



Universitat Oberta  
de Catalunya

# Bases de JavaScript

Front end web  
developer CFCC



**Unió Europea**  
**Fons social europeu**  
L'FSE inverteix en el teu futur



Generalitat de Catalunya  
Consorci per a la Formació Contínua  
de Catalunya

- ❖ Introducció a JavaScript (3)
- ❖ Fundamentos (24)
- ❖ Funciones (82)
- ❖ ES5 vs ES6 (114)
- ❖ Objetos (145)
- ❖ String, Number (167)
- ❖ Map, Date, Math (202)
- ❖ Arrays (I) (230)
- ❖ Arrays (II) (268)
- ❖ DOM - interacció JS con HTML y CSS (291)

# Introducción

¿Sabías que...?

El pájaro del logo de Twitter se llama Larry



HTML + CSS = maquetación

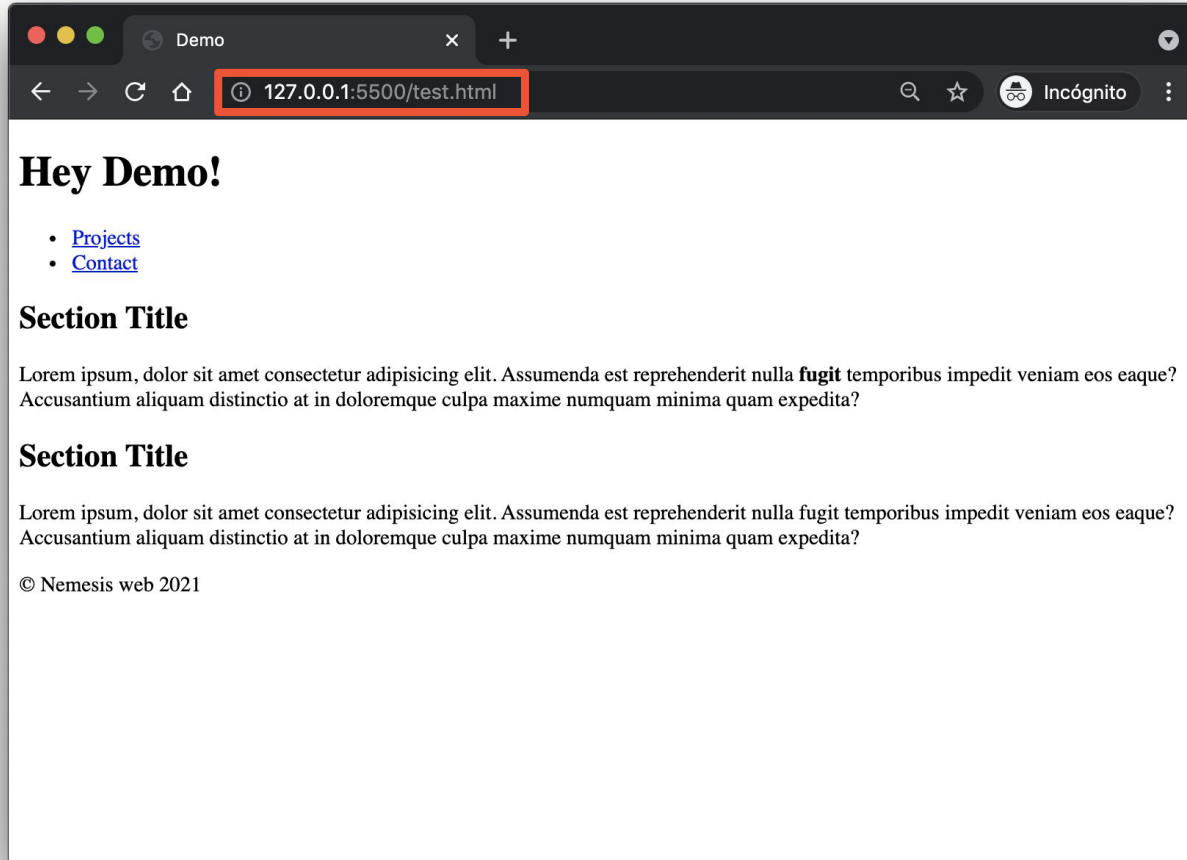
HTML + CSS + JS = front-end

JavaScript y Java son 2 lenguajes distintos

JavaScript es un lenguaje interpretado, mientras que Java es un lenguaje compilado.

JavaScript se puede ejecutar directamente en el navegador web, pero si hay algún error aparecerá durante la ejecución.

Java hay que compilarlo, de forma que si hay algún error en el código se verá en el momento de compilarlo. Una vez compilado ya se podrán ejecutar los programas.



Igual que con HTML, se recomienda usar VS Code con el plugin “Live Server” para crear un entorno con un servidor virtual.

JavaScript y Java son 2 lenguajes distintos

JavaScript es un lenguaje interpretado, mientras que Java es un lenguaje compilado.

JavaScript se puede ejecutar directamente en el navegador web, pero si hay algún error aparecerá durante la ejecución.

Java hay que compilarlo, de forma que si hay algún error en el código se verá en el momento de compilarlo. Una vez compilado ya se podrán ejecutar los programas.



HTML y CSS son la base de una página web y sólo con ellos se pueden crear webs.

JavaScript añade una capa de funcionalidades que aumenta muchísimo las posibilidades.

En sus orígenes se podía ejecutar en los navegadores web para poder ampliar la interacción de los usuarios con las web y darles más funcionalidades.

Actualmente los navegadores modernos soportan las últimas versiones de JS

También se pueden ejecutar JS en el servidor (backend) con [Node.js](https://nodejs.org/)

Todos los navegadores modernos interpretan el código JavaScript integrado en las páginas web. Para interactuar con una página web se provee al lenguaje JavaScript de una implementación del **Document Object Model** (DOM).

Eso significa que desde JS podemos interactuar con los elementos HTML y modificarlo. También se puede acceder a los estilos y modificar el CSS.

JS se puede ejecutar de forma similar a CSS: insertando el código directamente en el mismo documento HTML (inline), o crear un documento con extensión .js y llamarlo desde HTML.

Si se hace inline, todo el código JS debe ir dentro entre los tags:  
`<script> ... </script>`

Si se tiene uno o varios documentos de JS, se pueden llamar desde el documento HTML con:

`<script src="ruta/nombrescript.js"></script>`

## Script inline directament en HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>JavaScript</h1>
  <script>
    console.log('Hello World!')
  </script>
</body>
</html>
```

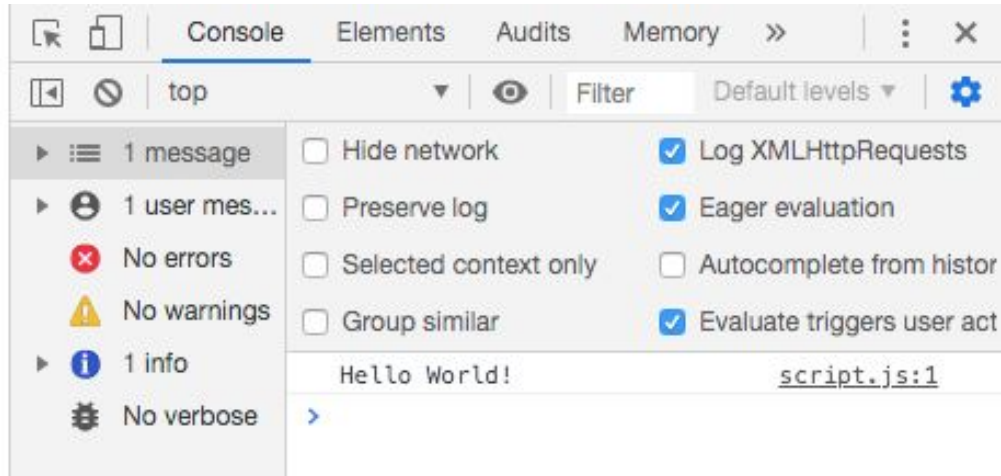
Si ejecutamos el código anterior en el navegador veremos que sólo aparece el texto que tenemos en:

```
<h1>JavaScript</h1>
```

Pero si abrimos la consola del navegador, en la pestaña Console, veremos que aparece el texto que hemos escrito en:

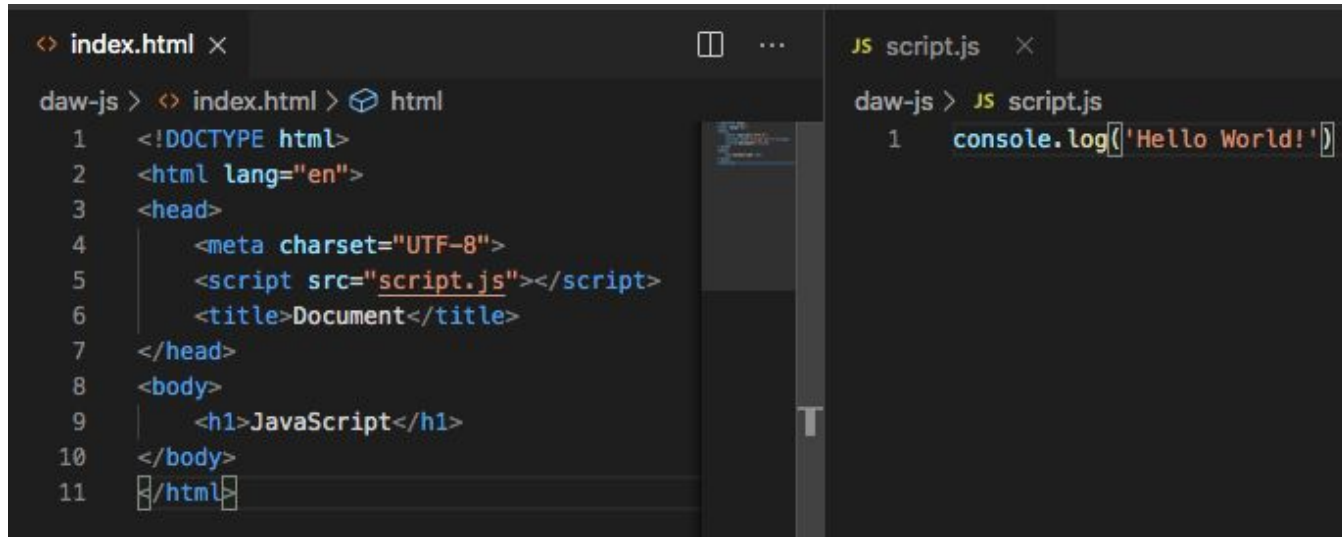
```
console.log('Hello World!');
```

## JavaScript



La instrucción `console.log(...)` es la forma más rápida de debugar el código y ver si el script se está ejecutando, o para mostrar los valores del código que estamos ejecutando

## HTML + fichero JS



```
<> index.html x
daw-js > <> index.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <script src="script.js"></script>
6    <title>Document</title>
7  </head>
8  <body>
9    <h1>JavaScript</h1>
10 </body>
11 </html>

JS script.js x
daw-js > JS script.js
1  console.log('Hello World!')
```



Si tenemos un fichero script y lo llamamos desde HTML, aunque puede hacerse, la llamada no tiene que hacerse obligatoriamente desde tag `<head></head>`

Hay que tener en cuenta que el script se ejecutará en el orden en el que esté puesto dentro del HTML

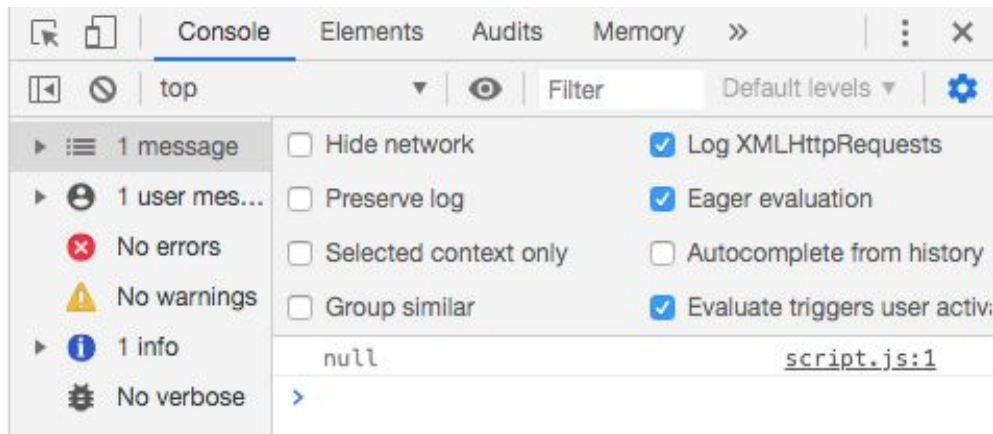
Si desde JS queremos acceder al elemento `<h1>`, pero llamamos al script antes del elemento, si abrimos la consola veremos que aparece: `null`

Eso se debe a que desde el script se está intentando acceder a un elemento que todavía no se ha cargado en el DOM

```
index.html x
daw-js > <> index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <script src="script.js"></script>
9   <h1 id="title">JavaScript</h1>
10 </body>
11 </html>

JS script.js x
daw-js > JS script.js
1 console.log( document.getElementById('title') );
```

## JavaScript



En cambio, si llamamos al script después del elemento, si abrimos la consola veremos que aparece: `<h1 id="title">JavaScript</h1>`

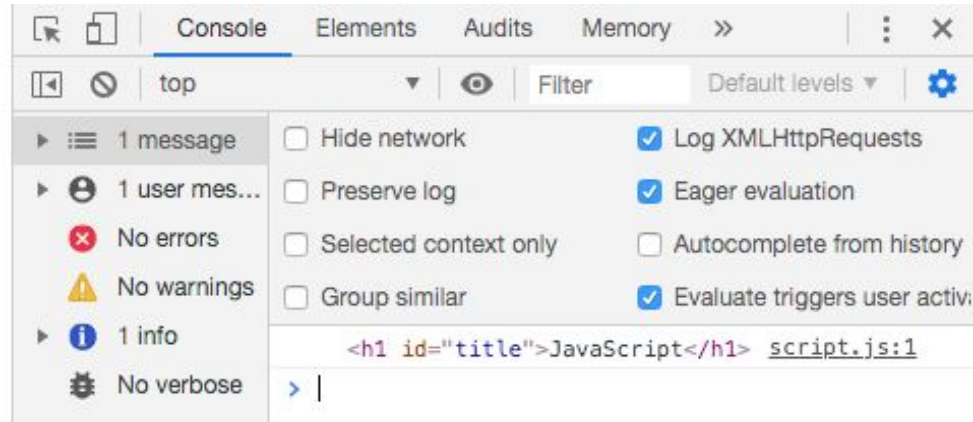
Como el elemento al `<h1>` al que hacemos referencia ya se ha cargado, podemos acceder a él sin problema

Por eso, es muy importante el orden de llamada de los scripts.

```
index.html x
daw-js > <> index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <h1 id="title">JavaScript</h1>
9   <script src="script.js"></script>
10 </body>
11 </html>

JS script.js x
daw-js > JS script.js
1 console.log( document.getElementById('title') );
```

## JavaScript



Si usamos ficheros JS, es recomendable llamarlos en el footer del HTML, justo antes de cerrar el tag `</body>`

De esta forma nos ahorramos errores, ya que todas la referencias a elementos de nuestro HTML van a devolver resultado.

# Fuentes

<https://es.wikipedia.org/wiki/JavaScript>

<https://www.w3schools.com/js/default.asp>

# Fundamentos



¿Sabías que...?

**El 7% de los adultos norteamericanos cree que el chocolate con leche proviene de vacas marrones.**



## Tag `<noscript>`

Es posible que algún usuario tenga desactivado el JavaScript del navegador.

Si se diera el caso, y nuestra web o aplicación requiere el uso de JS, se puede utilizar el tag `<noscript>...</noscript>`.

Todo lo que vaya dentro de ese tag o etiqueta, se mostrará únicamente cuando JavaScript esté deshabilitado.

Puede ser útil para mostrar un mensaje del tipo: debes habilitar JavaScript para poder ver bien esta página.

También es una buena práctica mostrar un contenido alternativo, aunque igual ni tan completo ni con funcionalidades tan potentes como podrían ser usando JS, pero por lo menos dar una alternativa a una página sin contenido.

JavaScript se ejecuta en el navegador, pero a no ser que tengamos alguna interacción con el front-end, no veremos el resultado del código ejecutado.

Para ello son muy útiles los métodos de **console**, que permiten mostrar información a través de la consola del navegador.

El más conocido probablemente sea `console.log()`, pero hay [muchos más](#).

```
console.log('hello console');
```

En JS hay 2 tipos de valores: **literales** y **variables**

**Literales:** son valores fijos que no cambian. Pueden ser números (entero o decimales con un punto) o textos (escritos entre comillas simples `'...'` o dobles `"..."`)

**Variables:** se usan para almacenar valores. Como su propio nombre indica, su valor puede variar a largo del programa.

Hay 3 tipos de variables: **var**, **let** y **const**. Más adelante veremos las diferencias. Por el momento, usaremos **var** para los ejemplos.

Para trabajar con variables, hay que **declararlas** e **inicializarlas**

## Declaración

Para declarar una variable sólo hay que indicar que es una variable y a continuación el nombre que queremos darle:

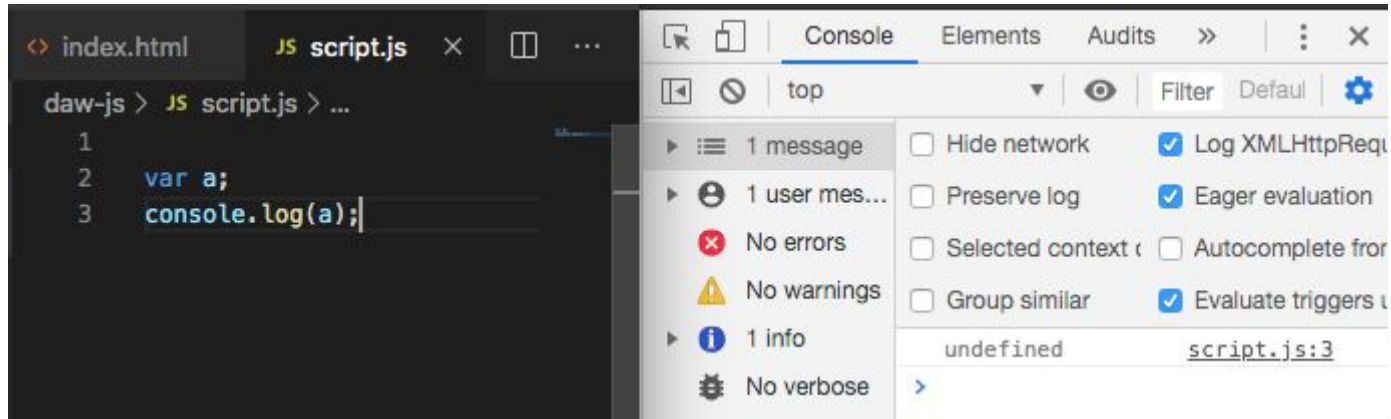
```
var a;
```

Si ejecutamos el siguiente código:

```
var a;  
console.log(a);
```

veremos que el valor que muestra es **undefined**. Eso significa que la variable está **declarada pero no inicializada**.

# JavaScript



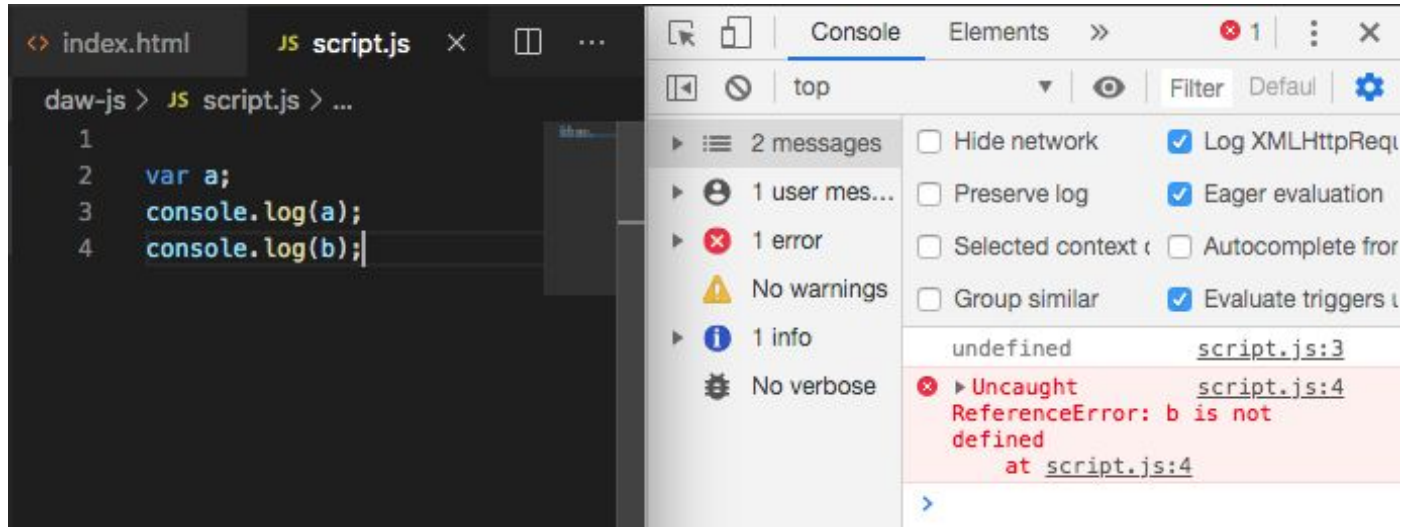


En cambio, si ejecutamos el siguiente código:

```
var a;  
console.log(a);  
console.log(b);
```

veremos que el primer valor es **undefined**, pero el segundo nos lanza un mensaje de error. Eso se debe a que **a** está declarada (aunque no tenga valor), pero **b** no está declarada

# JavaScript

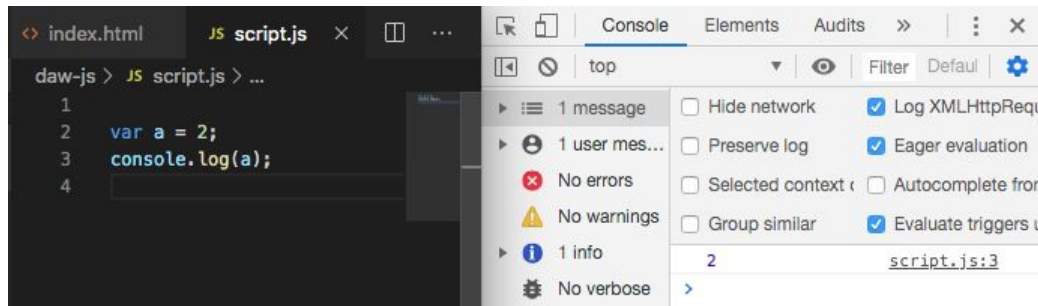


## Inicialización

Una vez se ha declarado la variable ya se le puede asignar un valor. Ese valor es el valor inicial, pero puede ir cambiando a lo largo del programa.

La inicialización puede hacerse en el mismo momento de la declaración, o más adelante.

