Acceder a atributos HTML

Las etiquetas HTML tienen ciertos atributos que definen su comportamiento. Una de las opciones para acceder a ellos en JS es llamarlos como propiedades. Para el caso concreto del atributo class, su traducción a propiedad es className para evitar el uso de una palabra reservada.



```
const element = document.querySelector("div");    // <div class="container"></div>  
  
element.id = "page";                // <div id="page" class="container"></div>  
element.style = "color: red";        // <div id="page" class="container" style="color: red"></div>  
element.className = "data";         // <div id="page" class="data" style="color: red"></div>
```

Acceso a atributos HTML mediante funciones

Aunque la forma anterior es la más rápida, tenemos algunos métodos para obtener los atributos HTML de forma clara y literal, sin problemas como los de `className`.

Método	Descripción
<code>hasAttributes ()</code>	Devuelve true si el elemento tiene atributos.
<code>hasAttribute (attr)</code>	Devuelve true si el elemento tiene el atributo attr.
<code>getAttributeNames ()</code>	Devuelve un array con los atributos del elemento.
<code>getAttribute (attr)</code>	Devuelve el valor del atributo attr o null si no existe.

Acceso a atributos HTML mediante funciones



```
<div id="page" class="info data dark" data-number="5"></div>
```



```
const element = document.querySelector("#page");

element.hasAttributes();           // true (tiene 3 atributos)
element.hasAttribute("data-number"); // true (data-number existe)
element.hasAttribute("disabled");   // false (disabled no existe)

element.getAttributeNames();        // ["id", "data-number", "class"]
element.getAttribute("id");         // "page"
```

Añadir, modificar y eliminar atributos HTML

Método	Descripción
<code>setAttribute(attr, value)</code>	Añade o cambia el atributo attr al valor value del elemento HTML.
<code>toggleAttribute(attr, force)</code>	Añade el atributo attr si no existe, si existe lo elimina. Force es un booleano opcional. Si es true añade el atributo, si es false lo elimina.
<code>removeAttribute(attr)</code>	Elimina el atributo attr del elemento HTML.

Añadir, modificar y eliminar atributos HTML

```
<div id="page" class="info data dark" data-number="5"></div>
```

```
//Con atributos con valor
const element = document.querySelector("#page");

element.setAttribute("data-number", "10");    // Cambiar data-number a 10
element.removeAttribute("id");                // Elimina el atributo id
element.setAttribute("id", "page");           // Vuelve a añadirla

//Con atributos booleanos
const button = document.querySelector("button");

button.setAttribute("disabled", true);    // ✗ <button disabled="true">Clickme!</button>
button.disabled = true;                  // ✓ <button disabled>Clickme!</button>
button.setAttribute("disabled", "");      // ✓ <button disabled>Clickme!</button>
button.toggleAttribute("disabled");      // Como ya existe "disabled", lo elimina
button.toggleAttribute("hidden");        // Como no existe "hidden", lo añade
```

Estilos CSS con el DOM

Las formas principales de modificar las clases o los estilos CSS mediante el DOM son las siguientes:

Propiedad	Descripción
<code>className</code>	Accede (y modifica) al valor del atributo class.
<code>checkVisibility()</code>	Devuelve true si el elemento es visible y false si está oculto (atributo hidden, display none o no existe).
<code>classList</code>	Devuelve un array con las clases del elemento HTML.
Acceso a los estilos CSS	
<code>style</code>	Objeto con las propiedades CSS asignadas al elemento.

Usando classList

Si accedemos a la propiedad u objeto `.classList`, nos devolverá un array (un `DOMTokenList` en realidad) de clases CSS de dicho elemento con métodos para trabajar con el.

Obtención de información	
Método	Descripción
<code>classList</code>	Devuelve un array con las clases del elemento.
<code>classList.length</code>	Devuelve el número de clases del elemento.
<code>classList.item(n)</code>	Devuelve la clase con el índice n dentro del array.
<code>classList.contains(clase)</code>	Indica si el elemento contiene la clase indicada.

Usando classList

Añadir y eliminar clases	
Método	Descripción
<code>classList.add(c1,c2...)</code>	Añade las clases indicadas al elemento HTML.
<code>classList.remove(c1,c2...)</code>	Elimina las clases indicadas del elemento HTML.
<code>classList.toggle(clase)</code>	Si la clase no existe la añade. Si no, la elimina.
<code>classList.replace(old,new)</code>	Reemplaza la clase old por la clase new.