```
var a;  
a = 2;  
0  
var a = 2;
```



También se puede hacer una declaración múltiple de las variables y luego su inicialización

```
var a, b, c;
```

```
a = 2;
```

```
b = 11;
```

```
c = 32;
```

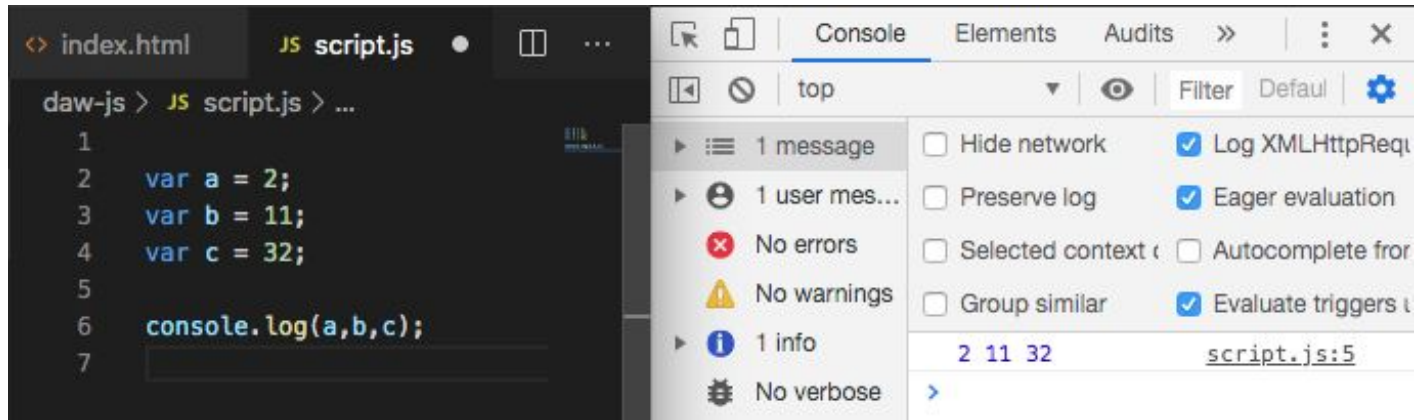
Aunque algunos recomiendan declarar cada variable por separado, para facilitar la legibilidad.

```
var a = 2;
```

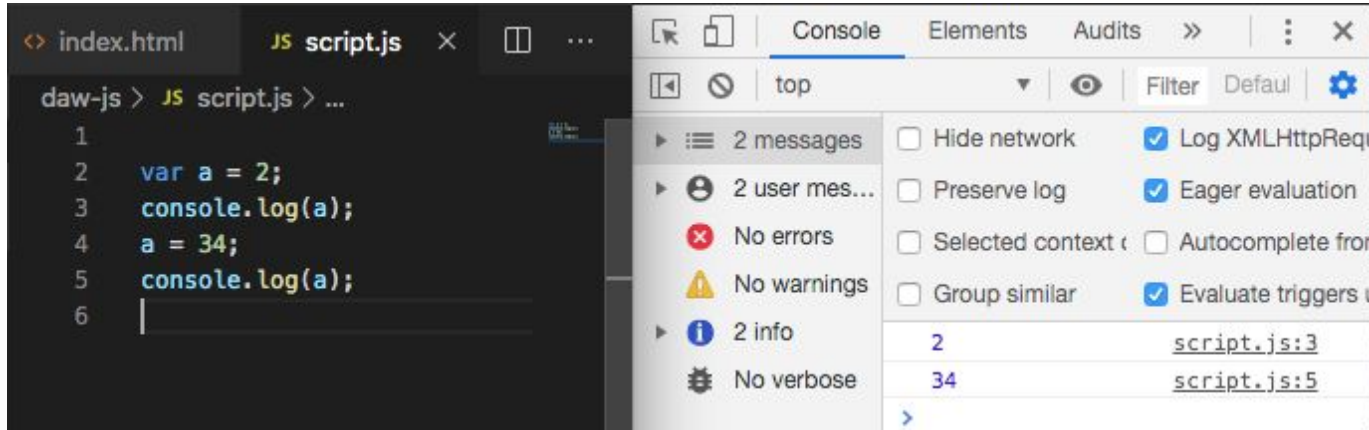
```
var b = 11;
```

```
var c = 32;
```

En un mismo console.log se pueden mostrar los valores de distintas variables



Como su propio nombre indica, las variables pueden cambiar de valor una vez declaradas e inicializadas.



The screenshot shows a web browser's developer console with the 'Console' tab selected. On the left, a code editor displays the following JavaScript code:

```
1  
2 var a = 2;  
3 console.log(a);  
4 a = 34;  
5 console.log(a);  
6
```

On the right, the console output shows two messages:

- 2 script.js:3
- 34 script.js:5

The console also displays summary statistics: 2 messages, 2 user messages, 0 errors, 0 warnings, 2 info, and 0 verbose. The 'Log' button is visible at the bottom of the console.

## scope

el concepto scope hace referencia al ámbito de uso de una variable. Puede llamarse también **ámbito** o **contexto**. Existen 2 tipos de scope: **global** y **local**.

Las variables declaradas en el ámbito global, pueden utilizarse en cualquier parte de nuestro código, mientras que las locales no.

## scope

Un ejemplo de una variable local, podría ser cualquier variable declarada **dentro** de un bloque { ... }\*. Desde fuera del bloque, no podemos acceder a dicha variable, pero sí podríamos acceder a una variable global (declarada **fuera** del bloque).

\* un bloque { } puede ser una función, un if, un loop, etc.



En el siguiente ejemplo, se declara una variable **global** **a**.

Se declara una función en la que se declara una variable **local** **b**. Dentro de la misma función se usan **a** y **b**.

Se hace una llamada a la función para que se ejecute.

Finalmente se hace una llamada fuera de la función a las variables **a** y **b**.

Se produce un error, ya que **b** sólo puede usarse dentro de la función.

The screenshot shows a web browser's developer console with the 'Console' tab selected. On the left, a code editor displays the following JavaScript code:

```
<> index.html JS script.js • [ ] ...  
daw-js > JS script.js > ...  
1  
2 var a = 2;  
3  
4 function demo() {  
5     var b = 3;  
6     console.log(a);  
7     console.log(b);  
8 }  
9  
10 demo();  
11  
12 console.log(a);  
13 console.log(b);  
14
```

The console on the right shows a summary of messages: 4 messages, 3 user messages, 1 error, no warnings, 3 info messages, and no verbose messages. The error message is expanded and highlighted in red:

```
Uncaught ReferenceError: b is not defined  
    at script.js:13
```

Below the error message, the console shows the execution flow with line numbers and file names:

```
2 script.js:6  
3 script.js:7  
2 script.js:12  
Uncaught ReferenceError: b is not defined at script.js:13
```

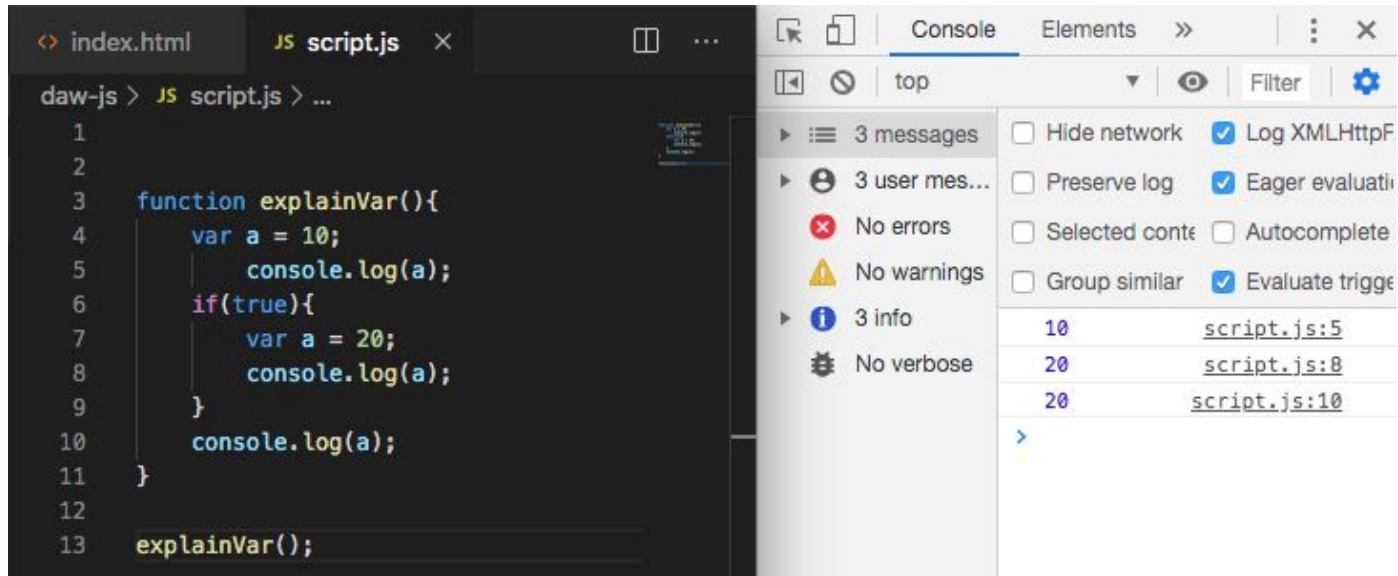
The console also includes a settings panel on the right with various options like 'Hide network', 'Log XMLHttpRequest', 'Preserve log', 'Eager evaluation', 'Selected context', 'Autocomplete from', 'Group similar', and 'Evaluate triggers'.

## Var

Las variables **var**, una vez declaradas e inicializadas se pueden ejecutar dentro del scope dónde han sido declaradas, però también puede accederse a ellas desde dentro de los elementos de bloque { ... }

En el siguiente ejemplo se están declarando 2 variables **a**, y una sobrescribe a la otra pese a estar en 2 contextos distintos (la segunda está dentro de un bloque { ... }, en ese caso un **if**.)

Eso nos puede llevar fácilmente a errores si no cuidamos el código, ya que podemos estar creando variables globales y modificarlas sin darnos cuenta.



The screenshot shows a web browser's developer console with the 'Console' tab selected. The left pane displays the source code of 'script.js' with line numbers 1 through 13. The right pane shows the console output, which includes three log messages: the value 10 at line 5, the value 20 at line 8, and the value 20 at line 10. The console also shows that there are no errors, warnings, or verbose messages.

```
1  
2  
3 function explainVar(){  
4     var a = 10;  
5     console.log(a);  
6     if(true){  
7         var a = 20;  
8         console.log(a);  
9     }  
10    console.log(a);  
11 }  
12  
13 explainVar();
```

Console output:

- 10 script.js:5
- 20 script.js:8
- 20 script.js:10

## Operadores aritméticos

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation ( <u>ES2016</u> )
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

fuelle: w3schools.com

## Asignación de operaciones

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

fuelle: w3schools.com

# JavaScript

The screenshot shows a web browser's developer console. On the left, the code editor displays a JavaScript file named `script.js` with the following code:

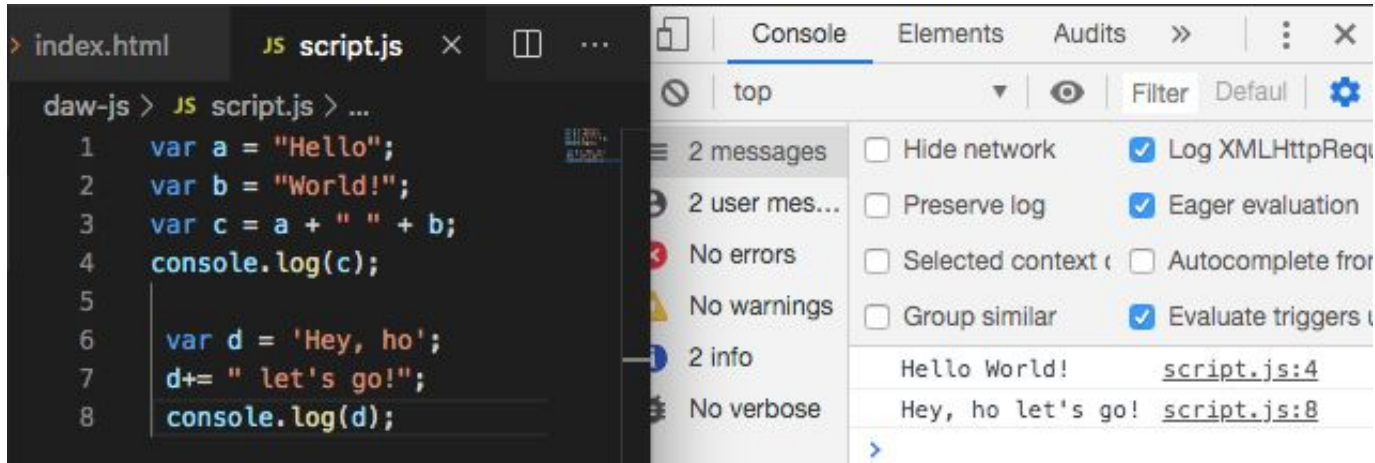
```
1 var a = 10;  
2 a = a+5;  
3 console.log(a);  
4  
5 var b = 10;  
6 b+=5;  
7 console.log(b);  
8
```

On the right, the console panel is open, showing the execution results. It displays two log messages:

- 15 `script.js:3`
- 15 `script.js:7`

The console also shows a summary of messages: 2 messages, 2 user messages, No errors, No warnings, 2 info, and No verbose. The console settings are visible on the right, including options for Hide network, Log XMLHttpRequests, Preserve logs, Eager evaluation, Selected content, Autocomplete, Group similar, and Evaluate in console.

El operador `+` también puede usarse para concatenar strings (cadenas de caracteres)



The screenshot shows a web browser's developer console with two panels. The left panel displays the source code of a file named `script.js`. The code defines variables `a` and `b` as strings, concatenates them into `c`, logs `c`, then defines `d` as a string and concatenates `" let's go!"` to it, logging `d`. The right panel shows the console output, which lists two messages: "Hello World!" from `script.js:4` and "Hey, ho let's go!" from `script.js:8`. The console also shows settings for the log, including "Log XMLHttpRequests" and "Eager evaluation" checked.

```
daw-js > JS script.js > ...  
1 var a = "Hello";  
2 var b = "World!";  
3 var c = a + " " + b;  
4 console.log(c);  
5  
6 var d = 'Hey, ho';  
7 d+= " let's go!";  
8 console.log(d);
```

Console  
top  
2 messages  
2 user mes...  
No errors  
No warnings  
2 info  
No verbose

Hide network  
Preserve log  
Selected context (c  
Group similar  
Log XMLHttpRequests  
Eager evaluation  
Autocomplete from  
Evaluate triggers (

Hello World! script.js:4  
Hey, ho let's go! script.js:8  
>



## Operadores de comparación

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

fuelle: w3schools.com

Las comparaciones con `==`, `===`, `!=`, `!==` puede ser algo confusas al principio.

`==` y `!=` comparan sólo los **valores** de los parámetros

`===` y `!==` comparan los **valores y el tipo** de los parámetros

The screenshot displays a web browser's developer console with the 'Console' tab selected. The left pane shows the source code of 'script.js' with line numbers 1 through 33. The right pane shows the console output, which includes 6 messages: 6 user messages, 0 errors, 0 warnings, and 6 info messages. The info messages are expanded, showing the following log entries:

- a: number (script.js:5)
- b: string (script.js:6)
- c: number (script.js:7)
- primero: iguales (script.js:13)
- segundo: distintos (script.js:25)
- tercero: iguales (script.js:30)

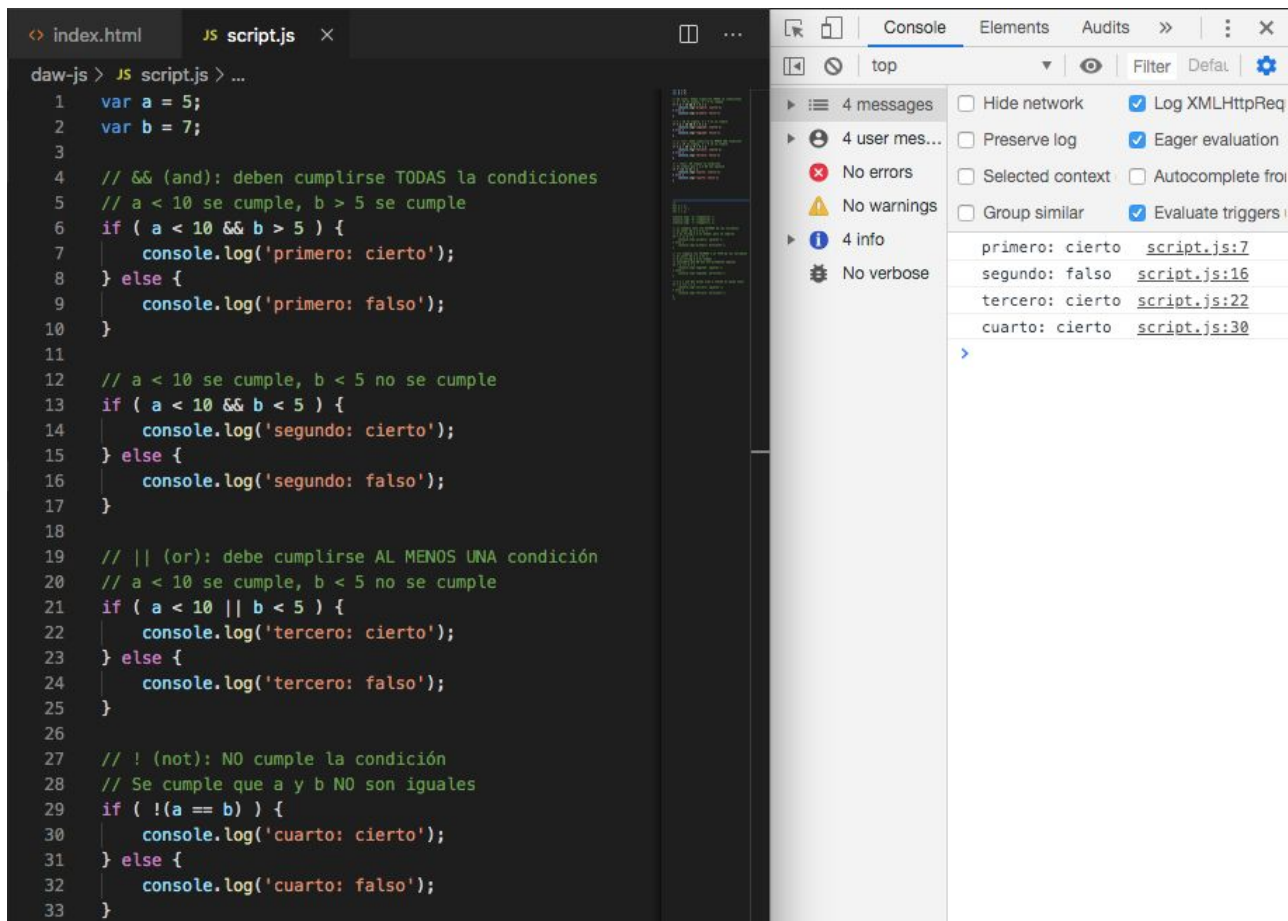
The code in 'script.js' defines three variables: `a` (number 1), `b` (string '1'), and `c` (number 1). It then logs the types of `a`, `b`, and `c` using `console.log`. It also performs three comparisons: `a == b` (true), `a === b` (false), and `a === c` (true), logging the results of each comparison.

```
1 var a = 1;
2 var b = '1';
3 var c = 1;
4
5 console.log( 'a:'+typeof(a) );
6 console.log( 'b:'+typeof(b) );
7 console.log( 'c:'+typeof(c) );
8
9 // == compara sólo los VALORES de las variables
10 // el valor de a y b es 1
11 // a es string y b es number pero no importa
12 if ( a == b ) {
13   console.log('primero: iguales');
14 } else {
15   console.log('primero: distintos');
16 }
17
18 // === compara los VALORES y el TIPO de las variables
19 // el valor de a y b es 1
20 // a es string y b es number
21 // considera que NO son estrictamente iguales
22 if ( a === b ) {
23   console.log('segundo: iguales');
24 } else {
25   console.log('segundo: distintos');
26 }
27
28 // a y c son del mismo tipo y tienen el mismo valor
29 if ( a === c ) {
30   console.log('tercero: iguales');
31 } else {
32   console.log('tercero: distintos');
33 }
```

## Operadores lógicos

Operator	Description
&&	logical and
	logical or
!	logical not

fuelle: w3schools.com



The image shows a web browser's developer console with a JavaScript file named `script.js` open. The code in the file defines four variables (`a = 5`, `b = 7`) and uses logical operators (`&&`, `||`, `!`) to evaluate conditions. The console output shows four messages: `primero: cierto`, `segundo: falso`, `tercero: cierto`, and `cuarto: cierto`. The right sidebar of the console shows the 'Messages' tab with 4 messages, 4 user messages, no errors, no warnings, and 4 info messages. The 'Log' tab is selected, and the messages are listed with their corresponding line numbers in `script.js`.

```
index.html JS script.js x
```

```
daw-js > JS script.js > ...  
1 var a = 5;  
2 var b = 7;  
3  
4 // && (and): deben cumplirse TODAS la condiciones  
5 // a < 10 se cumple, b > 5 se cumple  
6 if ( a < 10 && b > 5 ) {  
7     console.log('primero: cierto');  
8 } else {  
9     console.log('primero: falso');  
10 }  
11  
12 // a < 10 se cumple, b < 5 no se cumple  
13 if ( a < 10 && b < 5 ) {  
14     console.log('segundo: cierto');  
15 } else {  
16     console.log('segundo: falso');  
17 }  
18  
19 // || (or): debe cumplirse AL MENOS UNA condición  
20 // a < 10 se cumple, b < 5 no se cumple  
21 if ( a < 10 || b < 5 ) {  
22     console.log('tercero: cierto');  
23 } else {  
24     console.log('tercero: falso');  
25 }  
26  
27 // ! (not): NO cumple la condición  
28 // Se cumple que a y b NO son iguales  
29 if ( !(a == b) ) {  
30     console.log('cuarto: cierto');  
31 } else {  
32     console.log('cuarto: falso');  
33 }
```

Console Messages:

- 4 messages
- 4 user messages
- No errors
- No warnings
- 4 info
- No verbose

Log Messages:

- primero: cierto [script.js:7](#)
- segundo: falso [script.js:16](#)
- tercero: cierto [script.js:22](#)
- cuarto: cierto [script.js:30](#)

## this

Es muy común ver esta palabra clave cuando trabajamos con JavaScript. Dependiendo del contexto de uso devuelve un valor u otro, pero básicamente **hace referencia al objeto al que pertenece**.

Tiene diferentes valores según donde se utilice:

- En un **método**, *this* se refiere al **objeto propietario**.
- Si se usa **solo** en medio del script, *this* se refiere al **objeto global**.
- En una **función**, *this* se refiere al **objeto global**.
- En un **evento**, *this* se refiere al **elemento que recibió el evento**.

**Nota:** todos estos conceptos se tratarán más adelante, sólo se comenta para empezar a familiarizarse con el concepto *this*.

# Condicionales



Las declaraciones condicionales sirven para **realizar diferentes acciones en base a una decisión**. En JavaScript tenemos las siguientes:

- `if`
- `else`
- `else if`
- `switch`

**if:** para especificar un bloque de código que se ejecutará si una condición es **verdadera**

```
if ( mood == 'good' ) {  
    console.log('😊')  
}
```

Si el bloque de código se puede escribir en una sola línea, no es necesario { ... }

```
if ( mood == 'good' ) console.log('😊')
```

**else:** para especificar un bloque de código que se ejecutará, si **la misma** condición es **falsa**

```
if ( mood == 'good' ) {  
    console.log('😊')  
} else {  
    console.log('😞')  
}
```

El **operador ternario** sirve para asignar un valor según se cumpla o no una condición. Es una forma de escribir una condición **if / else** pero reducido a una sólo línea.

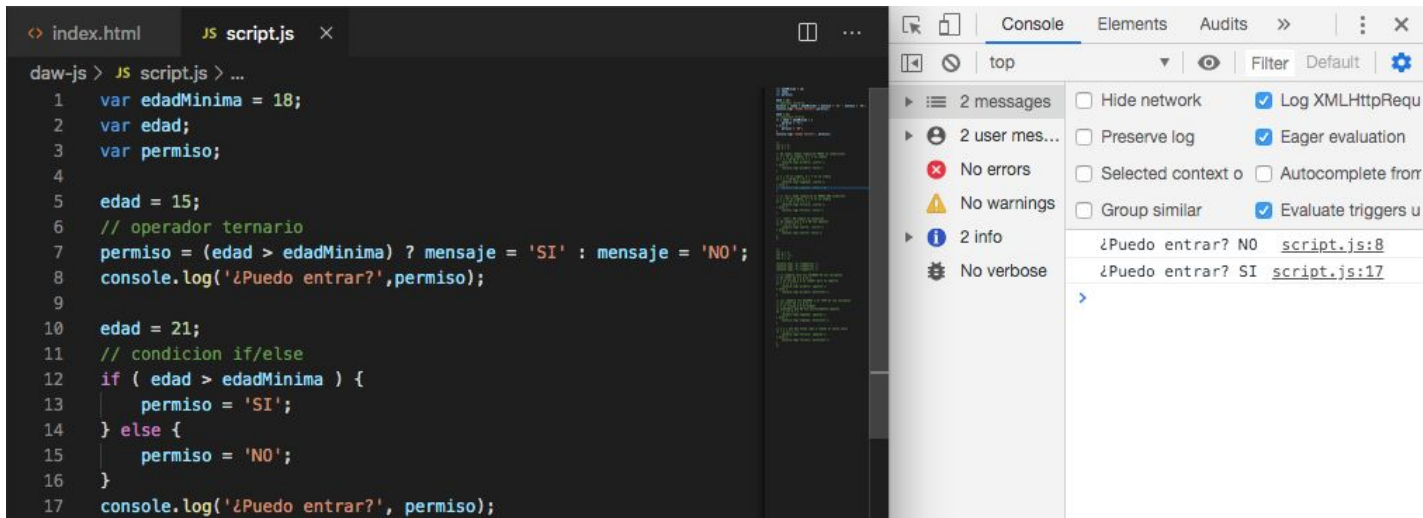
## **if/else:**

```
if ( condición ) {  
    valor_si_cumple  
} else {  
    valor_si_no_cumple  
}
```

## **operador ternario:**

```
variable = (condición) ? valor_si_cumple : valor_si_no_cumple
```

# JavaScript



The screenshot shows a web browser's developer console with two panels. The left panel displays the JavaScript code in `script.js`, and the right panel shows the console output.

**JavaScript Code:**

```
1  var edadMinima = 18;
2  var edad;
3  var permiso;
4
5  edad = 15;
6  // operador ternario
7  permiso = (edad > edadMinima) ? mensaje = 'SI' : mensaje = 'NO';
8  console.log('¿Puedo entrar?', permiso);
9
10 edad = 21;
11 // condicion if/else
12 if ( edad > edadMinima ) {
13   permiso = 'SI';
14 } else {
15   permiso = 'NO';
16 }
17 console.log('¿Puedo entrar?', permiso);
```

**Console Output:**

The console shows two log messages:

- Log 1: `¿Puedo entrar? NO` (from `script.js:8`)
- Log 2: `¿Puedo entrar? SI` (from `script.js:17`)

The right panel also includes a settings menu with the following options:

- ☐ Hide network
- ☒ Log XMLHttpRequests
- ☐ Preserve log
- ☒ Eager evaluation
- ☐ Selected context only
- ☐ Autocomplete from
- ☐ Group similar
- ☒ Evaluate triggers

**else if:** para especificar una **nueva condición** para probar, **si la primera condición es falsa**

```
if ( mood == 'good' ) {  
    console.log('😊')  
} else if ( mood == 'sad' ) {  
    console.log('😞')  
} else {  
    console.log('😐')  
}
```

**switch:** para especificar varios bloques de código alternativos que se ejecutarán en base a la condición de cada bloque

```
switch (mood) {  
  case 'happy':  
    console.log('😊')  
    break;  
  case 'sad':  
    console.log('😞')  
    break;  
  case 'crazy':  
    console.log('😜')  
    break;  
  default:  
    console.log('😊')  
}
```

# Alert



Hasta ahora, en todos los ejemplos que hemos visto, hemos introducido los datos directamente por código y los hemos visualizado a través de la consola.

Más adelante veremos cómo integrarlo todo con el front-end y como modificar el HTML, pero de momento veamos una forma mostrar datos sin tener que depender de la consola.

**Alert** es una función de JavaScript que muestra un popup con la información que queramos.

Es un popup **bloqueante**, es decir, que hasta que no se acepta, se para la ejecución del código de JS. Es por eso, que para debugar no es muy buena opción y siempre es mejor usar la consola.

Pero puede ser útil para mostrar algún mensaje al usuario. Además, en caso de querer mostrar un objeto o array entero en un alert, no se podría desplegar, a diferencia de la consola.

Para usar un alert únicamente hay que llamar a la función **alert()** y pasarle el texto que se quiera mostrar al usuario.

```
alert('Hello JavaScript!');
```

# Alert

Los alerts son bloqueantes.

Hasta que no se acepte uno,  
no se lanzará el siguiente.

```
const cars = {  
  name: 'Ferrari',  
  color: 'red'  
}  
  
alert('Hola!');  
alert(cars);  
alert(cars.name+', '+cars.color);
```

Esta página dice

Hola!

Aceptar

Esta página dice

[object Object]

Aceptar

Esta página dice

Ferrari, red

Aceptar

# Prompt

**Prompt** es un popup parecido al alert, pero éste permite mostrar un mensaje y además tiene un campo para entrar información.

Esta información la podemos almacenar en una variable. Es una forma rápida para capturar información del usuario para hacer pruebas rápidas y no tener que crear un formulario.

Para usar un prompt únicamente hay que asignarlo como valor de una variable (que será la que recoja el valor entrado por el usuario) y escribir el texto que se mostrará:

```
let msg = prompt('Escribe tu nombre');
```

# Prompt

```
const cars = {};  
  
function addCar() {  
  let name = prompt('Car name');  
  let color = prompt('Car color');  
  cars.name = name;  
  cars.color = color;  
}  
  
addCar();  
  
alert(cars.name + ',' + cars.color);
```

Esta página dice

Car name

Cancelar Aceptar

Esta página dice

Car color

Cancelar Aceptar

Esta página dice

Ferrari,Red

Aceptar



# Funciones y Métodos

Una **función** es un **bloque de código** escrito para realizar un conjunto específico de tareas.

Podemos definir una función usando la palabra clave de **function**, seguida de su nombre y parámetros opcionales. El cuerpo de la función está encerrado entre llaves { }

```
function functionName(parameters) {  
    // Content  
}
```

- La función se ejecuta cuando algo la llama / invoca.
- El nombre puede contener letras, dígitos, signos de dólar, subrayado.
- Los parámetros se enumeran entre paréntesis después del nombre de la función.
- Los argumentos son valores que recibe una función cuando se invoca.
- Cuando se alcanza el objetivo o condición de retorno, el código deja de ejecutarse y retorna un valor.

```
var func = function(a, b) {  
    var sum = a + b;  
    return sum;  
}
```

```
console.log(func(1, 2));
```

```
// 3
```

Un **método** es una **propiedad de un objeto** que contiene una definición de función.

Los métodos son funciones almacenadas como propiedades de objeto.

```
object = {  
    methodName: function() {  
        // Content  
    }  
};  
  
object.methodName()
```

- Los métodos JavaScript son las acciones que se pueden realizar en objetos.
- Los objetos también se pueden llamar sin usar paréntesis.
- *this* se refiere al objeto propietario en un método.

```
const employee = {  
    empname: "Homer",  
    sector : "7G",  
    details : function() {  
        return this.empname +  
            " works in Sector" +  
            this.sector;  
    }  
};  
  
console.log(employee.details());  
// Homer works in Sector 7G
```

<b>Función (function)</b>	<b>Método (method)</b>
Se puede llamar directamente por su nombre	Debe llamarse a través del objeto al que pertenece, usando un punto ( . ) o corchetes ( [ ] ) y el nombre del método
Puede recibir y devolver datos	Opera sólo con los datos del objeto al que pertenece
Necesita recibir datos explícitamente como argumento	Implícitamente usa los datos del objeto, no necesita recibirlos como argumento
Puede existir por sí misma	Está asociada al objeto dónde se ha declarado



[Tatiana Molina: var, let y const. Dónde, cuándo y por qué](#)

[w3schools: Variables JavaScript](#)

[w3schools: Operadores JavaScript](#)

[w3schools: Comparaciones JavaScript](#)

<https://developer.mozilla.org/es/docs/Web/API/Console>

<https://www.geeksforgeeks.org/difference-between-methods-and-functions-in-javascript/>

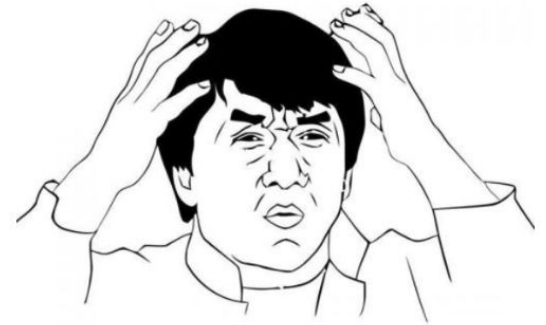
<https://emojipedia.org/>

```
language_attributes(); ?>
charset="<?php bloginfo( 'charset' ); ?>" />
name="viewport" content="width=device-width" />
wp_title( '|', true, 'right' ); ?> /title
rel="profile" href="http://gmpg.org/xfn/11" />
rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>" />
fruitful_get_favicon(); ?>
wp_head(); ?>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
esc_attr($theme_options['logo_position']);
```

# Funciones

¿Sabías que...?

Algunas serpientes submarinas respiran a través de su piel



# Funciones

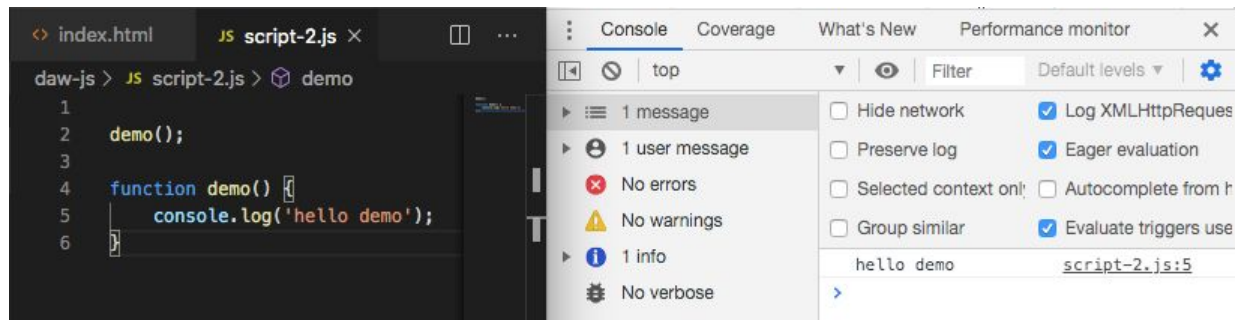
## Hoisting

Antes de seguir, es importante entender este concepto. El **hoisting** es un comportamiento de JavaScript que hace que internamente, la declaraciones de funciones se muevan al principio de su **scope** o ámbito de uso.

Es decir, que podemos llamar a una función y definirla más abajo, porque internamente JS la “subirá”.

Eso no es posible en todos los lenguajes de programación, ya que muchos requieren que en el flujo del código, primero se defina la función y después se llame

# Funciones



## Function

JavaScript, como la mayoría de lenguajes de programación, permite el uso de funciones.

Una función es un fragmento de código separado del hilo de código principal, que puede ser invocado en cualquier momento y cuyo propósito es resolver una tarea específica.

En JS se pueden declarar de 3 formas distintas.

## 1. Declaración

La primera forma de crear funciones es indicándolo con la palabra clave **function**, seguida del **nombre** que queremos darle, 2 **paréntesis** en los que se pueden pasar parámetros opcionalmente, y unas **claves** entre las que irá el código propio de la función. Si la función debe retornar algún valor, debe indicarse con la palabra reservada **return**.

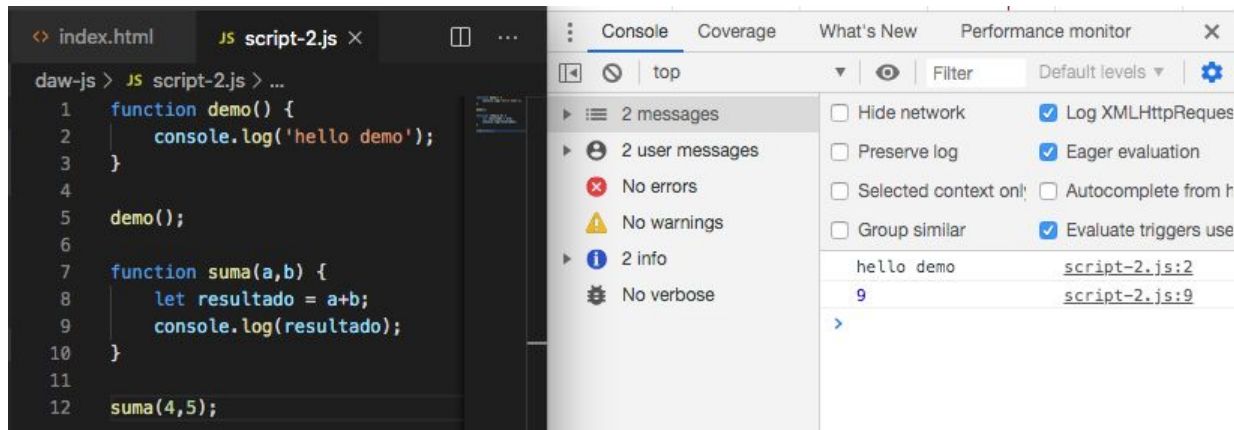
Para invocarla o llamarla, únicamente hay que escribir el nombre de la función seguido de los paréntesis (entre los que se pueden pasar parámetros, opcionalmente)



```
function demoFunction ( param ) {  
  
    // aquí se hace la magia con param  
  
    return resultado  
  
}
```

En este caso, la función **demo** no admite parámetros y muestra un mensaje por consola.

En cambio, la función **suma** admite 2 parámetros, los suma y muestra el resultado por consola. En el momento de invocarla, se le pasan los valores a sumar.



The screenshot shows a web browser's developer console with the following content:

```
daw-js > JS script-2.js > ...  
1 function demo() {  
2   console.log('hello demo');  
3 }  
4  
5 demo();  
6  
7 function suma(a,b) {  
8   let resultado = a+b;  
9   console.log(resultado);  
10 }  
11  
12 suma(4,5);
```

The console output shows two messages:

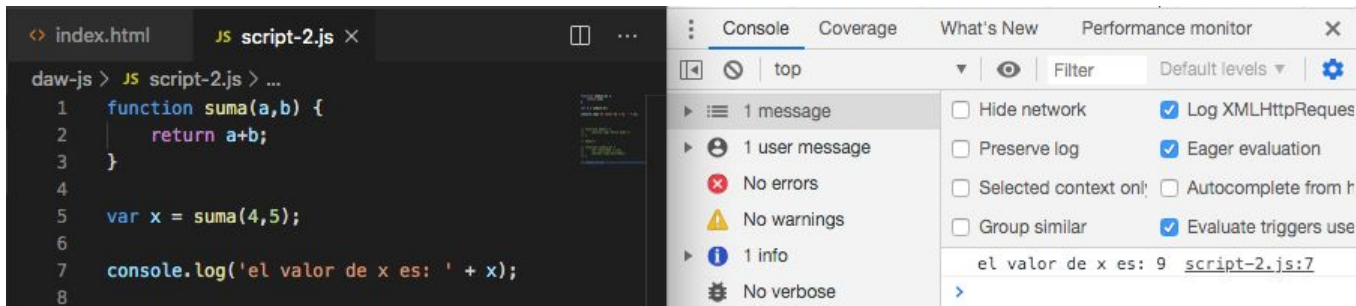
- hello demo (script-2.js:2)
- 9 (script-2.js:9)

The console also displays a summary of messages: 2 messages, 2 user messages, No errors, No warnings, 2 info, and No verbose.

Una función también puede ser invocada como valor de una variable. En ese caso, la variable se inicializa con el valor de retorno de la función.

La función recibe 2 parámetros, los suma y retorna el resultado.

La variable **x** toma el valor de retorno de la función.



```
<> index.html JS script-2.js x
daw-js > JS script-2.js > ...
1 function suma(a,b) {
2   return a+b;
3 }
4
5 var x = suma(4,5);
6
7 console.log('el valor de x es: ' + x);
8
```

The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following messages:

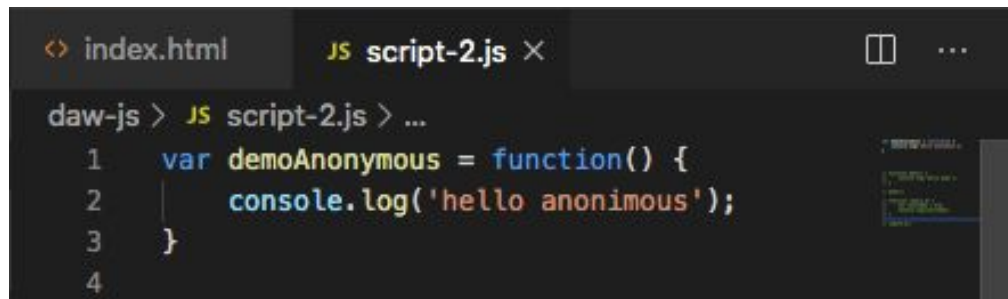
- 1 message
- 1 user message
- No errors
- No warnings
- 1 info
- No verbose

The 'info' message is expanded, showing the text: `el valor de x es: 9` and the source location: `script-2.js:7`. The right-hand pane of the developer tools shows various settings, including 'Log XMLHttpRequests', 'Eager evaluation', and 'Evaluate triggers use'.

## 2. Expresiones (anonymous function)

Una segunda manera de declarar la funciones es con las denominadas funciones anónimas.

Se llaman así porque se declarar sin nombre. Y son declaradas directamente como valor de una variable. Dicha variable se inicializa con el valor de retorno de la función.



```
<> index.html  JS script-2.js x  []  ...  
daw-js > JS script-2.js > ...  
1  var demoAnonymous = function() {  
2    console.log('hello anonymous');  
3  }  
4
```

Las funciones **anónimas** asignadas a una variable se invocan llamando a la variable

```
var x = function(a) { console.log(a) }  
x('Fry')  
// Fry
```

Las funciones **con nombre** se pueden llamar por si solas o asignarlas a una variable, en cuyo caso se ejecutan en ese momento.

```
function demo(a) { console.log(a) }  
var x = demo('Leela')  
// Leela
```

```
demo('Bender')  
// Bender
```

## 3. Arrow Functions

Es el tipo de funciones más nuevo de JavaScript, ya que se trata de una implementación de ES6.

Se trata de una forma más corta de escribirlas (no hace falta la palabra **function**).

También se pueden crear desde una variable y pueden recibir parámetros.

Si la función sólo ejecuta una sólo línea de código, se pueden omitir las claves.

**Nota:** los nombramos aquí, pero los trabajaremos en detalle más adelante

# Funciones

The screenshot shows a web browser's developer console with the following components:

- Code Editor:** Displays the following JavaScript code:

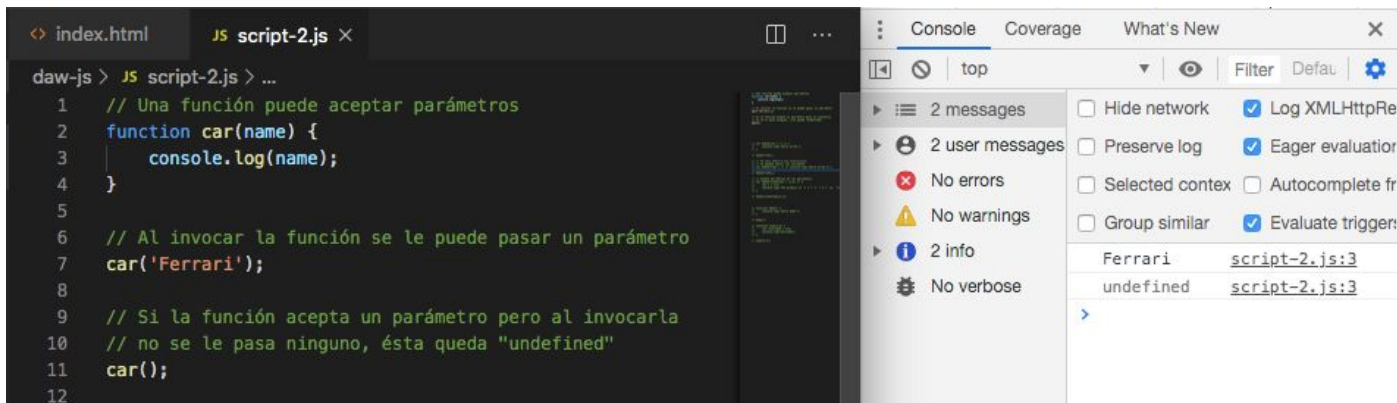
```
1 var demoArrow = () => {  
2   console.log('hello arrow');  
3 }  
4  
5 demoArrow();  
6  
7 // Si sólo efectúa una instrucción,  
8 // se pueden omitir los corchetes  
9 var demoArrow2 = () => console.log('hello arrow 2');  
10  
11 demoArrow2();  
12  
13 // Acepta parámetros en los paréntesis  
14 var demoArrowParams = (x,y) => {  
15   let z = x*y;  
16   console.log('The product of ' + x + ' * ' + y + ' is: ' + z);  
17 }  
18  
19 demoArrowParams(2,3);  
20
```
- Console Panel:** Shows the execution results of the code:
  - 3 messages
  - 3 user messages
  - No errors
  - No warnings
  - 3 info
  - No verbose
- Log Details:** The log shows three messages:
  - hello arrow (script-2.js:2)
  - hello arrow 2 (script-2.js:9)
  - The product of 2 \* 3 is: 6 (script-2.js:16)
- Settings:** The console settings are visible on the right, including options like "Log XMLHttpRequests", "Eager evaluation", and "Evaluate triggers user acti".



## Parámetros

Las funciones admiten parámetros, pero ¿qué pasa si se invoca una función y se le pasa ningún parámetro?

El parámetro queda **undefined**.



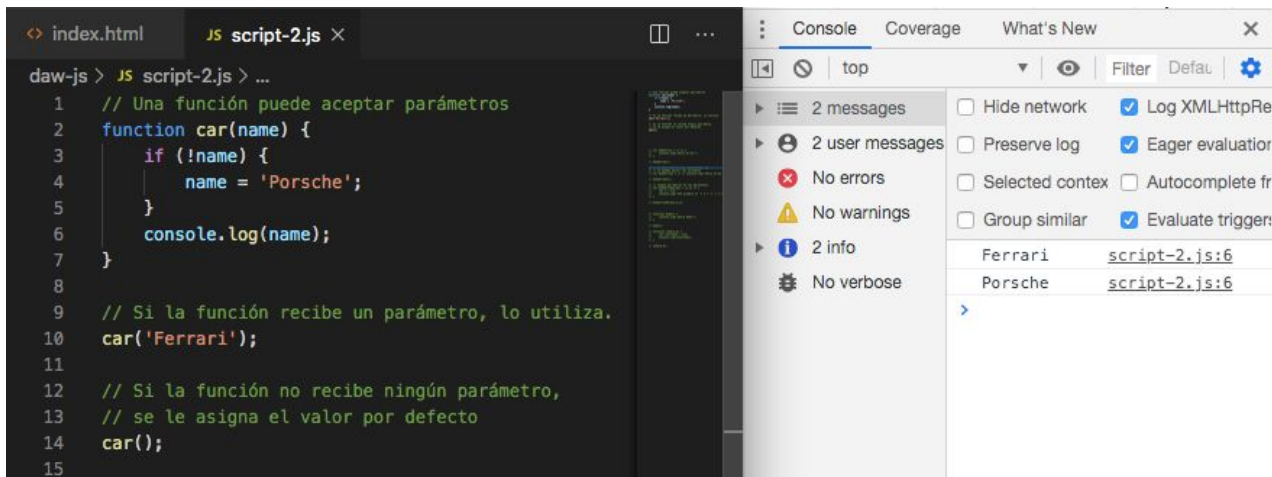
```
daw-js > JS script-2.js > ...
1 // Una función puede aceptar parámetros
2 function car(name) {
3   console.log(name);
4 }
5
6 // Al invocar la función se le puede pasar un parámetro
7 car('Ferrari');
8
9 // Si la función acepta un parámetro pero al invocarla
10 // no se le pasa ninguno, ésta queda "undefined"
11 car();
12
```

The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays two messages: 'Ferrari' and 'undefined'. The first message is an 'info' message from 'script-2.js:3', and the second is an 'info' message from 'script-2.js:3'. The code in the background defines a function 'car' that takes a parameter 'name' and logs it. The function is called twice: first with 'Ferrari' and then without any arguments.

## Parámetros

Los parámetros pueden inicializarse y asignarles un valor por defecto, en caso que no se les pase ninguno.

Puede hacerse desde la propia definición del parámetro, o controlarlo en el interior de la función.