Usando classList

```
const element = document.querySelector("#page");

// ¿Qué clases tiene?
element.classList;           // ["info", "data", "dark"] (DOMTokenList)
element.classList.value;     // "info data dark" (String)
element.classList.length;    // 3

// Convertirlas a array
Array.from(element.classList) // ["info", "data", "dark"] (Array)
[...element.classList];       // ["info", "data", "dark"] (Array)

// Consultarlas
element.classList.item(0);   // "info"
element.classList.item(1);   // "data"
element.classList.item(3);   // null

element.classList.add("uno", "dos");
element.classList; // ["info", "data", "dark", "uno", "dos"]

element.classList.remove("uno", "dos");
element.classList; // ["info", "data", "dark"]
```

El objeto style

Mediante el objeto style podemos acceder a las propiedades específicas, para obtener su valor. Hay muchas pero sólo algunas propiedades tendrán valores, que son las que están definidas en los estilos en línea del elemento HTML

Observa los
nombres de las
propiedades

```
title.style.backgroundColor; // "indigo"  
title.style.color; // "var(--color, #000)"  
title.style.padding; // "25px"  
title.style.paddingLeft; // "25px"  
title.style.paddingTop; // "25px"  
title.style["paddingTop"] // "25px"  
title.style["padding-top"] // "25px"  
title.style.fontFamily // ""
```

Métodos para acceder a los estilos

Utilizar las propiedades del objeto style es una forma antigua de programar.
Actualmente tenemos funciones que ofrecen más ventajas

Método	Descripción
<code>setProperty (propName , value)</code>	Añade o cambia el valor de una propiedad CSS.
<code>getPropertyValue (propName)</code>	Obtiene el valor de una propiedad CSS.
<code>removeProperty (propName)</code>	Elimina una propiedad CSS de un elemento.

getProperty

A diferencia del acceso mediante la propiedad del objeto style, con esta función se puede obtener, además de las propiedades CSS, las variables CSS.



```
title.style.getPropertyValue("background-color"); // "indigo"
title.style.getPropertyValue("color");           // "var(--color, #000)"
title.style.getPropertyValue("padding");          // "25px"
title.style.getPropertyValue("padding-left");     // "25px"
title.style.getPropertyValue("padding-top");      // "25px"
title.style.getPropertyValue("font-family");      // ""

const h1 = title.querySelector("h1");
h1.style.getPropertyValue("--color");             // "white"
```

setProperty

Para modificar los valores de las propiedades y de las variables CSS.



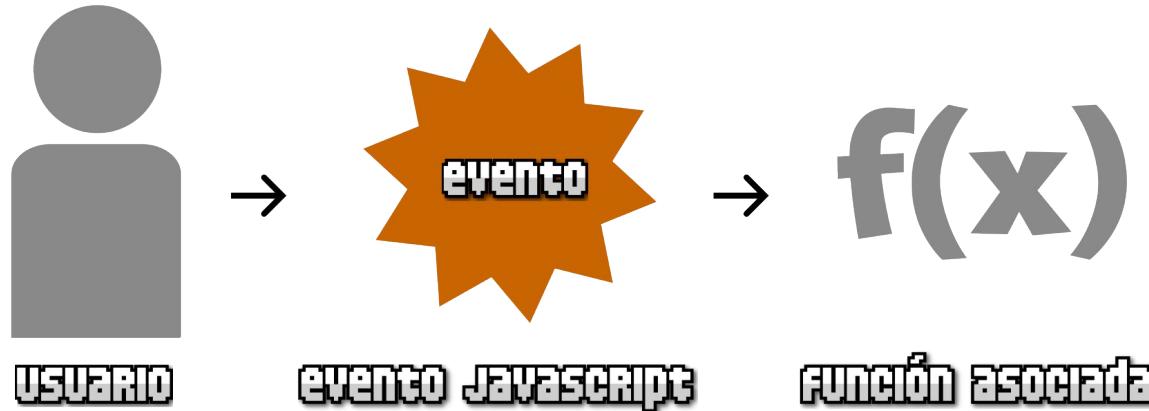
```
const h1 = title.querySelector("h1");
h1.style.setProperty("color", "gold");
h1.style.getPropertyValue("color");           // "gold"
```

EVENTOS

¿Qué son los eventos?

Existe un concepto llamado evento, una notificación de que alguna característica interesante acaba de ocurrir, generalmente relacionada con el usuario que navega por la página.

- Click de ratón del usuario sobre un elemento de la página
- Pulsación de una tecla específica del teclado
- Reproducción de un archivo de audio/video
- Scroll de ratón sobre un elemento de la página



Formas de manejar eventos

Mediante
addEventListener

Mediante atributos
HTML

Mediante
propiedades
Javascript

Eventos mediante atributos HTML

El valor del atributo onClick llevará la funcionalidad en cuestión que queremos ejecutar cuando se produzca el evento.



```
<script>
```

```
  function doTask( ) {  
    alert("Hello!");  
  }
```

```
</script>
```

```
<button onClick="doTask( )">Saludar</button>
```

Eventos mediante propiedades en javascript

La idea es la misma que en el caso anterior, salvo que en esta ocasión separamos el código HTML del código JS.

```
<button>Saludar</button>

<script>
const button = document.querySelector("button");
button.onclick = function() {
    alert("Hello!");
}
</script>
```

Mediante addEventListener

El método `.addEventListener()` permite añadir una escucha del evento indicado (primer parámetro), y en el caso de que ocurra, se ejecutará la función asociada indicada (segundo parámetro).