```
index.html JS script-2.js x
daw-js > JS script-2.js > ...
1 // Una función puede aceptar parámetros
2 function car(name) {
3   if (!name) {
4     name = 'Porsche';
5   }
6   console.log(name);
7 }
8
9 // Si la función recibe un parámetro, lo utiliza.
10 car('Ferrari');
11
12 // Si la función no recibe ningún parámetro,
13 // se le asigna el valor por defecto
14 car();
15
```

Console Coverage What's New

2 messages 2 user messages No errors No warnings 2 info No verbose

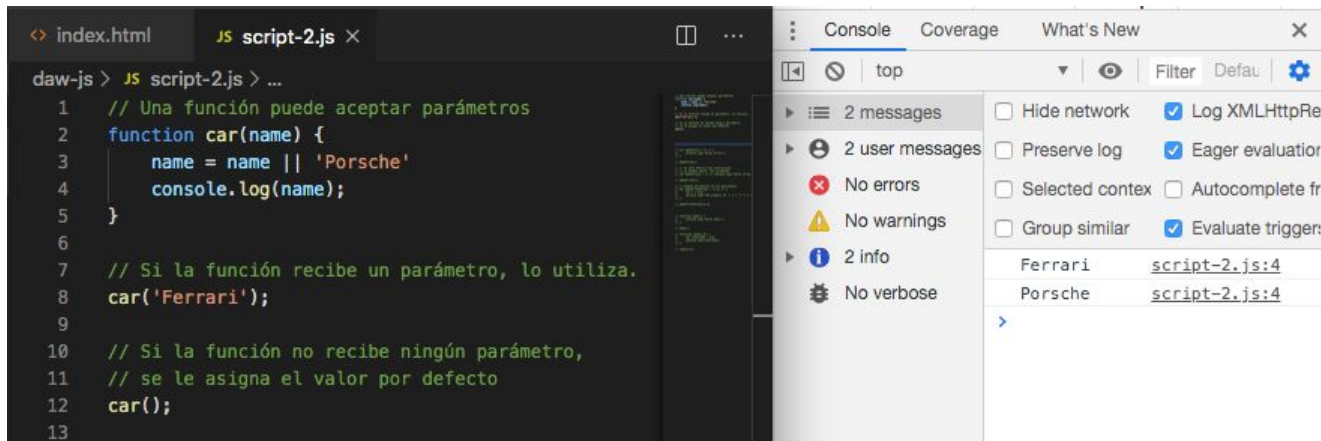
Hide network Log XMLHttpRequest Preserve log Eager evaluation Selected context Autocomplete fr Group similar Evaluate trigger

Ferrari script-2.js:6  
Porsche script-2.js:6

## Parámetros

Otra forma de inicializar los valores es con un condicional tipo:  
parámetro = parámetro || valor\_por\_defecto

Si el parámetro está definido, se le asigna su mismo valor; en caso contrario, le podemos asignar un valor



```
<> index.html JS script-2.js x
daw-js > JS script-2.js > ...
1 // Una función puede aceptar parámetros
2 function car(name) {
3     name = name || 'Porsche'
4     console.log(name);
5 }
6
7 // Si la función recibe un parámetro, lo utiliza.
8 car('Ferrari');
9
10 // Si la función no recibe ningún parámetro,
11 // se le asigna el valor por defecto
12 car();
13
```

Console Coverage What's New

top Filter Defau

2 messages

2 user messages

No errors

No warnings

2 info

No verbose

Hide network ☒ Log XMLHttpRequest

Preserve log ☐ Eager evaluation

Selected context ☐ Autocomplete fr

Group similar ☐ Evaluate trigger

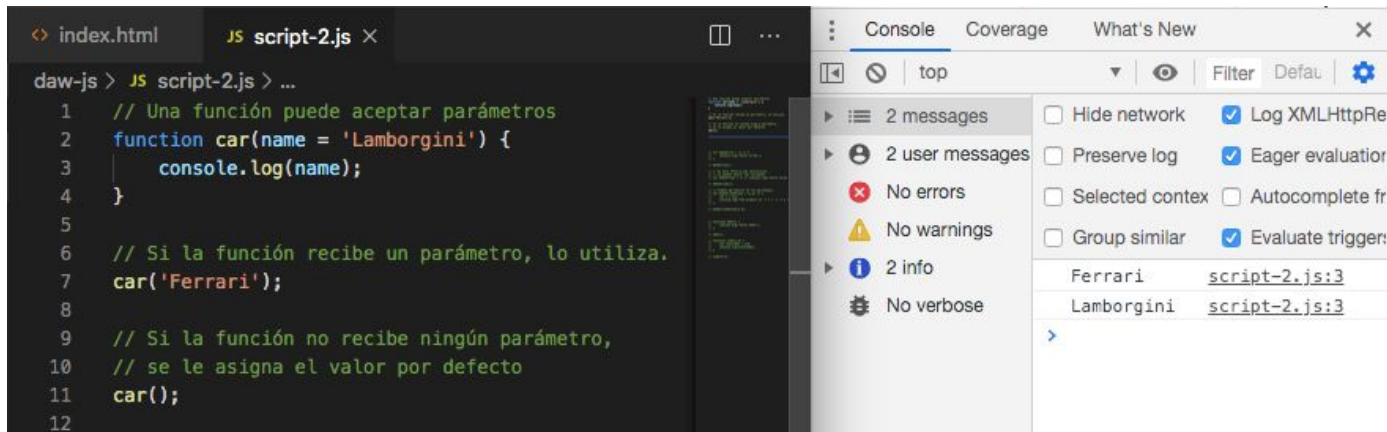
Ferrari script-2.js:4

Porsche script-2.js:4

## Parámetros

Finalmente, también se puede asignar el valor por defecto directamente desde la definición de la función.

Si la función acepta parámetros, y no recibe ninguno al ser invocada, se le asigna el valor por defecto ("Lamborghini" en el ejemplo). En cambio, si recibe algún parámetro al ser invocada, utilizará éste.



```
index.html JS script-2.js x
daw-js > JS script-2.js > ...
1 // Una función puede aceptar parámetros
2 function car(name = 'Lamborghini') {
3   console.log(name);
4 }
5
6 // Si la función recibe un parámetro, lo utiliza.
7 car('Ferrari');
8
9 // Si la función no recibe ningún parámetro,
10 // se le asigna el valor por defecto
11 car();
12
```

Console Coverage What's New

top Filter Defau

2 messages

2 user messages

No errors

No warnings

2 info

No verbose

Hide network ☒ Log XMLHttpRequest

Preserve log ☐ Eager evaluation

Selected context ☐ Autocomplete fr

Group similar ☒ Evaluate trigger

Ferrari script-2.js:3

Lamborghini script-2.js:3

## Funciones autoinvocadas

Hemos visto que las funciones, primero hay que declararlas y luego invocarlas

### **declaración**

```
function demo() { ... }
```

### **invocación**

```
demo()
```

## Funciones autoinvocadas

Existe una forma de invocar las funciones al mismo tiempo de su creación. Son las llamadas funciones autoinvocadas.

Su definición es:

```
(function() { ... })()
```

Una función autoinvocada se ejecuta en el mismo momento en que se crea. Su declaración puede resultar un poco confusa debido a la cantidad de paréntesis, que utiliza, pero es cuestión de acostumbrarse:

```
(function() { ... })()
```

```
(function() { ... })()
```

Los paréntesis en negro, engloban la propia función, incluidos los corchetes { ... } donde va el propio código que ejecutará la función

Los paréntesis azules son los que se pueden usar para pasar parámetros

Los paréntesis verdes son los que se encargan de invocar la función. Son los mismos que se utilizan para invocar una función cualquiera: demo()



# Funciones

The screenshot shows a web browser's developer console with the following content:

**Code Editor:**

```
daw-js > JS script-2.js > ...
1 // declaración
2 function demo() {
3   console.log('función invocada')
4 }
5 // invocación
6 demo();
7
8 // función autoinvocada
9 (function () {
10   console.log('función autoinvocada')
11 })()
12
13
```

**Console:**

- 2 messages
- 2 user messages
- No errors
- No warnings
- 2 info
- No verbose

**Log Details:**

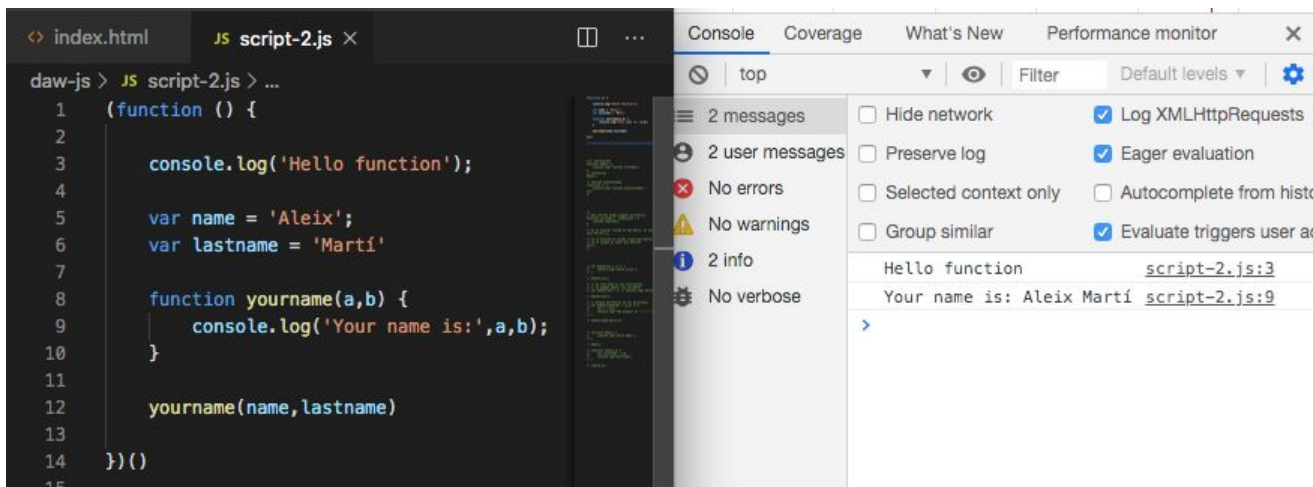
- función invocada `script-2.js:3`
- función autoinvocada `script-2.js:10`

**Settings:**

- ☐ Hide network
- ☐ Preserve log
- ☐ Selected context only
- ☐ Group similar
- ☒ Log XMLHttpRequest
- ☒ Eager evaluation
- ☐ Autocomplete from hi
- ☒ Evaluate triggers user

Una práctica habitual es englobar todo el código JavaScript dentro de una función autoinvocada. Dentro se pueden crear variables, otras funciones, etc.

De esta forma nos podemos asegurar que todas la variables y funciones estén dentro del mismo scope.



The screenshot shows a web browser with a dark-themed code editor. The code defines a self-invoking function (IIFE) that logs a message, declares variables, and calls a nested function. The browser's developer console is open, showing the execution results.

```
daw-js > JS script-2.js > ...
1  (function () {
2
3      console.log('Hello function');
4
5      var name = 'Aleix';
6      var lastname = 'Martí'
7
8      function yourname(a,b) {
9          console.log('Your name is:',a,b);
10     }
11
12     yourname(name,lastname)
13
14 })()
```

Console Coverage What's New Performance monitor

top Filter Default levels

2 messages

2 user messages

No errors

No warnings

2 info

No verbose

Hide network Log XMLHttpRequests

Preserve log Eager evaluation

Selected context only Autocomplete from histo

Group similar Evaluate triggers user ac

Hello function script-2.js:3

Your name is: Aleix Martí script-2.js:9

## Callback

Una función callback es una función que **se pasa como argumento** a otra función.

Esta técnica permite que una función llame a otra función.

Una función de callback se puede ejecutar después de que haya finalizado otra función.

```
function greeting(name) {  
    alert('Hello ' + name);  
}
```

```
function processUserInput(mycb) {  
    var yourname = prompt('Please enter your name.');
```

```
    mycb(yourname);  
}
```

```
processUserInput(greeting);
```

127.0.0.1:5500 dice

Please enter your name.

127.0.0.1:5500 dice

Hello Aleix

## Callback

Las funciones callback son especialmente útiles cuando dependen de alguna otra función que tarde un rato en devolver la información, como las funciones asíncronas.

# Funciones

```
function download(url) {  
    setTimeout(function() {  
        console.log('Downloading '+url+'...');  
    }, 3000);  
}  
  
function process(picture) {  
    console.log( 'Processing '+picture+'...');  
}  
  
var url = 'https://www.javascripttutorial.net/foo.jpg';  
download(url);  
process(url);  
  
// Processing https://javascripttutorial.net/foo.jpg  
// Downloading https://javascripttutorial.net/foo.jpg ...
```

# Funciones

```
function download(url, callback) {  
    setTimeout( function(){  
        console.log('Downloading '+url+'...');  
        // process the picture once it is completed  
        callback(url);  
    }, 3000);  
}  
  
function process(picture) {  
    console.log('Processing '+picture+'...');  
}  
  
let url = 'https://www.javascripttutorial.net/pic.jpg';  
download(url, process);  
  
// Downloading https://javascripttutorial.net/foo.jpg ...  
// Processing https://javascripttutorial.net/foo.jpg
```



<https://www.javascripttutorial.net/javascript-callback/>

[https://www.w3schools.com/js/js\\_callback.asp](https://www.w3schools.com/js/js_callback.asp)

<https://www.geeksforgeeks.org/difference-between-methods-and-functions-in-javascript/>

```
language_attributes(); ?>
charset="<?php bloginfo( 'charset' ); ?>" />
name="viewport" content="width=device-width" />
wp_title( '|', true, 'right' ); ?> /title
rel="profile" href="http://gmpg.org/xfn/11" />
rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>" />
fruitful_get_favicon(); ?>
wp_head(); ?>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $menu_pos = '';
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
if (isset($theme_options['menu_position']))
$menu_pos = esc_attr($theme_options['menu_position']);
```

## ES5 vs ES6

¿Sabías que...?

**Violet Jessop, azafata marítima, sobrevivió al hundimiento del Britannic, el Titanic y el Olympic, los tres grandes transatlánticos**



ECMAScript es una especificación de lenguaje de secuencias de comandos de marca registrada definida por ECMA International. Fue creado para estandarizar JavaScript.

El lenguaje de secuencias de comandos de ES tiene muchas implementaciones, y la más popular es JavaScript. Generalmente, ECMAScript se utiliza para la creación de scripts del lado del cliente de la World Wide Web.

**ES5** es una abreviatura de **ECMAScript 5** y también conocido como **ECMAScript 2009**.

La sexta edición del estándar ECMAScript es **ES6** o **ECMAScript 6**. También se conoce como **ECMAScript 2015**. ES6 es una mejora importante en el lenguaje JavaScript que nos permite escribir programas para aplicaciones complejas.

Aunque ES5 y ES6 tienen algunas similitudes en su naturaleza, también hay muchas diferencias entre ellos.

**ES12** o **ECMAScript 2021** es la nueva especificación que está previsto que salga en junio de 2021.

ES5	ES6
Las variables se definen con <b>var</b>	Las variables se pueden definir con <b>var</b> , <b>let</b> o <b>const</b>
Tiene menor rendimiento	Su rendimiento está más optimizado
La manipulación de objetos es más costosa a nivel de tiempo	La manipulación de objetos está más optimizada a nivel de tiempo
Las funciones necesitan las palabras <b>function</b> y <b>return</b> para ser definidas	Aparece una nueva forma de declarar funciones, las <b>arrow functions</b>
Hay una gran comunidad de soporte	Aún no tiene una gran comunidad de soporte

ES5

## Funciones

necesitan la palabra **function** y, en caso de devolver un resultado debe hacerse con la palabra **return**

```
function helloParam (param) {  
    return 'hello ' + param;  
}
```



## Concatenación

para concatenar variables con literales debe usarse +

```
var a = 'hello';
```

```
var b = 'world';
```

```
var out = a + ' ' + b;
```

```
// hello world
```

## Concatenación

hay que ir con cuidado, ya que `+` sirve tanto para concatenar como para sumar. Si no se indica el tipo de la variable, pueden salir resultados inesperados

```
var a = 2;
```

```
var b = 3;
```

```
var out = 'the result is: '+a+b;
```

```
// the result is: 23
```

## Concatenación

en esos casos, se puede usar la función nativa **parseInt()** para forzar que el resultado sea numérico

```
var a = 2;
```

```
var b = 3;
```

```
var out = 'the result is: '+parseInt(a+b);
```

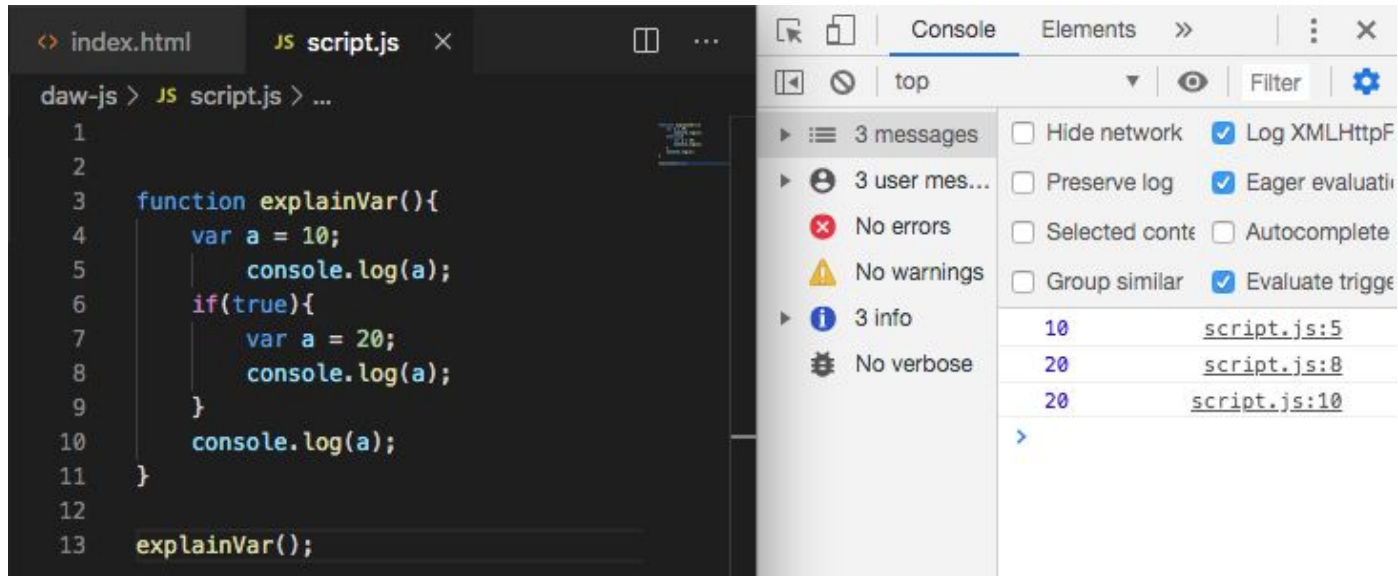
```
// the result is: 5
```

## Var

Las variables **var**, una vez declaradas e inicializadas se pueden ejecutar dentro del scope dónde han sido declaradas, però también puede accederse a ellas desde dentro de los elementos de bloque { ... }

En el siguiente ejemplo se están declarando 2 variables **a**, y una sobreescribe a la otra pese a estar en 2 contextos distintos (la segunda está dentro de un bloque { ... }, en ese caso un **if**.)

Eso nos puede llevar fácilmente a errores si no cuidamos el código, ya que podemos estar creando variables globales y modificarlas sin darnos cuenta.



The screenshot shows a web browser's developer console with the 'Console' tab selected. The left pane displays the source code of 'script.js' with line numbers 1 through 13. The right pane shows the console output, which includes three log messages: the value '10' at line 5, '20' at line 8, and '20' at line 10. The console also shows that there are no errors, warnings, or verbose messages.