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

function action() {
    alert("Hello!");
};

button.addEventListener("click", action);
```

Mediante addEventListener

Es muy habitual utilizar funciones anónimas, incluso funciones flecha a la hora de pasar el callback a addEventListener.



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

button.addEventListener("click", () => alert("Hello!"));
```

Múltiples listeners

Dicho método `.addEventListener()` permite asociar múltiples funciones a un mismo evento. Intentar modificar la propiedad asociada al evento, por el contrario, sustituirá el que ya hubiese.



```
const button = document.querySelector("button");
const action = () => alert("Hello!");
const toggle = () => button.classList.toggle("red");

button.onClick = action;
button.onClick = toggle; //Se sustituye el anterior

button.addEventListener("click", action);
button.addEventListener("click", toggle); //Se asocia un segundo callback
```

Opciones en addEventListener

El método addEventListener acepta un tercer parámetro opcional. Se trata de un objeto que puede incluir distintas opciones para influir en el comportamiento del listener.



```
document.addEventListener("keydown", (event) => {
    console.log(`Primera tecla presionada: ${event.key}`);
}, { once: true });
```

LOCALSTORAGE

¿Qué es localStorage?

localStorage es una API de almacenamiento proporcionada por los navegadores que permite guardar datos de manera persistente en el cliente (navegador) sin fecha de expiración. Los datos almacenados permanecen incluso si el usuario recarga la página o cierra el navegador.

Soportado por:



Características de localStorage

- **Persistencia:** Los datos no se eliminan a menos que se borren manualmente o mediante código.
- **Tamaño máximo:** Aproximadamente 5MB por dominio.
- **Solo almacena texto:** Aunque puedes almacenar objetos, es necesario convertirlos a formato JSON.
- **Basado en clave-valor:** Similar a un diccionario u objeto JavaScript.
- **Accesible sólo desde el mismo origen:** Un sitio no puede acceder a localStorage de otro dominio.
- **No es seguro para datos sensibles:** Cualquier script en la página puede acceder a localStorage.
- **Sin soporte para eventos en tiempo real:** Si localStorage se actualiza en una pestaña, otras pestañas no recibirán cambios automáticamente.

Métodos de localStorage

Método	Descripción
<code>setItem(clave, valor)</code>	Almacena un valor accesible desde una clave.
<code>getItem(clave)</code>	Recupera un valor mediante su clave.
<code>removeItem(clave)</code>	Elimina un valor, obtenido mediante su clave.
<code>clear()</code>	Elimina todos los datos almacenados.
<code>key(indice)</code>	Obtiene la clave almacenada en la posición del índice.
<code>length</code>	Propiedad que devuelve el número de elementos almacenados.

Uso habitual: preferencias de usuario



```
// Guardar preferencia
localStorage.setItem("modoOscuro", "true");

// Aplicar preferencia
if (localStorage.getItem("modoOscuro") === "true") {
    document.body.classList.add("dark-mode");
}
```

Otro ejemplo: carrito de la compra

Como solo almacena texto,
transformamos los objetos y arrays
en un JSON



```
let carrito = JSON.parse(localStorage.getItem("carrito")) || [];
carrito.push({ id: 1, producto: "Laptop", precio: 1200 });
localStorage.setItem("carrito", JSON.stringify(carrito));
```

En las DevTools

The screenshot shows the Chrome DevTools interface with the Application tab selected. The left sidebar lists storage types: Manifest, Service workers, and Storage. Under Storage, Local storage is expanded, showing items for the origin `http://127.0.0.1:5500`. Session storage and Extension storage are also listed, along with IndexedDB.

The main area displays the contents of the Local storage for the specified origin:

Key	Value
Nombre	Carlos
persona	{"nombre": "Carlos", "edad": 30}

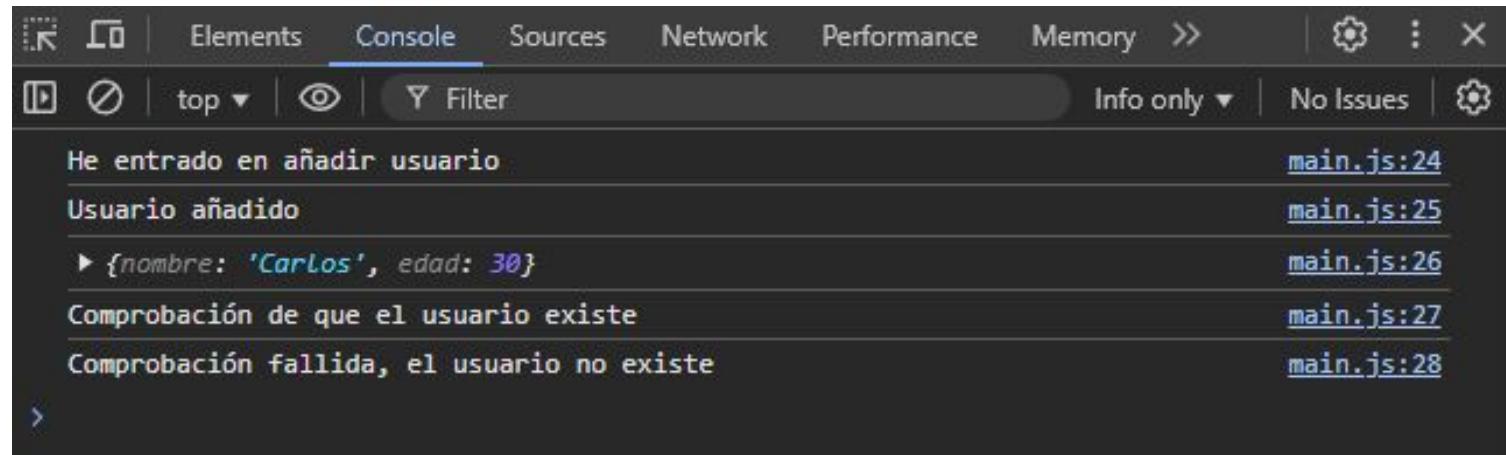
At the bottom, there is a summary table:

1	Carlos
---	--------

DEBUGGING

Depuración clásica con DevTools

Las DevTools cuentan con herramientas como la consola y el depurador que permiten inspeccionar variables, añadir breakpoints y mostrar mensajes para buscar errores.

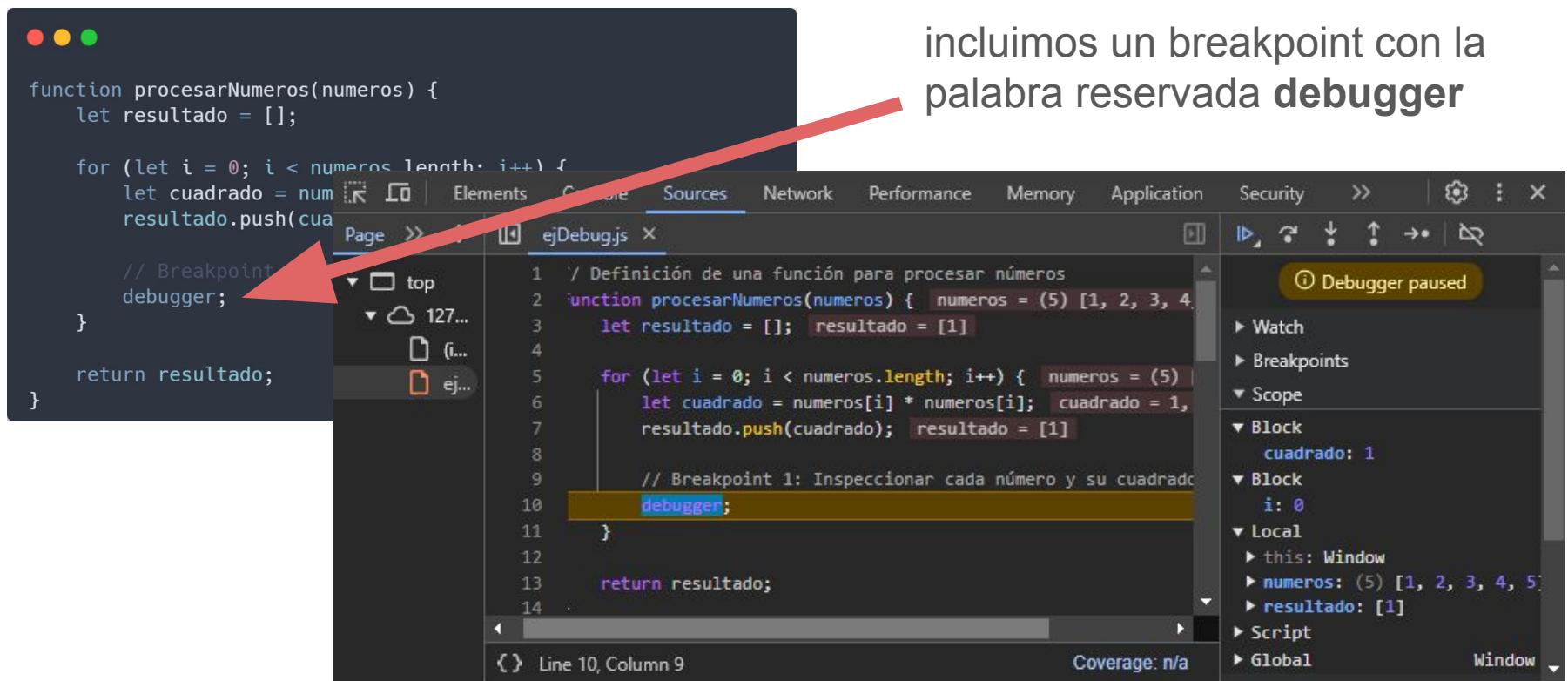


The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console output window displays the following messages:

- He entrado en añadir usuario main.js:24
- Usuario añadido main.js:25
- ▶ {nombre: 'Carlos', edad: 30} main.js:26
- Comprobación de que el usuario existe main.js:27
- Comprobación fallida, el usuario no existe main.js:28

Above the messages, there are several UI elements: a toolbar with icons for Elements, Network, and Performance; tabs for Elements, Console, Sources, Network, Performance, and Memory; and a status bar with 'Info only' and 'No Issues'.