```
1  
2  
3 function explainVar(){  
4     var a = 10;  
5     console.log(a);  
6     if(true){  
7         var a = 20;  
8         console.log(a);  
9     }  
10    console.log(a);  
11 }  
12  
13 explainVar();
```

Console output:

- 10 script.js:5
- 20 script.js:8
- 20 script.js:10

## Objetos

se pueden fusionar varios objetos en uno de solo con **Object.assign**

```
var obj1 = { a: 1, b: 2 }
```

```
var obj2 = { a: 2, c: 3, d: 4}
```

```
var obj3 = Object.assign(obj1, obj2)
```

```
// obj3 = { a: 2, b: 2, c: 3, d: 4}
```

## Objetos

se pueden desestructurar objetos asignándolos a variables

```
var obj1 = { a: 1, b: 2, c: 3, d: 4 }
```

```
var a = obj1.a;
```

```
var b = obj1.b;
```

```
var c = obj1.c;
```

```
var d = obj1.d;
```

## Objetos

se pueden definir objetos a partir de variables

```
var aa = 1;
```

```
var bb = 2;
```

```
var cc = 3;
```

```
var dd = 4;
```

```
var obj1 = {a: aa, b: bb, c: cc, d: dd }
```



## Callbacks

se pueden definir funciones que acepten como parámetro a otra función

```
function isGreater (a, b, cb){  
    var greater = false  
    if(a > b) {  
        greater = true  
    }  
    cb(greater)  
}
```

```
isGreater(1, 2, function (result){  
    if(result) {  
        console.log('greater');  
    } else {  
        console.log('smaller')  
    }  
})
```

ES6

## Funciones

no necesitan la palabra **function** y, en caso de devolver un resultado definido en una sola línea de código, tampoco necesita **return**.

Si no necesita parámetro, se usa **()**. Si sólo necesita un parámetro, se puede pasar sin paréntesis.

```
const helloWorld = () => 'hello world'
```

```
const helloParam = (p1,p2) => 'hello ' + p1 + ' ' + p2
```

```
const helloParam = param => 'hello ' + param
```

## Funciones

en caso de necesitar varias línea de código en la función, deberán usarse **{ }** y **return**. Si necesita varios parámetros, deben pasarse entre **( )**.

```
const nombreFuncion = (a,b) => {  
    let out = false;  
    if (a>b) out=true;  
    return out;  
}
```

## Interpolación

para fusionar textos literales con variabes, se puede usar la interpolación. Todo el conjunto debe escribirse entre `` (acento abierto, no comilla simple) y las variables con **`${var}`**

```
var a = 2;
```

```
var b = 3;
```

```
var out = `the sum of ${a} + ${b} is: ${a+b}`;
```

```
// the sum of 2 + 3 is: 5
```

## Let

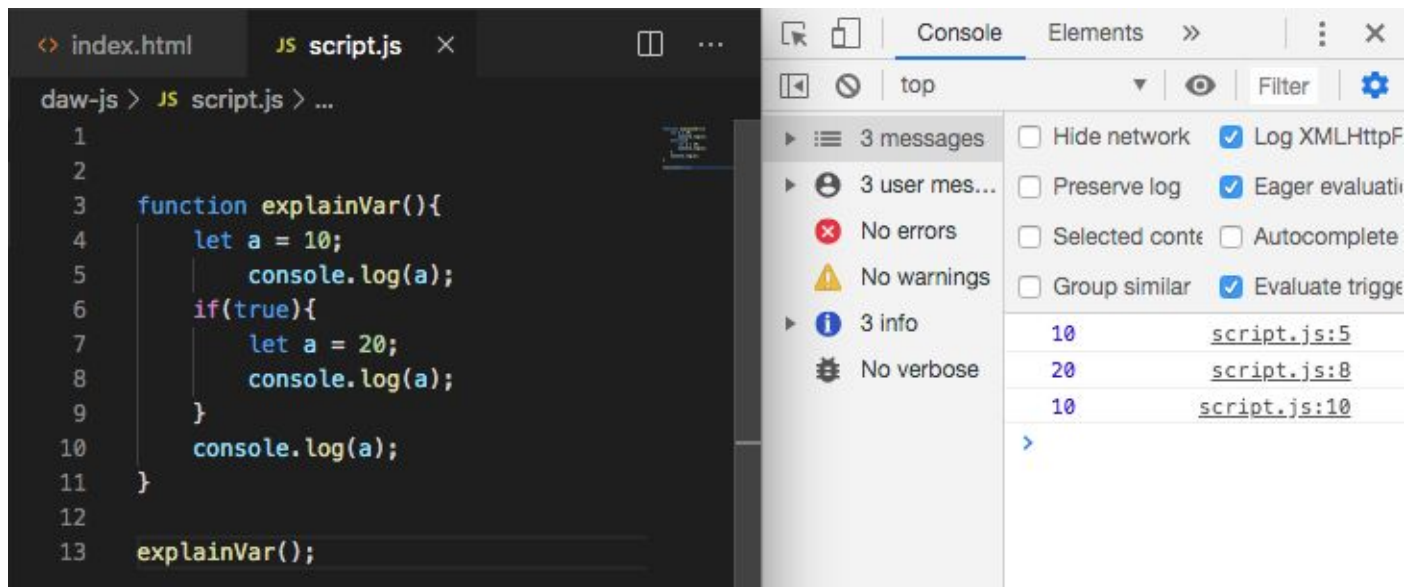
Las variables **let** son más recientes y pueden evitar fácilmente el problema de ES5 de sobrescribir variables `var` con el mismo nombre.

Este tipo de variables son accesibles sólo dentro del contexto dónde se han creado. Es decir, una variable `let` no puede acceder desde fuera de un bloque `{ ... }` a una variable `let` dentro de un bloque.

## Let

Hay quien dice que las variables **let** son las nuevas **var**. Es más seguro utilizar generalmente **let**, y usar **var** sólo cuando se quieran usar variables globales.

Repitiendo el ejemplo anterior, pero con variables **let**, vemos que cambian los valores, y cada una muestra el valor dentro de su contexto.



The screenshot shows a web browser with a dark-themed code editor on the left and a console on the right. The code editor displays the following JavaScript code:

```
1  
2  
3 function explainVar(){  
4   let a = 10;  
5   console.log(a);  
6   if(true){  
7     let a = 20;  
8     console.log(a);  
9   }  
10  console.log(a);  
11 }  
12  
13 explainVar();
```

The console on the right shows the execution results:

- 3 messages
- 3 user messages
- No errors
- No warnings
- 3 info
- No verbose

The console also displays the following log messages:

- 10 script.js:5
- 20 script.js:8
- 10 script.js:10



## Const

Las variables tipo **const** actúan como las de tipo **let** a nivel de contexto, pero una vez asignadas no se le puede cambiar el valor.

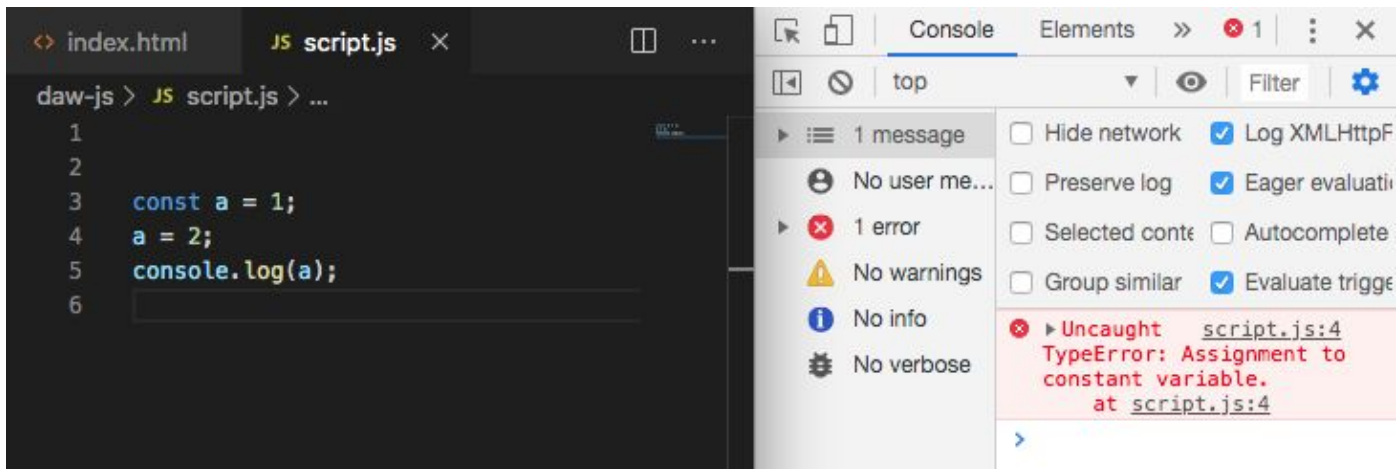
Si que puede cambiar su valor si se trata de un objeto (veremos el concepto más adelante), pero para entenderlo, sería un conjunto de información de tipo clave:valor

Ej:

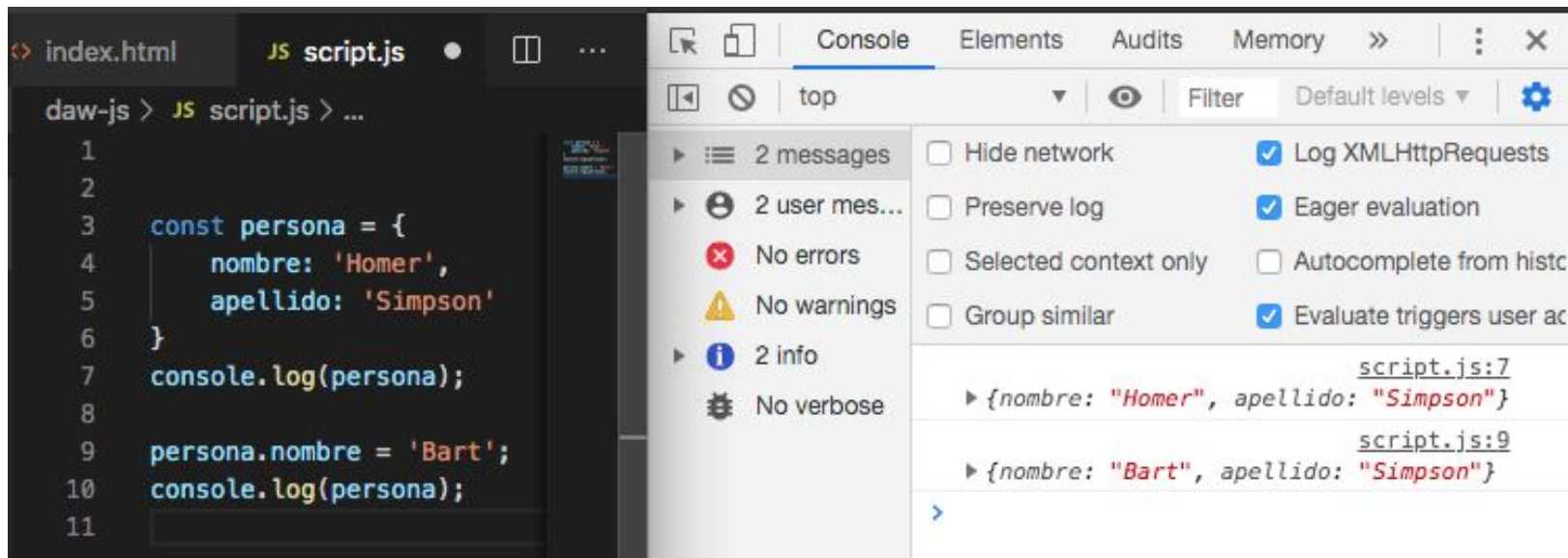
```
const persona = { nombre: Aleix, apellido: Martí }
```

Por esto, si prevemos que una variable va a conservar su valor, podemos declararla como **const**.

Si se intenta cambiar el valor a una **const**, se genera un error



Sí que se puede cambiar el valor de la información dentro de un objeto de tipo **const**



The screenshot shows a code editor on the left and a browser console on the right. The code editor displays the following JavaScript code:

```
1
2
3  const persona = {
4      nombre: 'Homer',
5      apellido: 'Simpson'
6  }
7  console.log(persona);
8
9  persona.nombre = 'Bart';
10 console.log(persona);
11
```

The browser console on the right shows the output of the code. It has tabs for Console, Elements, Audits, and Memory. The Console tab is active, showing a list of messages. The first message is an info message (2 info) showing the initial object: `{nombre: "Homer", apellido: "Simpson"}` at `script.js:7`. The second message is an info message (2 info) showing the object after the change: `{nombre: "Bart", apellido: "Simpson"}` at `script.js:9`. The console also shows settings for the log, including checkboxes for "Hide network", "Preserve log", "Selected context only", "Group similar", "Log XMLHttpRequests", "Eager evaluation", "Autocomplete from history", and "Evaluate triggers user action".

## Objetos

se pueden fusionar varios objetos en uno de solo el operador **spread**: ...

```
var obj1 = { a: 1, b: 2 }
```

```
var obj2 = { a: 2, c: 3, d: 4 }
```

```
var obj3 = { ...obj1, ...obj2 }
```

```
// obj3 = { a: 2, b: 2, c: 3, d: 4 }
```

## Objetos

se pueden desestructurar objetos asignándolos a variables en una sola operación

```
const obj1 = { a: 1, b: 2, c: 3, d: 4 }
```

```
const {
```

```
  a,
```

```
  b,
```

```
  c,
```

```
  d
```

```
} = obj1
```

## Objetos

se pueden definir objetos a partir de variables

```
var a = 1;
```

```
var b = 2;
```

```
var c = 3;
```

```
var d = 4;
```

```
var obj1 = {a, b, c, d }
```

## Promises

permeten **resolver** y **rechazar** el resultado según el código de la función

```
const isGreater = (a, b) => {  
  return new Promise ((resolve, reject) =>  
  {  
    if(a > b) {  
      resolve(true)  
    } else {  
      reject(false)  
    }  
  })  
}
```

```
isGreater(1, 2)  
  .then(result => {  
    console.log('greater')  
  })  
  .catch(result => {  
    console.log('smaller')  
  })
```

<https://medium.com/recraftrelic/es5-vs-es6-with-example-code-9901fa0136fc>

<https://www.javatpoint.com/es5-vs-es6>

<https://www.geeksforgeeks.org/difference-between-es5-and-es6/>



```
language_attributes(); ?>
charset="<?php bloginfo( 'charset' ); ?>" />
name="viewport" content="width=device-width" />
wp_title( '|', true, 'right' ); ?> /title
rel="profile" href="http://gmpg.org/xfn/11" />
rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>" />
fruitful_get_favicon(); ?>
<?php wp_head(); ?>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
esc_attr($theme_options['logo_position'])
esc_attr($theme_options['logo_position'])
```

# Objetos

¿Sabías que...?

**El sonido del agua cuando se vierte cambia ligeramente  
según la temperatura a la que esté**



# Objetos


Los objetos se utilizan para almacenar varias colecciones en formato clave - valor y otras entidades más complejas.

Son un tipo de variable que puede almacenar múltiples valores.

Almacenan los datos en formato **clave : valor**

Además de datos, también pueden contener **métodos**.

Es muy común declararlos en una variable de tipo **const**.

Object	Properties	Methods
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>

Los objetos se definen usando claves ( `{ }` ) y los datos deben ir en formato **clave:valor**, separados entre ellos con una coma simple ( `,` )

```
const person = {  
  firstName: "Stan",  
  lastName: "Smith",  
  age: 42,  
  hair: "black"  
};
```

Los pares **clave:valor** de un objeto se llaman **propiedades** (properties).

Para acceder a ellos se puede hacer con un punto ( . ) o corchetes ( [ ] ). En ambos casos es necesario saber el nombre de la clave para poder obtener su valor.

**object.propName**

```
console.log( person.firstName )  
// Stan
```

**object["propName"]**

```
console.log( person["firstName"]  
// Stan
```

A parte de valores, en un objeto también se pueden definir **métodos**.  
Se declarar una función como el valor de una clave.

```
const person = {  
  firstName: "Stan",  
  lastName: "Smith",  
  age: 42,  
  hair: "black",  
  fullname: function() {  
    return this.firstName + " " + this.lastName  
  }  
};
```



Para llamar a un método, debe llamarse de la forma: `objeto.metodo()`

```
person.fullname()  
// Stan Smith
```

**Nota:** si se llama al método sin los paréntesis, lo que se devuelve es la definición de la función

```
person.fullname  
//f () {  
    return this.firstName + " " + this.lastName  
}
```

Para mostrar objetos hay varias formas de hacerlo, pero primero deben ser tratados. Si se intenta mostrar el objeto directamente, sólo se verá:

```
[object Object]
```

Hay varias alternativas para mostrar objetos:

- Mostrar las propiedades por **nombre**
- Mostrar las propiedades con un **bucle**
- Mostrar el objeto con el método **Object.values()**
- Mostrar el objeto usando **JSON.stringify()**

Para los siguientes ejemplos partiremos del siguiente código:

Contenedor HTML para mostrar el resultado

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

Objeto

```
const person = {  
  name: "Bart",  
  age: 10,  
  city: "Springfield"  
};
```

Capturar por ID desde JS el contenedor HTML y printar allí el resultado

```
const out = document.getElementById("container");  
out.innerHTML = person;
```

Mostrar el objeto **entero** directamente

```
out.innerHTML = person;
```

```
// [object Object]
```

Mostrar el valor de las propiedades del objeto por **nombre** (clave)

```
out.innerHTML = person.name + ' from ' + person.city;
```

```
// Bart from Springfield
```

Mostrar el objeto utilizando un **bucle**

```
let txt = "";  
for (let x in person) {  
    txt += person[x] + " ";  
};
```

```
out.innerHTML = txt;
```

```
// Bart 10 Springfield
```

Un objeto puede convertirse en un array usando **Object.values()**

```
const myArray = Object.values(person);
```

```
out.innerHTML = myArray;
```

```
// Bart,10,Springfield
```



Un objeto puede convertirse en un string usando **JSON.stringify()**

```
const myString = JSON.stringify(person);
```

```
out.innerHTML = myString;
```

```
// {"name":"Bart","age":10,"city":"Springfield"}
```

## Constructor

Se pueden crear **constructores** para crear objetos a partir de una función. Es una buena práctica que los nombre empiecen en **mayúscula** para distinguirlos de los otros tipos de función.

Para crear un nuevo objeto se usa la palabra clave **new**.

```
function Simpson(name, age, haircolor){  
    this.name = name;  
    this.age = age;  
    this.hairColor = haircolor;  
}  
  
const bart = new Simpson('Bart',10,'yellow');  
const marge = new Simpson('Marge',36,'blue');
```

```
console.log(bart);
```

```
name: "Bart",  
age: 10,  
hairColor: "yellow"
```

```
console.log(marge);
```

```
name: "Marge",  
age: 36,  
hairColor: "blue"
```

Se puede editar un valor accediendo a su clave y asignándole un nuevo valor.

```
bart.name = "Bartholomew"
```

```
console.log(bart);
```

```
name: "Bartholomew",  
age: 10,  
hairColor: "yellow"
```

Exercici:

Demandar dades a l'usuari: nom, cognom, edat

Crear un objecte amb aquestes dades utilitzant un constructor d'objectes

L'objecte ha de tenir un mètode per retornar la informació amb un alert

[https://www.w3schools.com/js/js\\_objects.asp](https://www.w3schools.com/js/js_objects.asp)

[https://www.w3schools.com/js/js\\_object\\_display.asp](https://www.w3schools.com/js/js_object_display.asp)

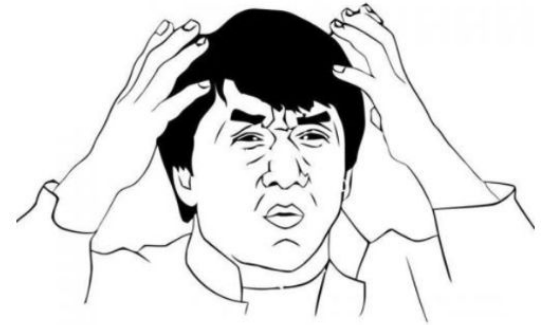
[https://www.w3schools.com/js/js\\_object\\_constructors.asp](https://www.w3schools.com/js/js_object_constructors.asp)

```
language_attributes(); ?>
charset="<?php bloginfo( 'charset' ); ?>" />
name="viewport" content="width=device-width" />
wp_title( '|', true, 'right' ); ?> /title
rel="profile" href="http://gmpg.org/xfn/11" />
rel="pingback" href="<?php bloginfo( 'pingback_url' ); ?>" />
fruitful_get_favicon(); ?>
<script src="<?php echo get_template_directory_uri(); ?>/js/main.js"></script>
<?php wp_head(); ?>
</head>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position'])) {
$logo_pos = esc_attr($theme_options['logo_position']);
}
</?php>
```

# String, Number

¿Sabías que...?

**Las Islas Canarias toman su nombre de los perros (can), no  
de los canarios**





# Strings

Los strings en JavaScript se usan para guardar y manipular textos.

Se pueden definir con comillas simples o dobles.

Aunque entre las comillas no haya nada, también se considera string.

Los números entre comillas se consideran strings.

Se pueden combinar tipos de comillas, teniendo en cuenta el orden.

```
let text = "Hello String";  
let text = 'Hello String';  
let text = 'The console says "Hello String"';
```

Se pueden escapar caracteres usando “\”

```
let text = 'This can\'t be possible!';  
// This can't be possible!
```

Se pueden concatenar distintos strings con " + "

```
let text = 'Hello ' + 'String!';  
// Hello String!
```

```
let text = 'Hello' + 'String!';  
// HelloString!
```

Hay [muchos métodos para trabajar con los strings](#), a continuación veremos algunos ejemplos, pero hay muchos más.

## Longitud

Se puede saber la longitud de un string con la propiedad **length**

```
let text = "Hello String";  
text.length;  
// 12
```

## Substring

Existen 3 métodos similares para obtener partes más pequeñas de un string: **slice**(start,end) , **substring**(start,end) , **substr**(start,length)

Cada método usa unos argumentos distintos.

Hay que tener en cuenta que el primer caracter es la posición 0.

```
let long = "Bond. James Bond";  
let short = long.substr(6,5);  
console.log(short)  
// James
```

## Replace

Con el método **replace**(search, replace) se pueden sustituir todas las coincidencias de texto dentro de un string. Requiere 2 argumentos: el texto a reemplazar y el nuevo valor.

Devuelve un nuevo string, dejando intacto el original.

```
let oldText = "Bart Simpson";  
let newText = oldText.replace("Bart", "Lisa")  
console.log(newText)  
// Lisa Simpson
```



## Replace

También acepta [expresiones regulares \(RegExp\)](#) como argumentos.

```
let dumb = "La Tierra es plana";  
let mimimi = dumb.replace(/[aeiou]/gi, "i");  
console.log(mimimi);  
// Li Tiirri is plini
```

## Mayúsculas/minúsculas

Los strings se pueden convertir a letras mayúsculas o minúsculas con los métodos **toUpperCase()** y **toLowerCase()**

```
let oldText = "Maggie Simpson";  
let newText = oldText.toUpperCase()  
console.log(newText)  
// MAGGIE SIMPSON
```

## concat

Además de usar '+' para concatenar strings, también se puede usar el método **concat()**

Acepta un argumento opcional para indicar un string que actúe como "pegamento" para unir las 2 cadenas originales.

Devuelve un nuevo string, manteniendo intactos los originales

```
let text1 = "Hello";  
let text2 = "World";  
let text3 = text1.concat(" ", text2);  
console.log(text3)  
// Hello World
```

## trim

El método `trim()` eliminar los espacios en blanco del inicio y final de un string.

```
let oldText = "  Maggie Simpson  ";  
let newText = oldText.trim()  
console.log(oldText)  
console.log(newText)  
//   Maggie Simpson  
// Maggie Simpson
```

## split

El método **split**(separador) divide un string por el separador y devuelve un array de substrings.

```
let text = "Bart,Lisa,Maggie";  
let arr = text.split(",");  
console.dir(arr);  
// ["Bart","Lisa","Maggie"]
```

## split

Si se utiliza una cadena vacía ("" ) como separador, la cadena se divide entre cada carácter.

```
let text = "Bart";  
let arr = text.split("");  
console.dir(arr);  
// ["B","a","r","t"]
```

## indexOf

Devuelve la posición de la **primera** coincidencia del substring indicado como argumento.

Empieza a contar desde la posición 0 y retorna -1 si no hay coincidencias.

```
let text = "ornitorrinco";  
let pos = text.indexOf("i");  
console.log(pos)  
// 3
```

## lastIndexOf

Devuelve la posición de la **última** coincidencia del substring indicado como argumento.

Empieza a contar desde la posición 0 y retorna -1 si no hay coincidencias.

```
let text = "ornitorrinco";  
let pos = text.lastIndexOf("i");  
console.log(pos)  
// 8
```



## match

El método **match()** busca en un string una coincidencia con una expresión regular y devuelve las coincidencias como un objeto Array.

Devuelve **null** si no hay coincidencias.

```
let text = "El ornitorrinco pegó un brinco";  
text.match(/inco/g)  
//["inco", "inco"]
```

## includes

El método **includes** (`substring`) busca en un string una coincidencia con un substring

Devuelve **true** o **false** según haya o no coincidencias.

```
let text = "El ornitorrinco pegó un brinco";  
text.includes("brinco")  
//true
```

# Numbers

A diferencia de muchos otros lenguajes de programación, JavaScript no define diferentes tipos de números, como enteros, cortos, largos, de punto flotante, etc.

JavaScript tiene sólo un tipo de número. Los números se pueden escribir con o sin decimales.

```
let x = 3.24  
let y = 5
```

Los números enteros (números sin un período o notación exponencial) tienen una precisión de hasta 15 dígitos.

El máximo número de decimales es 17.

JavaScript usa el operador ' + ' tanto para la suma como para la concatenación.

Si ambos valores son números, se agregan. Las cadenas se concatenan.

Un número entre comillas se convierte a String.

```
let x = 10;  
let y = 20;  
let z = x + y;  
console.log(z);  
// 30
```

```
let x = "10";  
let y = "20";  
let z = x + y;  
console.log(z);  
// 1020
```

Si se mezclan números y strings el resultado se convierte a string

```
let x = 10;  
let y = "20";  
let z = x + y;  
console.log(z);  
// ¿Qué mostrará la consola?
```

Si se mezclan números y strings el resultado se convierte a string

```
let x = 10;  
let y = "20";  
let z = x + y;  
console.log(z);  
// ¿Qué mostrará la consola?  
// 1020
```



Si se mezclan números y strings el resultado se convierte a string

```
let x = 10;  
let y = 20;  
let z = "30";  
let result = x + y + z;  
// ¿Qué mostrará la consola?
```

Si se mezclan números y strings el resultado se convierte a string

```
let x = 10;  
let y = 20;  
let z = "30";  
let result = x + y + z;  
// ¿Qué mostrará la consola?  
// 3030
```

El intérprete de JavaScript funciona de izquierda a derecha.

Los primeros  $10 + 20$  se suman porque **x** y **y** son números.

Entonces  $30 + "30"$  se concatenan porque **z** es un string.

Para las operaciones matemáticas diferentes a ' + ' JavaScript trata de convertir los strings a números y operar con ellos.

```
let x = 100;  
let y = "30";  
let a = x+y;  
let b = x-y;  
console.log(a,b);  
// 10030 , 70
```

**NaN** (Not a Number) es una palabra reservada de JavaScript para referirse un valor que no es un número legal. Se puede obtener como resultado de una operación entre un string y un número.

```
let a = "ornitorrinco";  
let b = 100;  
let z = a - b  
console.log(z)  
// NaN
```

**isNaN()** se puede usar para saber si el valor recibido como argumento es un número válido o no. Devuelve un booleano.

```
let a = "ornitorrinco";  
let b = 100;  
let c = "50";
```

```
isNaN(a-b)  
// true  
isNaN(b-c)  
// false
```

**isNaN()** se puede usar para saber si el valor recibido como argumento es un número

```
let compare = (a,b)=>{  
  let y = a-b;  
  let z;  
  isNaN(y) ? z="error!" : z=y;  
  return z;  
}
```

```
console.log(compare(30,5))  
// 25
```

```
console.log(compare(30,"b"))  
// error!
```

**parseInt()** se usa para convertir un string a un número.

```
let a = "50";  
let b = 100;  
console.log(a+b);  
// 50100
```

```
let a = "50";  
let b = 100;  
console.log(parseInt(a)+b)  
// 150
```



[https://www.w3schools.com/js/js\\_strings.asp](https://www.w3schools.com/js/js_strings.asp)

[https://www.w3schools.com/js/js\\_string\\_methods.asp](https://www.w3schools.com/js/js_string_methods.asp)

[https://www.w3schools.com/jsref/jsref\\_obj\\_string.asp](https://www.w3schools.com/jsref/jsref_obj_string.asp)

[https://www.w3schools.com/js/js\\_string\\_search.asp](https://www.w3schools.com/js/js_string_search.asp)

```
<?php language_attributes(); ?>
<?php bloginfo( 'charset' ); ?>
<?php wp_title( '|', true, 'right' ); ?>
<?php bloginfo( 'description' ); ?>
<?php wp_head(); ?>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
</div>
```

# Map, Date, Math

¿Sabías que...?

**Hay más tigres en cautiverio en EE.UU. que viviendo en la naturaleza en todo el mundo**



# Map

Un mapa es un tipo de objeto.

Contiene pares clave-valor donde las **claves pueden ser de cualquier tipo de datos**, a diferencia de los objetos comunes, que sólo pueden ser strings o símbolos (otro tipo de objeto).

Recuerda el **orden de inserción original** de las claves.

Tiene una propiedad que representa el **tamaño del mapa** (número de elementos).

## Creación

Se puede crear un nuevo mapa con **new Map()** y pasarle un array en el momento de crearlo, o crear un mapa vacío y añadirle elementos usando **.set()**

```
const fruits = new Map([  
    ["apple", 50],  
    ["kiwi", 100],  
    ["orange", 80]  
])
```

```
fruits.set("pear", 30)
```

## Consulta

Es posible obtener el tamaño de un Map con la propiedad **.size**

Se puede obtener un valor con el método **.get()** a partir de una clave

Se puede consultar si el mapa contiene un objeto con el método **.has()**

```
fruits.size  
// 4
```

```
fruits.get("apple")  
// 50
```

```
fruits.has("apple")  
// true
```

## **Iteración**

Hay varios métodos que permiten recorrer los elementos de un Map

**forEach()**

**entries()**

**keys()**

**values()**



## forEach()

Permite pasarle una función como *callback* que se aplicará a cada uno de los elementos del mapa

```
let text = "";  
fruits.forEach (function(value, key) {  
    text += key + ' = ' + value + '; '  
})
```

```
console.log(text);  
// apple = 50; kiwi = 100; orange = 80;
```

## keys()

Itera todo el mapa y devuelve todas sus claves

```
let fruitList = "";  
for (let x of fruits.keys()) {  
  fruitList += x+'';  
}
```

```
console.log(fruitList);  
// apple;kiwi;orange;
```

## **values()**

Itera todo el mapa y devuelve todos sus valores

```
let sum = 0;  
for (let x of fruits.values()) {  
  sum += x;  
}
```

```
console.log(sum);  
// 230
```

## **entries()**

Itera todo el mapa y devuelve todos sus pares de claves y valores

```
let text = "";  
for (let x of fruits.entries()) {  
    text += x + ' ; ' ;  
}  
  
console.log(text);  
// "apple,50 | kiwi,100 | orange,80 |"
```

# Date

Date es un objeto de fecha de JavaScript que representa un único momento en el tiempo en un formato independiente de la plataforma en la que se ejecute.

Los objetos Date contienen un número que representa los milisegundos que han pasado desde la medianoche del 1 de enero de 1970 [UTC](#).

Este formato se llama tiempo Unix.

## Creación

Para crear un nuevo objeto de fecha se usa **new Date()**.

Se puede crear sin argumentos, pasándole un número de ms, un string de fecha, varios argumentos desde año hasta ms.

```
var d = new Date();  
var d = new Date(milliseconds);  
var d = new Date(dateString);  
var d = new Date(year, month, day, hours, minutes, seconds, milliseconds);
```

## Date()

si no se usan argumentos se creará la fecha del mismo momento actual.

```
const a = new Date()
```

```
// Sat Aug 28 2021 17:53:51 GMT+0200 (hora de verano de Europa  
central)
```



## **Date(dateString)**

se puede pasar un string con una fecha para crear un nuevo objeto Date

```
const a = new Date("October 13, 2014 11:13:00");  
// Mon Oct 13 2014 11:13:00 GMT+0200 (hora de verano de Europa  
central)
```

```
const b = new Date("2021/08/28");  
// Sat Aug 28 2021 00:00:00 GMT+0200 (hora de verano de Europa  
central)
```

## Date(milliseconds)

se puede pasar un número de milisegundos para crear una fecha. Creará una fecha sumando los milisegundos indicados a la fecha 1 de enero de 1970

```
// 24*60*60*1000 = 86400000
```

```
const b = new Date(86400000)
```

```
// Fri Jan 02 1970 01:00:00 GMT+0100 (hora estándar de Europa  
central)
```

## **Date(year, month, day, hours, minutes, seconds, milliseconds)**

se pueden pasar varios argumentos en formato numérico para crear una nueva fecha, desde el año hasta los milisegundos.

```
const a = new Date(2018, 11, 24, 10, 33, 30, 0);  
// Mon Dec 24 2018 10:33:30 GMT+0100 (hora estándar de Europa  
central)
```

Existe una larga lista de [métodos para trabajar con el objeto Date\(\)](#), tanto para obtener como para modificar las fechas.

Se puede trabajar sobre el año, día, segundos... se trata de revisar la documentación y usar aquellos que se ajusten más al objetivo.

**toString(), toJSON(), setTime(), now()** son algunos interesantes, pero hay un listado muy extenso.

# Math

El objeto **Math** permite realizar tareas matemáticas.

**Math** no es un constructor.

Todas las propiedades / métodos de Math se pueden llamar utilizando Math como un objeto, sin crearlo:

```
let a = Math.PI  
// 3.141592653589793
```

```
let b = Math.sqrt(25)  
// 5
```

## Propiedades

**Math** ofrece 8 propiedades, que retornan unas constantes

```
Math.E           // Número de Euler
Math.PI          // PI
Math.SQRT2       // Raíz cuadrada de 2
Math.SQRT1_2     // Raíz cuadrada de 1/2
Math.LN2         // Logaritmo 2
Math.LN10        // Logaritmo de 10
Math.LOG2E       // Logaritmo de E en base 2
Math.LOG10E      // Logaritmo de E en base 10
```

## Métodos

**Math** tiene un [listado muy extenso de métodos](#) que permiten realizar numerosas operaciones matemáticas.

A continuación veremos algunos interesantes, pero la lista entera es muy larga.



## Redondeo

Para redondear un número decimal a su entero superior (**ceil**) o inferior (**floor**). Para redondear un decimal a su entero más próximo (**round**)

```
Math.ceil(2.4)  
// 3
```

```
Math.round(2.4)  
// 2
```

```
Math.floor(2.4)  
// 2
```

```
Math.round(2.6)  
// 3
```

## Máximo / Mínimo

Se puede obtener el valor máximo o mínimo de un listado de números.

```
Math.max(2, 45, 23, 13, 1, 6, 8, 4)  
// 45
```

```
Math.min(2, 45, 23, 13, 1, 6, 8, 4)  
// 1
```

## Random

`Math.random()` devuelve un número decimal entre 0 (incluido) y 1 (no incluido)

Combinado con otras funciones se puede obtener un número entero.

```
Math.floor( Math.random() * 10 )
```

## Random

Se puede crear una función que reciba 2 parámetros para que devuelva un número aleatorio entre 2 valores

```
function randomInteger(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}
```

[https://www.w3schools.com/js/js\\_object\\_maps.asp](https://www.w3schools.com/js/js_object_maps.asp)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working with Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects)

[https://www.w3schools.com/jsref/jsref\\_obj\\_date.asp](https://www.w3schools.com/jsref/jsref_obj_date.asp)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global Objects/Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)

[https://www.w3schools.com/js/js\\_dates.asp](https://www.w3schools.com/js/js_dates.asp)

<https://time.is/UTC>

[https://www.w3schools.com/jsref/jsref\\_obj\\_math.asp](https://www.w3schools.com/jsref/jsref_obj_math.asp)

[https://www.w3schools.com/js/js\\_math.asp](https://www.w3schools.com/js/js_math.asp)

[https://www.w3schools.com/js/js\\_strings.asp](https://www.w3schools.com/js/js_strings.asp)

[https://www.w3schools.com/js/js\\_string\\_methods.asp](https://www.w3schools.com/js/js_string_methods.asp)

```
<?php language_attributes(); ?>
<?php bloginfo( 'charset' ); ?>
<?php wp_title( '|', true, 'right' ); ?>
<?php wp_title( 'profile', true, 'right' ); ?>
<?php bloginfo( 'pingback_url' ); ?>
<?php fruitful_get_favicon(); ?>
<?php echo get_template_directory_uri(); ?>
<?php wp_head(); ?>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
</div>
```

# Arrays (I)

¿Sabías que...?

Las probabilidades de obtener una escalera real en  
poker (A,K,Q,J,10) son de una entre 649.740



# Arrays



Antes de empezar con los arrays, veamos otra forma de mostrar los datos a través de la consola.

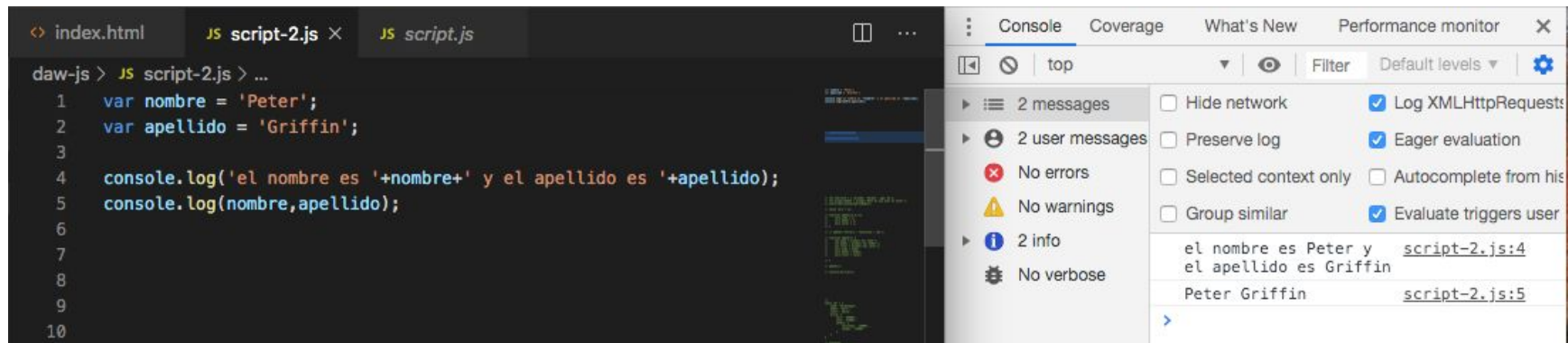
Hasta ahora hemos utilizado **console.log()**, muy útil para representar valores simples o strings. Permite encadenar strings y variables usando **+**

```
console.log( 'el nombre es ' + nombre + ' y el apellido  
es ' + apellido )
```

o mostrar distintos valores, separados por **,** (una coma simple)

```
console.log( nombre, apellido )
```

# Arrays



The screenshot shows a web browser's developer console with the following content:

**Code Editor:**

```
daw-js > JS script-2.js > ...
1  var nombre = 'Peter';
2  var apellido = 'Griffin';
3
4  console.log('el nombre es '+nombre+' y el apellido es '+apellido);
5  console.log(nombre,apellido);
6
7
8
9
10
```

**Console Panel:**

- 2 messages
- 2 user messages
- No errors
- No warnings
- 2 info
- No verbose

**Log Details:**

- el nombre es Peter y el apellido es Griffin (script-2.js:4)
- Peter Griffin (script-2.js:5)

**Console Settings:**

- ☐ Hide network
- ☐ Preserve log
- ☐ Selected context only
- ☐ Group similar
- ☒ Log XMLHttpRequests
- ☒ Eager evaluation
- ☐ Autocomplete from history
- ☒ Evaluate triggers user

Hay muchas formas de [mostrar contenido en la consola](#). El más “famoso” es `console.log`, pero otra forma muy útil es **`console.dir()`**. Es especialmente útil cuando queremos mostrar objetos o arrays.

Veamos este ejemplo, con un objeto y un array, y los dos mostrándolos por consola usando `console.log` y `console.dir`

**`console.log`** muestra directamente toda la información (si es muy larga se parte la línea)

**`console.dir`** nos muestra el tipo de información que estamos viendo (array u objeto)

# Arrays

The screenshot shows a web browser's developer console with the following components:

- Code Editor:** Contains JavaScript code for creating an object and an array.

```
daw-js > JS script-2.js > ...  
1 // objeto  
2 const myObject = {  
3   name: 'Volkswagen',  
4   model: 'Golf',  
5   color: 'white',  
6 }  
7 console.log(myObject);  
8 console.dir(myObject);  
9  
10 // array  
11 const myArray = ['Audi', 'Seat', 'Peugeot']  
12  
13 console.log(myArray);  
14 console.dir(myArray);  
15
```
- Console Panel:** Displays the output of the code.
  - 4 messages
  - 4 user messages
  - No errors
  - No warnings
  - 4 info
  - No verbose
- Log Details:** Shows the output of the console.log and console.dir calls.
  - `script-2.js:6`: `{name: "Volkswagen", model: "Golf", color: "white"}`
  - `script-2.js:7`: `Object`
  - `script-2.js:11`: `(3) ["Audi", "Seat", "Peugeot"]`
  - `script-2.js:12`: `Array(3)`

Un **array** es una colección de valores. Sirven para almacenar distintos valores dentro de una sola variable. De hecho, son un tipo especial de objetos.

La diferencia, es que los **objetos necesitan una clave (o nombre) para acceder a sus valores**, y en los **arrays se accede por la posición que ocupa cada elemento** (0, 1, 2, 3...).

Se definen con unos corchetes simples

```
const myArray = [ ]
```

# Arrays

The screenshot shows a web browser's developer console with the following elements:

- Code Editor:** Displays the following JavaScript code:

```
1  const miArray = [];  
2  
3  console.dir(miArray);  
4  
5  
6  
7  
8  
9
```
- Console Panel:** Shows a list of messages:
  - 1 message (expanded)
  - 1 user message
  - No errors
  - No warnings
  - 1 info
  - No verbose
- Log Entry:** The first log entry is an array, displayed as `Array(0)`, with the source `script-2.js:3`. A blue arrow points to the log entry.
- Settings Panel:** On the right, there are checkboxes for various console settings:
  - ☐ Hide network
  - ☒ Log XMLHttpRequest
  - ☐ Preserve log
  - ☒ Eager evaluation
  - ☐ Selected context
  - ☐ Autocomplete from
  - ☐ Group similar
  - ☒ Evaluate triggers

# Arrays

De forma similar a los objetos, los arrays pueden construirse de base, o se pueden ir modificando posteriormente.

The screenshot shows a web browser's developer console with the following content:

```
index.html JS script-2.js
daw-js > JS script-2.js > ...
1 const myArray = ['Audi', 'Seat', 'Peugeot']
2
3 console.dir(myArray);
4
5
6
7
8
9
10
11
12
13
```

The console also displays a message log on the right side, showing the following information:

- 1 message
- 1 user message
- No errors
- No warnings
- 1 info
- No verbose

The details of the message show an `Array(3)` object with the following properties:

- `0: "Audi"`
- `1: "Seat"`
- `2: "Peugeot"`
- `length: 3`
- `__proto__: Array(0)`

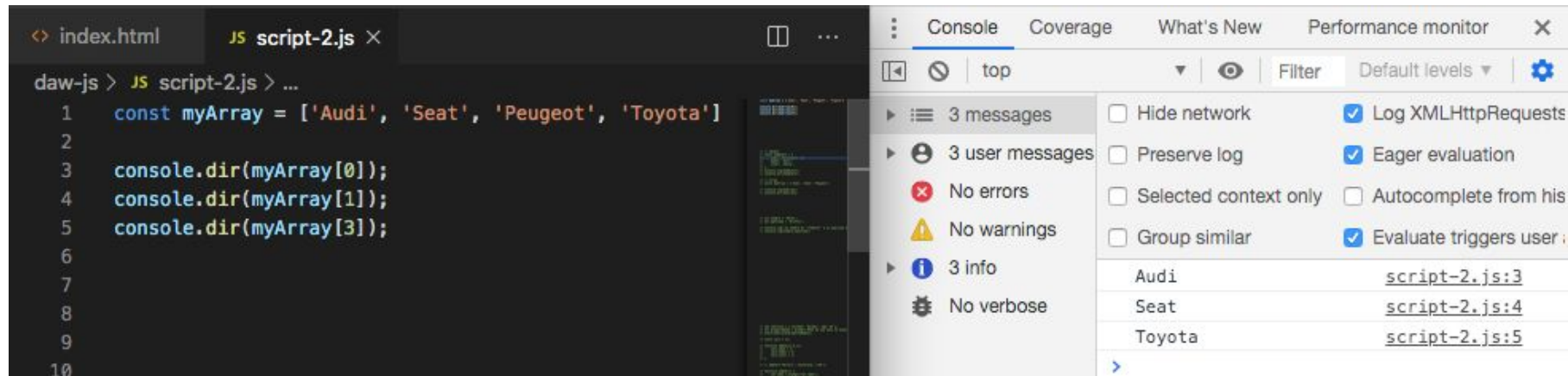
## Consultar

Para acceder a uno de los valores del array, debe accederse por su posición.

**Importante:** tener en cuenta que **los arrays empiezan a contar desde cero**. Eso significa que el **primer** elemento de un array **ocupa la posición 0**, el segundo la 1, y así sucesivamente.



# Arrays



The screenshot shows a web browser's developer console with the following content:

**Code Editor:**

```
1 const myArray = ['Audi', 'Seat', 'Peugeot', 'Toyota']
2
3 console.dir(myArray[0]);
4 console.dir(myArray[1]);
5 console.dir(myArray[3]);
6
7
8
9
10
```

**Console Output:**

3 messages

- 3 user messages
- No errors
- No warnings
- 3 info
- No verbose

**Log Details:**

Message	Source
Audi	script-2.js:3
Seat	script-2.js:4
Toyota	script-2.js:5

## Modificar

Para modificar un valor del array, hay que acceder a él por su posición y asignarle un nuevo valor.