Depuración clásica con DevTools



The screenshot shows the Chrome DevTools interface with the Sources tab selected. A red arrow points from the text "incluirmos un breakpoint con la palabra reservada **debugger**" to the word "debugger" in the code editor. The code editor displays a JavaScript function named "procesarNumeros". A yellow box highlights the line "10 debugger;". The status bar at the bottom indicates "Line 10, Column 9". To the right of the code editor is the DevTools sidebar, which shows the "Debugger paused" status and a list of scopes and variables. The "Scope" section is expanded, showing the variable "cuadrado" with a value of 1, the variable "i" with a value of 0, and the local variables "this", "numeros", and "resultado".

```
function procesarNumeros(numeros) {
  let resultado = [];

  for (let i = 0; i < numeros.length; i++) {
    let cuadrado = numeros[i] * numeros[i];
    resultado.push(cuadrado);
  }

  // Breakpoint
  debugger;

  return resultado;
}

// Definición de una función para procesar números
function procesarNumeros(numeros) { numeros = [5, 1, 2, 3, 4];
  let resultado = []; resultado = [1];

  for (let i = 0; i < numeros.length; i++) { numeros = [5];
    let cuadrado = numeros[i] * numeros[i]; cuadrado = 1;
    resultado.push(cuadrado); resultado = [1];

    // Breakpoint 1: Inspeccionar cada número y su cuadrado
    debugger;
  }

  return resultado;
}

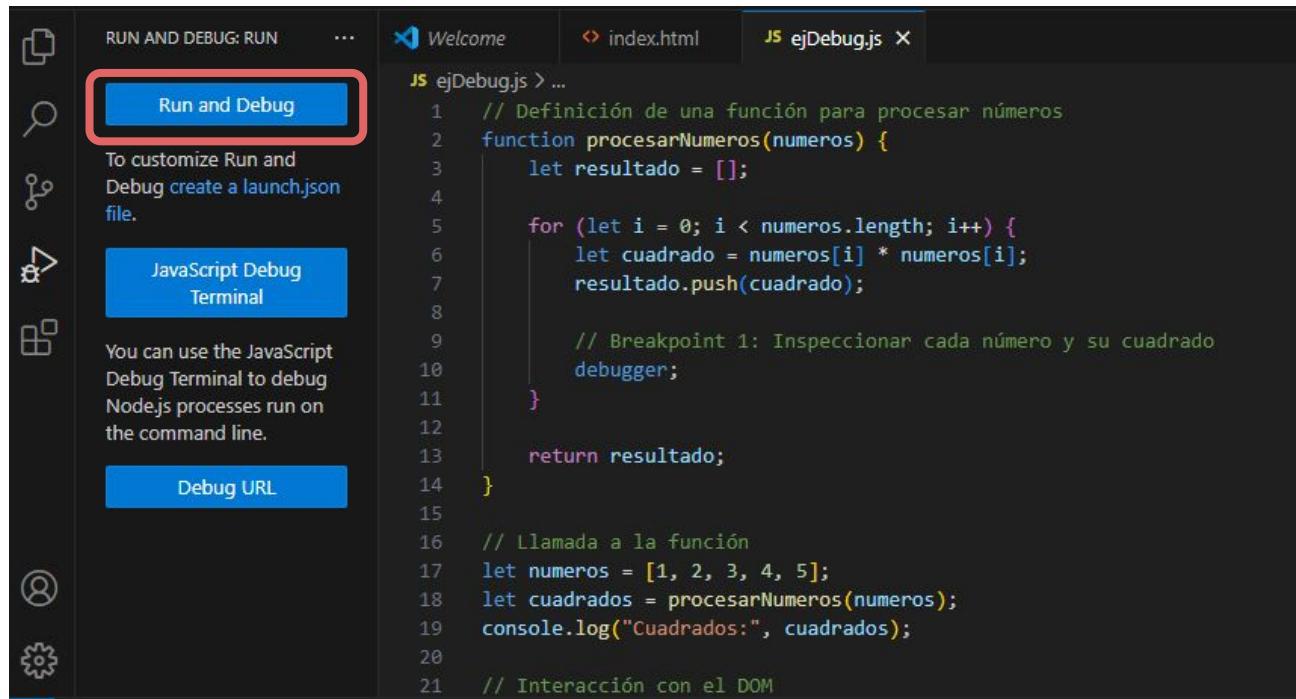
// Line 10, Column 9
```

Debugger paused

- Watch
- Breakpoints
- Scope
 - Block
 - cuadrado: 1
 - Block
 - i: 0
 - Local
 - this: Window
 - numeros: [5, 1, 2, 3, 4]
 - resultado: [1]
 - Script
 - Global

Coverage: n/a

Depuración con VSC



The screenshot shows the Visual Studio Code interface with the following elements:

- Run and Debug: RUN**: A button in the top left.
- Run and Debug**: A button in the top right of the Run and Debug sidebar, highlighted with a red box.
- To customize Run and Debug create a launch.json file.**: A note in the Run and Debug sidebar.
- JavaScript Debug Terminal**: A button in the bottom left of the Run and Debug sidebar.
- Debug URL**: A button in the bottom right of the Run and Debug sidebar.
- File Explorer**: The sidebar on the left.
- Search**: The sidebar on the left.
- Problems**: The sidebar on the left.
- Terminal**: The sidebar on the left.
- Output**: The sidebar on the left.
- Run and Debug**: The main tab bar with tabs for Welcome, index.html, and ejDebug.js (highlighted).
- ejDebug.js**: The code editor window containing the following JavaScript code:

```
// Definición de una función para procesar números
function procesarNumeros(numeros) {
    let resultado = [];

    for (let i = 0; i < numeros.length; i++) {
        let cuadrado = numeros[i] * numeros[i];
        resultado.push(cuadrado);

        // Breakpoint 1: Inspeccionar cada número y su cuadrado
        debugger;
    }

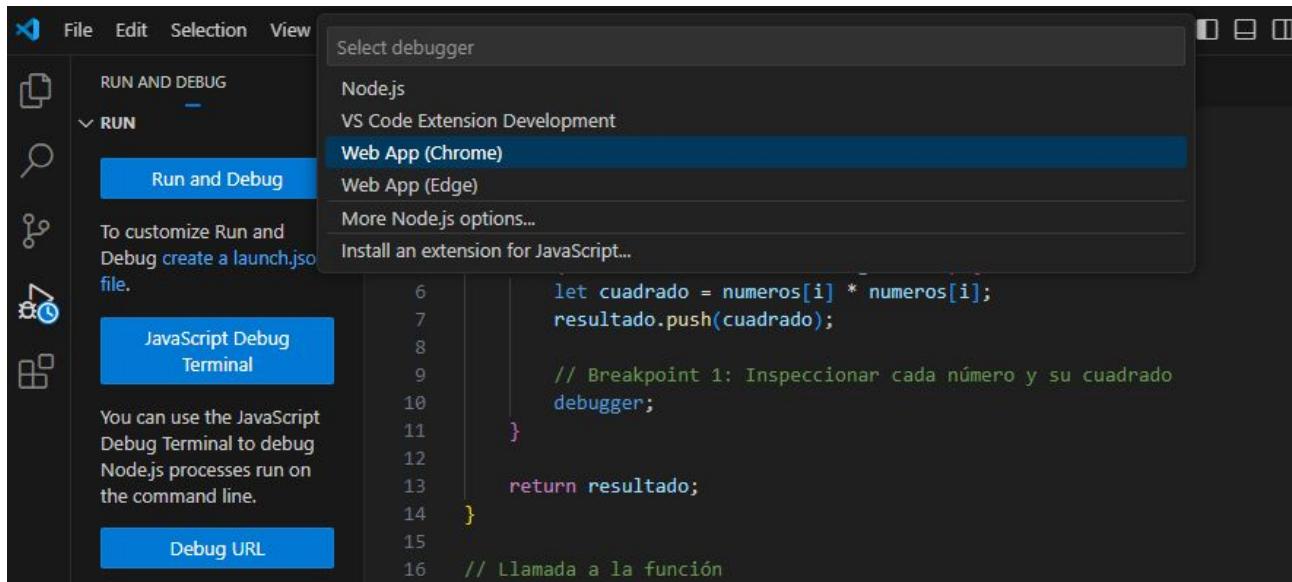
    return resultado;
}

// Llamada a la función
let numeros = [1, 2, 3, 4, 5];
let cuadrados = procesarNumeros(numeros);
console.log("Cuadrados:", cuadrados);

// Interacción con el DOM
```

Para configurar el debugger, lo primero es pulsar el botón de Run and Debug, en la pestaña de depuración de VSC.