```
myArray[2] = nuevo_valor;
```

# Arrays

The screenshot shows a web browser's developer console with a JavaScript file named `script-2.js` open. The code in the file is as follows:

```
1 var fruits = ["Banana", "Orange", "Apple", "Mango"];
2 fruits[1] = "Strawberry"
3 console.dir(fruits);
4
5
6
7
8
9
10
11
12
```

The console shows the execution of the script. The `console.dir(fruits);` statement has been executed, and the resulting array is displayed in the console's log. The array is an `Array(4)` with the following elements:

- 0: "Banana"
- 1: "Strawberry"
- 2: "Apple"
- 3: "Mango"

The array's `length` property is 4, and its `__proto__` is `Array(0)`. The console also shows a message box with the text "1 message", "1 user message", "No errors", "No warnings", "1 info", and "No verbose".

## Añadir

Para añadir un elemento a un array, se usa la función **push()**.

```
myArray.push('nuevo_elemento')
```

# Arrays

The screenshot shows a web browser's developer console with a JavaScript file named `script-2.js` open. The code in the file is as follows:

```
1 var fruits = ["Banana", "Orange", "Apple", "Mango"];
2 fruits.push("Lemon");
3 console.dir(fruits);
4
5
6
7
8
9
10
11
12
13
14
```

The console shows the output of `console.dir(fruits)` as an `Array(5)` object. The array contains the following elements:

- 0: "Banana"
- 1: "Orange"
- 2: "Apple"
- 3: "Mango"
- 4: "Lemon"

The array's `length` property is 5, and its `__proto__` is `Array(0)`. The console also shows a message box with the text "1 message" and a user message "1 user message". The console settings are visible on the right, showing options like "Hide network", "Preserve log", "Selected context only", "Group similar", "Log XMLHttpRequests", "Eager evaluation", "Autocomplete from his", and "Evaluate triggers user".

## Eliminar

Para eliminar un elemento a un array, se usa la función **delete** y se indica la **posición del elemento** que se quiere eliminar.

Importante tener en cuenta que el primer elemento es el 0.

Éste método puede dejar huecos indefinidos dentro del array. Más adelante veremos otras formas de eliminar elementos.

```
delete myArray[2]
```

# Arrays

The screenshot shows a web browser's developer console with the following elements:

- Code Editor:** Contains the following JavaScript code:

```
1 var fruits = ["Banana", "Orange", "Apple", "Mango"];
2 delete fruits[2];
3 console.dir(fruits);
```
- Console:** Displays the output of `console.dir(fruits)` as an `Array(4)` object. The array contains the following values:
  - 0: "Banana"
  - 1: "Orange"
  - 3: "Mango"
  - length: 4
  - \_\_proto\_\_: Array(0)
- Settings:** The console settings are visible on the right, showing options like "Log XMLHttpRequests", "Eager evaluation", and "Evaluate triggers user".

# Propiedades de Array



## length

Retorna la longitud de un array

```
const fruits = ["Banana", "Orange", "Apple"];  
fruits.length;  
  
// 3
```

## prototype

Permite añadir nuevos métodos y propiedades a un objeto Array.  
En el siguiente ejemplo se crea un nuevo método **myUcase** para convertir el array en mayúsculas

```
Array.prototype.myUcase = function() {...}
```

```
Array.prototype.myUcase = function() {  
    for (let i = 0; i < this.length; i++) {  
        this[i] = this[i].toString().toUpperCase();  
    }  
};
```

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.myUcase();  
// ["BANANA", "ORANGE", "APPLE", "MANGO"]
```

# Métodos de Array

Hemos visto cómo consultar, añadir, modificar y eliminar elementos de un array, pero podemos hacer otras operaciones.

En este apartado veremos algunos métodos muy útiles para trabajar con arrays, pero no todos.

Se puede consultar la documentación completa en [MDN Web Docs](#) o [w3schools](#), entre otros sitios.

## **join()**

Convierte un array en un string, uniendo todos los elementos y usando como separador el caracter entrado por parámetro

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.join(" | ");  
  
// Banana | Orange | Apple | Mango
```

## **pop()**

Elimina el último elemento de un array.

Retorna dicho elemento.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let lastFruit = fruits.pop();  
  
// ["Banana", "Orange", "Apple"]  
  
// lastFruit = "Mango"
```

## push()

**Añade** un nuevo elemento **al final** de un array.

Retorna la nueva longitud del array.

```
const fruits = ["Banana", "Orange", "Apple"];  
let newFruit = fruits.push("Mango");  
  
// ["Banana", "Orange", "Apple", "Mango"]  
  
// newFruit = 4
```



## **shift()**

Elimina el primer elemento de un array y reposiciona el resto  
Retorna el elemento que se ha eliminado.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let firstFruit = fruits.shift();  
  
// ["Orange", "Apple", "Mango"]  
  
// firstFruit = "Banana"
```

## unshift()

Añade un nuevo elemento en primera posición de un array y reposiciona el resto.

Retorna la nueva longitud del array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let y = fruits.unshift("Kiwi");  
  
// ["Kiwi", "Banana", "Orange", "Apple", "Mango"]  
  
// y = 5
```

## **splice()**

Añade uno o varios elementos en la posición indicada.

Puede eliminar (sobrescribir) los elementos anteriores o insertar los nuevos.

El primer parámetro define la posición donde insertar la nueva entrada

El segundo parámetro define cuántos elementos hay que eliminar

El resto de parámetros son los elementos a insertar

Retorna un array con los elementos eliminados

## **splice()**

Insertar 2 elementos a partir de la 3a posición

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(3, 0, "Lemon", "Kiwi");  
  
// ["Banana", "Orange", "Apple", "Lemon", "Kiwi", "Mango"]
```

## **splice()**

Reemplazar 2 elementos a partir de la 2a posición

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Melon"];  
fruits.splice(2, 2, "Lemon", "Kiwi");  
  
// ["Banana", "Orange", "Lemon", "Kiwi", "Melon"]
```

## **splice()**

Eliminar el elemento de la 2a posición

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Melon"];  
  
let deleted = fruits.splice(2, 1);  
  
// ["Banana", "Orange", "Mango", "Melon"]  
  
// deleted = ["Apple"]
```

## **concat()**

Crea un nuevo array a partir de 2 arrays existentes.

Los arrays originales no se ven afectados

```
const fruits = ["Banana", "Orange"];  
const colors = ["Blue", "Black"];  
const mix = fruits.concat(colors);  
// mix = ["Banana", "Orange", "Blue", "Black"]
```

## **slice()**

Crea un nuevo array cortando un array existente a partir de la posición indicada.

El array original no se ve afectado.

```
const colors = ["Orange", "Black", "Red", "Green", "Blue"];  
const rgb = colors.slice(2);  
  
// rgb = ["Red", "Green", "Blue"]
```



## indexOf()

Devuelve la posición (índice) del elemento indicado.

```
const colors = ["Orange", "Black", "Red", "Green", "Blue"];  
let pos = colors.indexOf("Black");  
  
// pos = 1
```

## **indexOf() + splice()**

Combinación muy útil si queremos eliminar un elemento en concreto pero desconocemos su posición en el array.

En este ejemplo, eliminaremos “Red” del array.

```
const colors = ["Orange", "Black", "Red", "Green", "Blue"];  
let pos = colors.indexOf("Red");  
colors.splice( pos, 1)  
// ["Orange", "Black", "Green", "Blue"]
```

<https://developer.mozilla.org/en-US/docs/Web/API/console>

[https://www.w3schools.com/jsref/jsref\\_obj\\_array.asp](https://www.w3schools.com/jsref/jsref_obj_array.asp)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

```
<?php language_attributes(); ?>
<?php bloginfo( 'charset' ); ?>
<?php wp_title( '|', true, 'right' ); ?>
<?php wp_title( 'profile', true, 'right' ); ?>
<?php bloginfo( 'pingback_url' ); ?>
<?php fruitful_get_favicon(); ?>
<?php echo get_template_directory_uri(); ?>
<?php wp_head(); ?>
</head>
<?php body_class(); ?>
<div id="page-header" class="hfeed site">
<?php
$theme_options = fruitful_get_theme_options();
$logo_pos = $theme_options['logo_position'];
if (isset($theme_options['logo_position']))
$logo_pos = esc_attr($theme_options['logo_position']);
</div>
```

## Arrays (II)

¿Sabías que...?

Los Furbys tienen los ojos en la parte delantera de la cabeza, lo que los convierte en depredadores.



# Métodos

## **reverse()**

reordena un array al revés de su orden original

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.reverse()
```

```
// ["Mango", "Apple", "Orange", "Banana"]
```

## **sort()**

ordena un array alfabéticamente en orden ascendente ( A  $\rightarrow$  Z)

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort()
```

```
// ["Apple", "Banana", "Mango", "Orange"]
```



## **sort()**

no sirve para ordenar números, ya que convierte los ítems en strings y los ordena alfabéticamente.

```
const nums = [30, 100, 2];  
nums.sort()
```

```
// [100, 2, 30]
```

## sort()

Acepta un parámetro con una función de **comparación** de 2 elementos

compara( a, b ) → Si el resultado es **> 0**, ordena **b delante de a**

compara( a, b ) → Si el resultado es **< 0**, ordena **a delante de b**

compara( a, b ) → Si el resultado es **= 0**, considera los valores **iguales**

```
const nums = [30, 100, 2];
```

```
nums.sort(function(a,b){ return a - b})
```

```
// [2, 30, 100]
```

```
nums.sort(function(a,b){ return b - a})
```

```
// [100, 30, 2]
```

## map()

Aplica una función a cada uno de los elementos del array, por orden.  
Devuelve un nuevo array con el resultado.  
El array original no sufre cambios.

```
const nums = [2,3,5];
```

```
ES5: const double = nums.map( function(x){ return x * 2} )
```

```
ES6: const double = nums.map( x => x * 2 )
```

```
// double = [4, 6, 10]
```

## **filter()**

Devuelve un array con los elementos que cumplen una condición concreta.

Devuelve un nuevo array con el resultado.

El array original no sufre cambios.

```
const ages = [12, 23, 15, 32];
```

```
ES5: const teens = ages.filter( function(x){ return x < 20} )
```

```
ES6: const teens = ages.filter( x => x < 20 )
```

```
// teens = [12, 15]
```

## find()

Devuelve el **primer** elemento que cumple una condición concreta.

```
const ages = [12, 18, 23, 15, 32];
```

```
ES5: const adult = ages.find( function(x){ return x > 18} )  
// adult = 23
```

```
ES6: const adult = ages.find( x => x >= 18 )  
// adult = 18
```

## findIndex()

Devuelve la **posición** (index) del **primer** elemento que cumple una condición concreta.

```
const ages = [12, 18, 23, 15, 32];
```

```
ES5: const adult = ages.findIndex( function(x){ return x > 18} )  
// adult = 2
```

```
ES6: const adult = ages.findIndex( x => x >= 18 )  
// adult = 1
```

## fill()

**Rellena** todos los elementos de un array con un **valor estático**.  
El array original se **sobreescribe**.

```
const letters = ["A", "B", "C", "D"];  
letters.fill("Z");  
  
// letters = ["Z", "Z", "Z", "Z"]
```

## every()

**Comprueba** que **todos** los elemento del array cumplan una condición. Devuelve **true** o **false** según si se cumple en todos o no.

```
const nums1 = [2, 3, 4, 6];  
const nums2 = [2, 8, 4, 6];
```

```
ES5: let isEven = nums1.every( function(x){ return (x%2)== 0 })  
// isEven = false
```

```
ES6: let isEven = nums2.every( x => (x%2)==0 )  
// isEven = true
```



## some()

**Comprueba** si **algún** elemento del array cumple una condición.

Aplica la comprobación para cada elemento del array.

Si **uno o más** elemento(s) cumple(n) la condición, devuelve **true**.

Si **ningún** elemento cumple la condición, devuelve **false**.

```
const ages1 = [12, 23, 14, 16];  
const ages2 = [12, 13, 14, 16];
```

```
ES5: let someAdult = ages1.some( function(x){ return x >= 18 } )  
// someAdult = true
```

```
ES6: let someAdult = ages2.some( x => x >= 18 )  
// someAdult = false
```

# Iteraciones

## bucle for

Los arrays se pueden iterar mediante bucles. Una forma de hacerlo es usando un bucle **for**

El bucle for recibe 3 parámetros, y ejecuta el código tantas veces como se le defina.

```
for (param1; param2; param3) {  
    ...  
}
```

param1: se ejecuta 1 vez. Se usa para inicializar el contador de veces que se ejecutará

param2: condición. Mientras se cumpla la condición, el bucle se irá ejecutando

param3: actualización del contador. Se ejecuta al final de cada iteración

Este es el ejemplo más clásico de un bucle. Se inicializa  $i$  a 0 y se empieza.

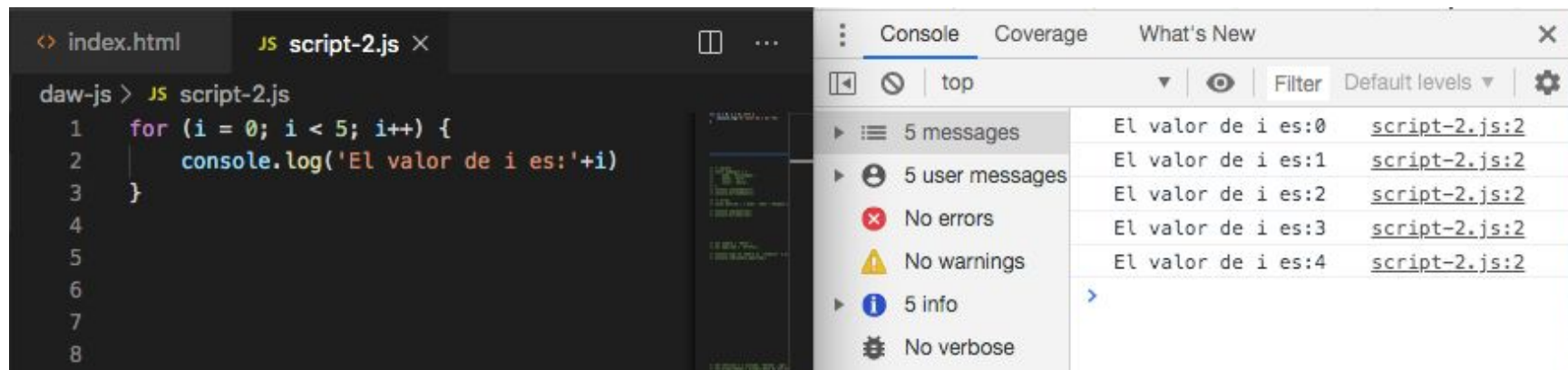
Antes de cada iteración se comprueba la condición: si  $i$  es menor que 5, se ejecuta la iteración.

El código entre corchetes se ejecuta una vez por cada iteración.

Al final de cada iteración, se incrementa en 1 el valor de  $i$ .

Antes de empezar la siguiente iteración, se comprueba la condición: si se cumple, se ejecuta el código. Y así sucesivamente hasta que la condición deja de cumplirse y termina el bucle

# JavaScript



The screenshot shows a web browser's developer console with the following components:

- Code Editor:** Displays a JavaScript file named `script-2.js` with the following code:

```
1 for (i = 0; i < 5; i++) {  
2   console.log('El valor de i es:'+i)  
3 }  
4  
5  
6  
7  
8
```
- Console Panel:** Shows a summary of messages:
  - 5 messages (expanded)
  - 5 user messages
  - No errors
  - No warnings
  - 5 info
  - No verbose
- Message List:** Displays five log messages from `script-2.js:2`:
  - El valor de i es:0
  - El valor de i es:1
  - El valor de i es:2
  - El valor de i es:3
  - El valor de i es:4

Esta misma lógica se puede aplicar para iterar por todos los elementos de un array.

Se empieza inicializando **i** a cero.

El bucle debe finalizar cuando se hayan recorrido todos los elementos del array. Para saber su número, podemos hacer uso de **length**.

Podemos usar el valor de **i**, que va incrementando en cada iteración para acceder a los elementos del array por su posición.

# Iteraciones

The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the output of a JavaScript script. The script defines an array of car brands and iterates through them, logging each brand to the console. The output shows four messages: 'Audi', 'Seat', 'Peugeot', and 'Ferrari', each followed by the file path 'script-2.js:4'. The console also shows that there are no errors or warnings, and four info messages.

```
index.html x JS script-2.js x
daw-js > JS script-2.js > ...
1 const cars = ['Audi', 'Seat', 'Peugeot', 'Ferrari']
2
3 for (i = 0; i < cars.length; i++) {
4   console.log(cars[i])
5 }
6
7
```

Console Coverage What's New

top

4 messages

4 user messages

No errors

No warnings

4 info

Audi	script-2.js:4
Seat	script-2.js:4
Peugeot	script-2.js:4
Ferrari	script-2.js:4

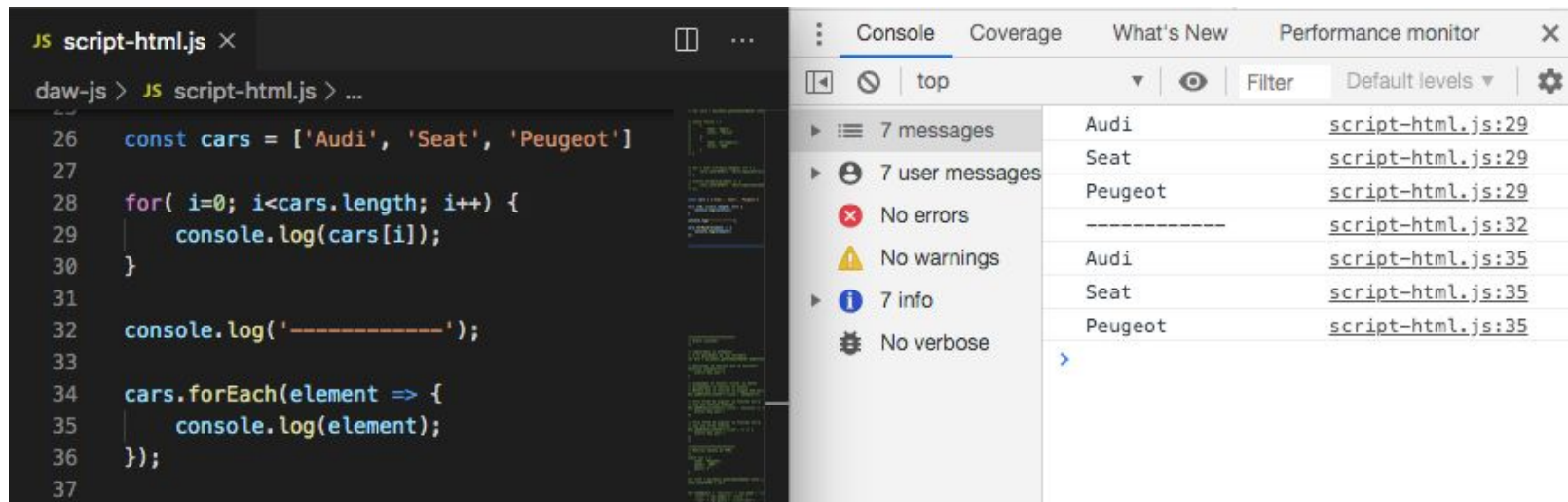
## **bucle foreach**

es otra forma de crear bucles. Usando foreach no hay que preocuparse de controlar el número de iteraciones, ya que automáticamente se ejecutará para cada uno de los elementos a iterar (de ahí el nombre *for each*)

Se implementa como una arrow function, y para acceder a cada uno de los elementos se hace con la palabra clave que definamos antes de la flecha (en el ejemplo, es "element")



# Iteraciones



The screenshot shows a code editor with a JavaScript file named `script-html.js`. The code defines an array `cars` with the values `'Audi'`, `'Seat'`, and `'Peugeot'`. It then uses a `for` loop to iterate over the array, logging each element to the console. Additionally, it logs a separator line `-----` and then iterates over the array again using `cars.forEach`, logging each element.

```
26 const cars = ['Audi', 'Seat', 'Peugeot']
27
28 for( i=0; i<cars.length; i++) {
29   console.log(cars[i]);
30 }
31
32 console.log('-----');
33
34 cars.forEach(element => {
35   console.log(element);
36 });
37
```

The console output on the right shows the following messages:

- 7 messages
- 7 user messages
- No errors
- No warnings
- 7 info
- No verbose

Message	Location
Audi	<a href="#">script-html.js:29</a>
Seat	<a href="#">script-html.js:29</a>
Peugeot	<a href="#">script-html.js:29</a>
-----	<a href="#">script-html.js:32</a>
Audi	<a href="#">script-html.js:35</a>
Seat	<a href="#">script-html.js:35</a>
Peugeot	<a href="#">script-html.js:35</a>

[https://www.youtube.com/watch?v=Ah7-PPjQ5Ls&ab\\_channel=midudev](https://www.youtube.com/watch?v=Ah7-PPjQ5Ls&ab_channel=midudev)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/sort](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort)

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Remainder>

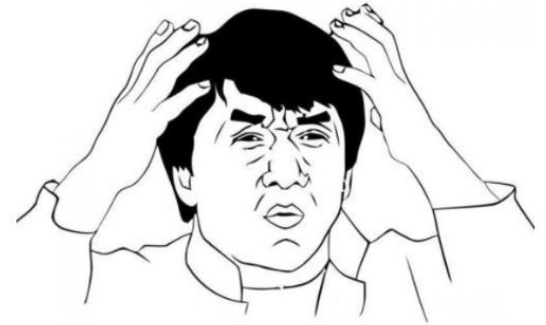


# DOM

## Interacción JS con HTML y CSS

¿Sabías que...?

**Las bananas son curvadas porque crecen hacia el  
sol**



# HTML + JS

JavaScript sirve para ampliar las funcionalidades de la parte del HTML.

Hasta ahora sólo hemos visto como tratar la funciones, variables y objetos únicamente desde JavaScript.

En este apartado veremos como pueden interactuar JS y HTML

Desde HTML se pueden invocar funciones de JavaScript, principalmente desde botones y formularios.

Desde JavaScript podemos modificar elementos del HTML.

## Capturar elemento HTML por ID: `document.getElementById()`

Esta instrucción sirve para encontrar un elemento en el HTML con un **id** concreto.

**document:** hace referencia a todo el documento HTML

**getElementById():** busca dentro del documento un elemento que coincida con el id que se le pasa por parámetro. De ahí la importancia de no tener id repetidos en el HTML, ya que sino, esta función no sabría sobre cuál de ellos actuar.

Toda esta instrucción sirve para encontrar un elemento en el HTML. Una vez encontrado, podemos guardar la referencia de éste elemento en una variable.

```
var myElement = document.getElementById('myButton');
```

## Event Listener: `addEventListener()`

Un **event listener** se utilitza para que un elemento del HTML “esté atento” a que ocurra un evento en concreto. Cuando dicho evento ocurre, se ejecuta una función.

El caso más típico es el “click” del ratón. Es decir, podemos programar un elemento para que se ejecute una función de JavaScript sólo en el momento en que el usuario haga clic con el ratón sobre éste.

Para ello, únicamente necesitamos asignar un **id** al elemento HTML y utilizar las funciones **`document.getElementById()`** y **`addEventListener()`**



&lt;&gt; index.html x

daw-js &gt; &lt;&gt; index.html &gt; html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <h1 id="title">JavaScript</h1>
9   <button id="myButton">Click me!</button>
10
11   <script src="script-events.js"></script>
12 </body>
13 </html>
```

JS script-events.js x

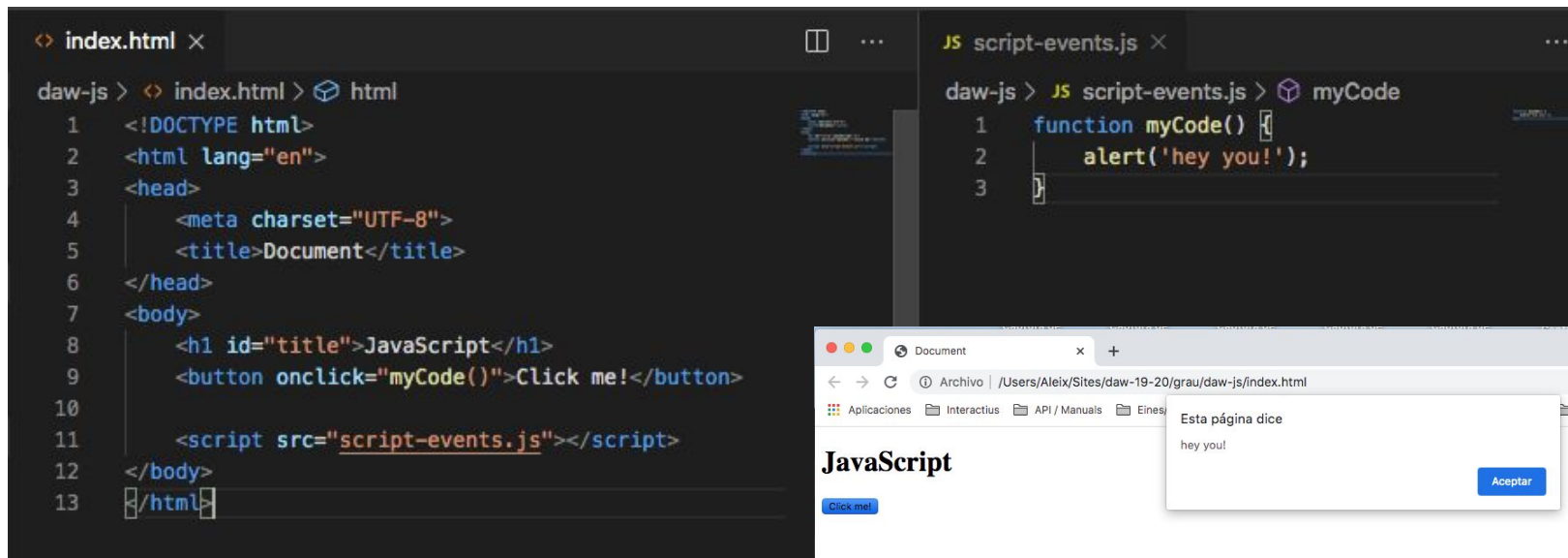
daw-js &gt; JS script-events.js &gt; btn.addEventListener('click') callb

```
1 // capturamos el elemento
2 // y lo guardamos en una variable
3 var btn = document.getElementById('myButton');
4
5 // definimos la función que se ejecutará
6 function showAlert(){
7   alert('hey you!')
8 }
9
10 // asignamos el evento 'click' al botón
11 // asignamos la función al evento
12 // Nótese que la función se asigna SIN paréntesis
13 btn.addEventListener('click', showAlert);
14
15 // otra forma de asignar la función sería
16 // con una función anónima
17 btn.addEventListener('click', function () {
18   alert('hey you!')
19 })
20
21 // otra forma de asignar la función sería
22 // con una arrow función
23 btn.addEventListener('click', () => {
24   alert('hey you!')
25 })
```

## **onclick**

Este evento lo podemos asignar a un botón `<button>` junto a una función directamente desde el HTML.

Cuando se haga click en el botón, se ejecutará la función de JavaScript. De este modo, se asigna el Event Listener en el elemento directamente desde HTML



The image shows a code editor with two files open: `index.html` and `script-events.js`. The `index.html` file contains the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <h1 id="title">JavaScript</h1>
9   <button onclick="myCode()">Click me!</button>
10
11   <script src="script-events.js"></script>
12 </body>
13 </html>
```

The `script-events.js` file contains the following JavaScript code:

```
1 function myCode() {
2   alert('hey you!');
3 }
```

Below the code editor, a web browser window is shown. The browser's address bar displays the URL `/Users/Aleix/Sites/daw-19-20/grau/daw-js/index.html`. The page content includes the heading `JavaScript` and a button labeled `Click me!`. A dialog box is open over the button, displaying the message `Esta página dice hey you!` and an `Aceptar` button.

## Modificar el HTML: innerHTML

Desde JavaScript podemos modificar el HTML.

Para ello, primero necesitamos obtener el elemento que queremos modificar ( podemos usar `document.getElementById()` ).

Y luego, se asigna el nuevo contenido usando la instrucción **innerHTML**, asignando con un `=` el nuevo contenido.

```
index.html x
daw-js > index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6 </head>
7 <body>
8   <h1 id="title">JavaScript</h1>
9   <!-- <button id="myButton">Click me!</button> -->
10
11   <div id="info"></div>
12
13   <script src="script-events.js"></script>
14 </body>
15 </html>

JS script-events.js x
daw-js > JS script-events.js > ...
1 const car = {
2   name: 'Peugeot',
3   model: '307',
4   doors: 5
5 }
6
7 var info = document.getElementById('info');
8 info.innerHTML = car;
9
10
11 var newObject = '<ul><li>' + car.name + '</li>' +
12   '<li>' + car.model + '</li>' +
13   '<li>' + car.doors + '</li></ul>';
14
15 info.innerHTML = newObject;
16
```

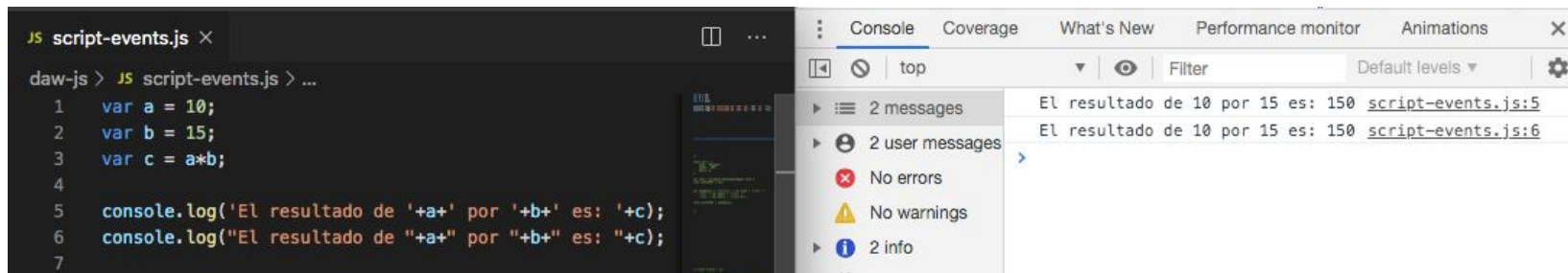
## Interpolación de variables

Esta es una nueva forma de combinar las variables con textos o strings para mostrar resultados.

Se puede utilizar tanto para mostrar información a través de **console.log()** o para mostrar información utilizando **innerHTML**.

Hemos visto que se pueden concatenar strings y variables usando + y escribiendo los string entre comillas simples o dobles

```
console.log('El resultado de '+a+' por '+b+' es: '+c);
```



The image shows a web browser's developer console with the following content:

```
JS script-events.js ×
daw-js > JS script-events.js > ...
1  var a = 10;
2  var b = 15;
3  var c = a*b;
4
5  console.log('El resultado de '+a+' por '+b+' es: '+c);
6  console.log("El resultado de "+a+" por "+b+" es: "+c);
7
```

The console also displays a list of messages:

- 2 messages
- 2 user messages
- No errors
- No warnings
- 2 info

The messages are:

- El resultado de 10 por 15 es: 150 [script-events.js:5](#)
- El resultado de 10 por 15 es: 150 [script-events.js:6](#)

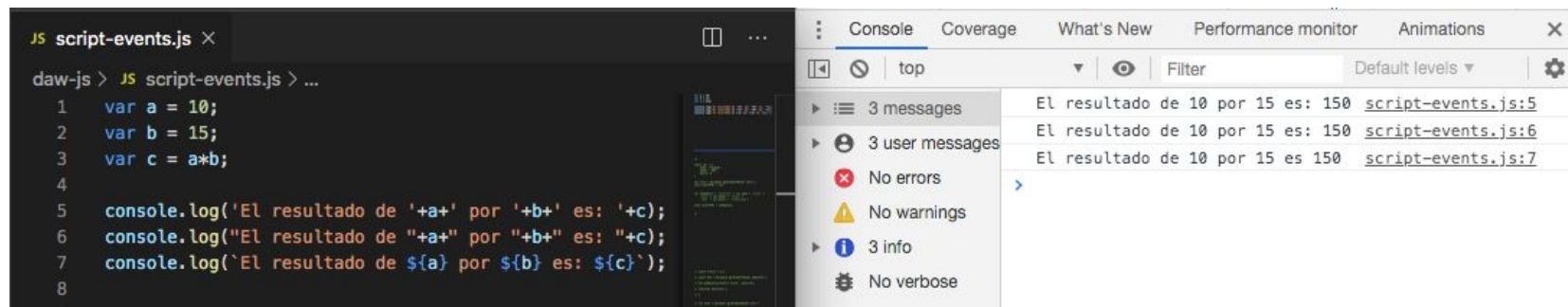
## Interpolación de variables

Otra forma de hacerlo es escribiendo todo el texto y variables de un tirón, todo entre ` (acento abierto), y escribir las variables dentro de la expresión `${ }`

```
console.log( `El resultado de ${a} por ${b} es: ${c}` )
```

Es una forma más limpia, ya que no hay que estar concatenando `'` y `+` para insertar variables, pero es importante no confundirse y ponerlo todo entre acento abierto en lugar de comilla simple o doble





The screenshot shows a code editor with a file named `script-events.js`. The code defines variables `a` and `b`, calculates `c = a * b`, and logs three messages to the console. The console is open, showing the output of these log statements. The messages are: 'El resultado de '+a+' por '+b+' es: '+c);', 'El resultado de "+a+" por "+b+" es: "+c);', and 'El resultado de `\${a}` por `\${b}` es: `\${c}`)'. The console also shows that there are 3 messages, 3 user messages, no errors, no warnings, 3 info messages, and no verbose messages.

```
JS script-events.js x
daw-js > JS script-events.js > ...
1  var a = 10;
2  var b = 15;
3  var c = a*b;
4
5  console.log('El resultado de '+a+' por '+b+' es: '+c);
6  console.log("El resultado de "+a+" por "+b+" es: "+c);
7  console.log(`El resultado de ${a} por ${b} es: ${c}`);
8
```

Console Coverage What's New Performance monitor Animations

top Filter Default levels

- 3 messages
  - El resultado de 10 por 15 es: 150 [script-events.js:5](#)
  - El resultado de 10 por 15 es: 150 [script-events.js:6](#)
  - El resultado de 10 por 15 es 150 [script-events.js:7](#)
- 3 user messages
- No errors
- No warnings
- 3 info
- No verbose

## Tagged templates

En este ejemplo vamos a crear un array de objetos, insertar nuevos elementos y mostrar la información en una [tabla HTML](#) .

```
<body>
  <h1 id="title">JavaScript</h1>

  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Color</th>
      </tr>
    </thead>
    <tbody id="info">
    </tbody>
  </table>

  <button id="btnAdd">Add Fruit</button>
  <button id="btnClear">Clear All</button>

  <script src="script-html.js"></script>
</body>
```

```
// tabla HTML dónde irá la información
var info = document.getElementById('info')
// botón para añadir nuevo elemento (fila) a la tabla
var btn = document.getElementById('btnAdd')
// botón para limpiar la tabla
var btnClear = document.getElementById('btnClear')

// asignar funciones a los botones
btn.addEventListener('click', addFruit);
btnClear.addEventListener('click', clearAll);

// array de objetos
// cada objeto representa una fruta con nombre y color
const fruits = [
  {
    name: 'Apple',
    color: 'Yellow'
  },
  {
    name: 'Strawberry',
    color: 'Red'
  }
]
```

Podemos intercalar tags HTML y variables. A ese formato se le llama *tagged templates*.

```
80
81 // creamos una variable con contenido vacío
82 let data = '';
83
84 // recorremos el array de frutas
85 // para cada elemento del array (es decir, por para fruta) creamos
86 // una nueva fila para la tabla con la información de la fruta
87 // Toda la información se va concatenando en la variable "data". Es decir,
88 // no se sobrescribe la variable, sino que se va añadiendo la información
89 // con la instrucción +=
90 fruits.forEach(fruit => {
91   data += `<tr>
92     <td>${fruit.name}</td>
93     <td>${fruit.color}</td>
94   </tr>`;
95 });
96
97 // una vez tenemos toda la información,
98 // sobrescribimos el contenido de la tabla
99 info.innerHTML = data;
```

```
// añadir item a la tabla
function addFruit() {
  // capturamos el nombre y color de la fruta desde un prompt
  let fruitName = prompt('Fruit name');
  let fruitColor = prompt('Fruit color');
  // creamos un nuevo objeto con la nueva fruta
  let newFruit = {
    name: fruitName,
    color: fruitColor
  }
  // añadimos el nuevo objeto a la tabla
  fruits.push(newFruit);

  // recorremos todo el array de frutas y sobrescribimos la tabla
  let data = '';

  fruits.forEach(fruit => {
    data += `<tr>
      <td>${fruit.name}</td>
      <td>${fruit.color}</td>
    </tr>`;
  });

  info.innerHTML = data;
}

// limpiar tabla
function clearAll() {
  // sobrescribimos la tabla con un string vacío
  info.innerHTML = '';
}
```

Acceso	Sintaxis
ID (único)	<code>document.getElementById('id')</code>
Atributo Name (múltiples)	<code>document.getElementsByName('name')</code>
Tag HTML (múltiples)	<code>document.getElementsByTagName('tagHTML')</code>
Clase CSS (múltiples)	<code>document.getElementsByClassName('classname')</code>
Selector CSS (único)	<code>document.querySelector('selector CSS')</code>
Selector CSS (múltiples)	<code>document.querySelectorAll('selector CSS')</code>

**IMPORTANTE:** los selectores preparados para que devuelven múltiples valores, devuelven siempre un array, colección o lista, aunque el resultado sólo sea un elemento.

**IMPORTANT:** los selectores preparados para que devuelven múltiples valores, devuelven siempre un array, colección o lista, aunque el resultado sólo sea un elemento.

Se puede convertir el resultado a un Array con la función `Array.from`

```
const elems = document.querySelectorAll('.elem');  
const elemsArray = Array.from(elems);
```

HTML	Ejemplo
<code>&lt;div id="myDiv"&gt;</code>	<code>document.getElementById('myDiv')</code>
<code>&lt;input type="radio" name="colors"&gt;</code>	<code>document.getElementsByName('colors')</code>
<code>&lt;label&gt;</code>	<code>document.getElementsByTagName('label')</code>
<code>&lt;button class="cta"&gt;</code>	<code>document.getElementsByClassName('cta')</code>
<code>&lt;h1 class="title"&gt;</code>	<code>document.querySelector('.title')</code>
<code>&lt;ul&gt;&lt;li&gt;&lt;/li&gt;&lt;li&gt;&lt;/li&gt;...&lt;/ul&gt;</code>	<code>document.querySelectorAll('ul li')</code>

## Creación de nuevos elementos

### `document.createElement`

Desde JavaScript podemos crear y añadir nuevos elementos en el DOM. Para crear un nuevo elemento podemos usar el método **`createElement()`**

```
let newDiv = document.createElement("DIV")
```



## innerText / innerHTML

Una vez tenemos creado el nuevo elemento, le podemos añadir contenido (texto plano u otros elementos HTML). Para ello podemos usar los métodos **innerText** (para texto plano) o **innerHTML** (para otros elementos HTML)

```
let newDiv = document.createElement("DIV")
```

```
newDiv.innerHTML("<p>Hello world</p>")
```

```
newDiv.innerText("Hello")
```

## appendChild

Finalmente podemos añadir el nuevo elemento al DOM. Podemos añadirlo directamente al **body** o podemos seleccionar un elemento existente ( por ejemplo con `document.getElement...` ) y añadirsele:

```
let newDiv = document.createElement("DIV")
newDiv = document.innerHTML("<p>Hello world</p>")
```

```
//añadimos newDiv al body
document.body.appendChild(newDiv)
```

```
//añadimos newDiv a un elemento existente
let elem = document.getElementById("myElement")
elem.appendChild(newDiv)
```

## Insertar elementos

### document.insertAdjacent

Desde JavaScript podemos insertar elementos dentro o adyacentes a otros elementos del DOM.

```
element.insertAdjacentHTML(position, text);
```

El primer parámetro indica la posición dónde se quiere insertar el elemento, y el segundo es el elemento a insertar. Podemos elegir 4 posiciones distintas:

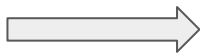
```
<!-- beforebegin -->  
<p>  
  <!-- afterbegin -->  
  foo  
  <!-- beforeend -->  
</p>  
<!-- afterend -->
```

## Insertar elementos

### `document.insertAdjacent( position, text )`

```
subject.insertAdjacentHTML('afterbegin', '<strong>inserted text</strong>');
```

```
<p>  
  foo  
</p>
```



```
<p>  
  <strong>inserted text</strong>  
  foo  
</p>
```

## data attributes

En los tags HTML existen una serie de atributos “de serie” como *id*, *class*, *href*, *src*... pero también podemos crear nuestros propios atributos.

Para hacerlo, debemos añadirlo con el prefijo “*data-*” y el nombre que queramos darle.”

Por ejemplo, podemos crear un atributo “color” en un <div>.

```
<div data-color="blue">
```

Desde JavaScript podemos acceder a los data attributes (leer y modificar) usando ***dataset***

```
<div id="myDiv" data-color="blue">
```

```
let myDiv = document.getElementById("myDiv");  
console.log(myDiv.dataset.color);
```

```
// blue
```

```
myDiv.dataset.color = "red";  
console.log(myDiv.dataset.color)
```

```
// red
```

Los *data attributes* pueden ser útiles para guardar **información temporal** o datos dinámicos mientras se ejecuta la aplicación.

Hay que tener en cuenta que esos datos **no son persistentes**, y si los modificamos, al refrescar la página volverán a tener su valor inicial. Pero los podemos usar para guardar o asociar valores a un elemento.

## **console.dir**

Console log es útil para debugar y mostrar mensajes por consola, pero si queremos ver información más detallada es más útil usar **console.dir**.

De esta forma, si hacemos console.dir de un elemento (por ejemplo, un <div>) podremos ver **toda su información** (id, clases, atributos, elementos padre e hijos... )



## console.dir

Desde JavaScript podemos acceder a cualquiera de esas propiedades.

Por ejemplo, podemos ver los data attributes de un elemento, en la clave *dataset*.

```
<div id="myDiv" data-color="red">
```

```
let myDiv = document.getElementById("myDiv");
```

```
console.dir(myDiv);
```

```
childNodes: 2  
▶ childNodes: NodeList(2) [style, img.lnXdpd]  
▶ children: HTMLCollection(2) [style, img.lnXdpd]  
▶ classList: DOMTokenList(2) ['k1zIA', 'rSk4se',  
  className: "k1zIA rSk4se"  
  clientHeight: 92  
  clientLeft: 0  
  clientTop: 0  
  clientWidth: 272  
  contentEditable: "inherit"  
▼ dataset: DOMStringMap  
  color: "red"  
  ▶ [[Prototype]]: DOMStringMap  
  dir: ""  
  draggable: false  
  elementTiming: ""  
  enterKeyHint: ""  
▶ firstChild: style  
▶ firstElementChild: style  
  hidden: false  
  id: ""  
  innerHTML: "<style data-iml=\"1650960848007\">.  
  innerText: ""
```

# CSS + JS

## ClassList

Desde JavaScript se puede acceder a la lista de clases de un elemento y efectuar algunas acciones sobre ella. Algunas de las más útiles son añadir, eliminar y alternar.

## **classList.add**

añade una clase al elemento

```
document.getElementById("id_elemento").classList add("miclase");
```

## **classList.remove**

elimina una clase del elemento

```
document.getElementById("id_elemento").classList remove("miclasse");
```

## **classList.toggle**

alterna una clase: si el elemento no tiene la clase, la añade; si ya la tiene, la elimina

```
document.getElementById("id_elemento").classList toggle ("miclase");
```

En este ejemplo creamos un `<button>` y un `<div>` con un texto. Con CSS definimos 2 estilos para el texto: oculto y visible.

Asociamos una función al botón y al hacer clic alternamos la clase del div para ocultarlo o mostrarlo

```
<style>
  .panel {
    display: none; /* no se muestra */
  }
  .panel.show {
    display: block; /* se muestra */
    width: 350px;
    background-color: #eee;
    border-radius: 5px;
    border: 1px solid;
    padding: 10px;
  }
</style>
</head>

<body>

  <button id="myButton">Click me!</button>
  <div id="text" class="panel">
    <p>Lorem ipsum dolor sit amet, consectetur
      magna aliqua. Ut enim ad minim veniam,
      consequat.</p>
  </div>
```

```
4
5  var btn = document.getElementById('myButton')
6  var text = document.getElementById('text')
7
8
9  btn.addEventListener('click', () => {
10    text.classList.toggle('show');
11  })
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

Click me!

Lorem ipsum dolor sit amet, consectetur  
adipiscing elit, sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua. Ut enim ad  
minim veniam, quis nostrud exercitation ullamco  
laboris nisi ut aliquip ex ea commodo consequat.



## ClassName

Es una instrucción similar a `classList`. Es útil para trabajar **con una sola clase**, en lugar de con toda la lista de clases que puede tener un elemento.

No tiene los métodos `add`, `remove` o `toggle`, pero es útil si queremos reemplazar una clase por otra de nueva. Por ejemplo, si queremos cambiar la clase “red” por “blue”:

1. `<div id="myDiv" class="blue">`
2. `document.getElementById('myDiv').className = 'red';`
3. `<div id="myDiv" class="red">`

## ***style.property***

Con el método **style** podemos modificar directamente los estilos o propiedades CSS del elemento seleccionado, indicando qué propiedad queremos modificar y qué valor queremos darle

```
document.getElementById(id).style.property = new style
```

## ***style.property***

Por ejemplo, podemos cambiar el color del texto de un elemento con el id “demo”:

```
document.getElementById("demo").style.color = "blue";
```

Hay que tener en cuenta que si se trata de una propiedad de más de una palabra (en CSS se escribe con guiones) se deberá usar la nomenclatura tipo *camelCase*

```
document.getElementById("demo").style.backgroundColor = "gold";
```

## style.cssText

Con el método **style** y **cssText** podemos cambiar múltiples valores de CSS. Se pueden usar comillas simples, dobles o *template literals* ( el acento abierto ` como si se usara para interpolar variables )

```
document.getElementById(id).style.cssText = "color:red;  
font-size: 2rem"
```

En este caso, la propiedades de más de una palabra se deben escribir tal y como se hace en CSS, sin *camelCase*.

[w3schools: eventos JavaScript](#)

[w3schools: tablas HTML](#)

[Google Drive: videos ejemplos](#)

[Google Drive: código ejemplos](#)