Depuración con VSC



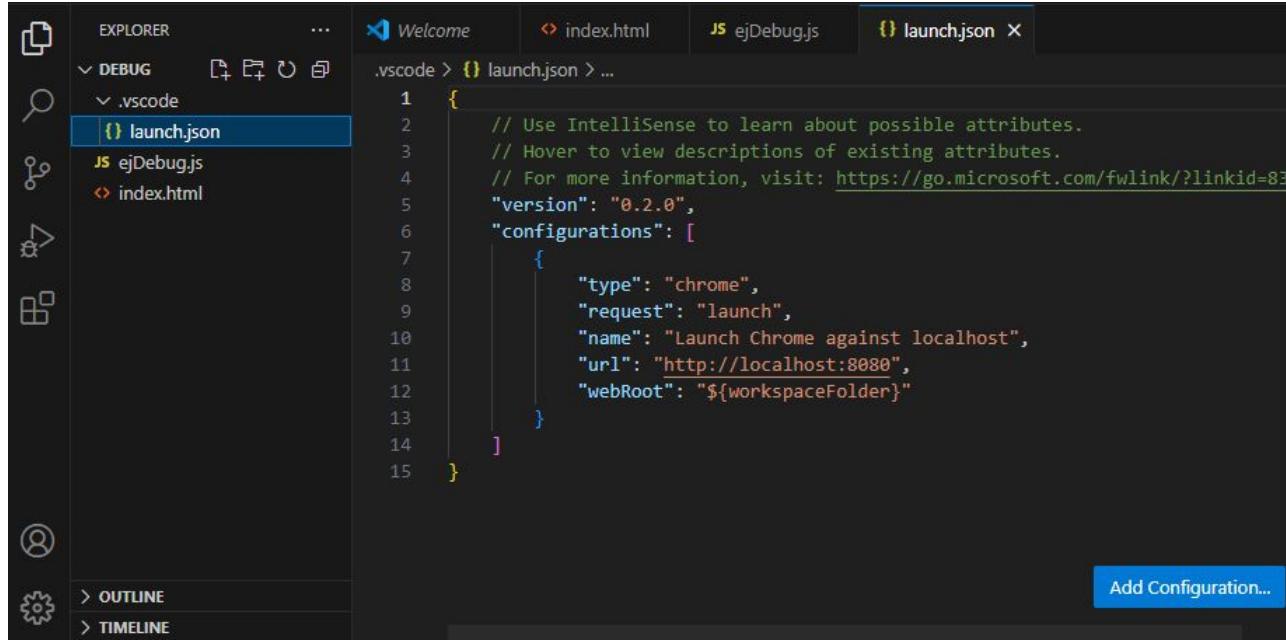
The screenshot shows the VS Code interface with the 'Run and Debug' menu open. The 'Select debugger' dropdown is visible, with 'Web App (Chrome)' highlighted in blue. The code editor displays a snippet of JavaScript code:

```
6  let cuadrado = numeros[i] * numeros[i];
7  resultado.push(cuadrado);
8
9  // Breakpoint 1: Inspeccionar cada número y su cuadrado
10 debugger;
11 }
12
13 return resultado;
14 }
15
16 // Llamada a la función
```

The code includes a breakpoint annotation and a call to the 'debugger' statement.

Seleccionamos el navegador con el que trabajamos.

Depuración con VSC



The screenshot shows the VS Code interface with the following details:

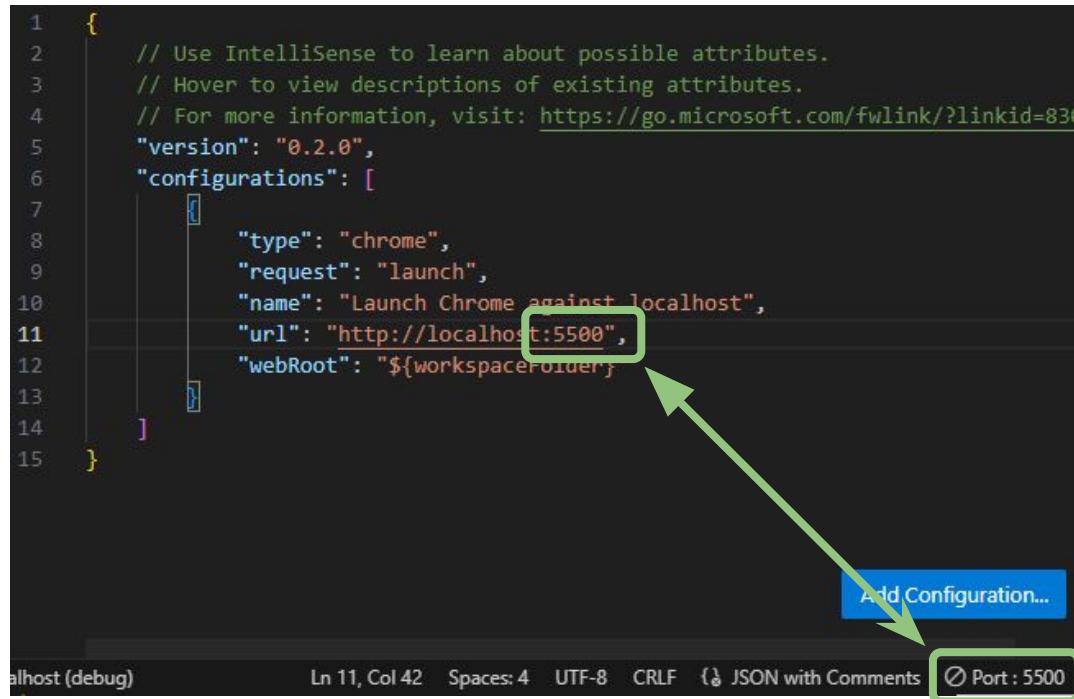
- EXPLORER** view: Shows a tree structure with **DEBUG** expanded, containing **.vscode**, **ejDebug.js**, and **index.html**. The **launch.json** file is selected.
- EDITOR**: The active tab is **launch.json**, which contains the following JSON configuration:

```
1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "type": "chrome",
9              "request": "launch",
10             "name": "Launch Chrome against localhost",
11             "url": "http://localhost:8080",
12             "webRoot": "${workspaceFolder}"
13         }
14     ]
15 }
```

- COMMANDS**: A blue button labeled **Add Configuration...**.

Se creará un JSON de configuración en la carpeta del proyecto.

Depuración con VSC

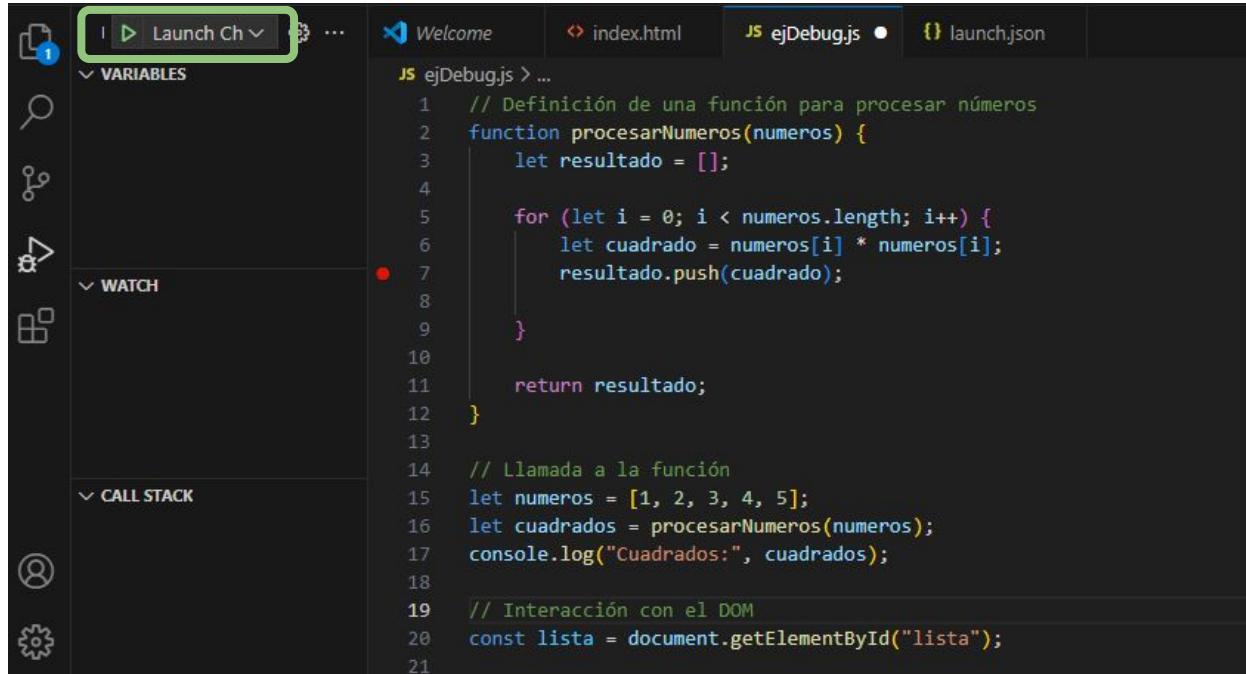


```
1  {
2      // Use IntelliSense to learn about possible attributes.
3      // Hover to view descriptions of existing attributes.
4      // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830
5      "version": "0.2.0",
6      "configurations": [
7          {
8              "type": "chrome",
9              "request": "launch",
10             "name": "Launch Chrome against localhost",
11             "url": "http://localhost:5500",
12             "webRoot": "${workspaceFolder}"
13         }
14     ]
15 }
```

alhost (debug) Ln 11, Col 42 Spaces: 4 UTF-8 CRLF { JSON with Comments ⚙ Port : 5500

Hay que modificar el puerto en la configuración del depurador para que coincida con el del servidor local que estamos utilizando (En nuestro caso, LiveServer con el puerto 5500)

Depuración con VSC



The screenshot shows the Visual Studio Code interface with the following details:

- Top Bar:** Shows tabs for "Welcome", "index.html", "ejDebug.js", and "launch.json".
- Left Sidebar:** Includes icons for file operations, search, and other development tools.
- Debugger Sidebar:** Opened on the left, showing sections for "VARIABLES", "WATCH", and "CALL STACK". A red dot in the "WATCH" section indicates a breakpoint is active.
- Code Editor:** Displays the following JavaScript code in the "ejDebug.js" file:

```
// Definición de una función para procesar números
function procesarNumeros(numeros) {
    let resultado = [];

    for (let i = 0; i < numeros.length; i++) {
        let cuadrado = numeros[i] * numeros[i];
        resultado.push(cuadrado);
    }

    return resultado;
}

// Llamada a la función
let numeros = [1, 2, 3, 4, 5];
let cuadrados = procesarNumeros(numeros);
console.log("Cuadrados:", cuadrados);

// Interacción con el DOM
const lista = document.getElementById("lista");
```

Ahora podemos incluir breakpoints en VSC y lanzar el depurador.